Learn Linux, 101: The Linux command line

Getting comfortable with GNU and UNIX commands

Ian Shields
Senior Programmer
IBM

31 August 2009
(First published 25 August 2009)

GUIs are fine, but to unlock the real power of Linux®, there's no substitute for the command line. In this article, Ian Shields introduces you to some of the major features of the bash shell, with an emphasis on the features that are important for LPI certification. By the end of this article, you will be comfortable using basic Linux commands like echo and exit, setting environment variables, and gathering system information. [*The first two notes following Listing 8 have been updated to correct the process IDs (PIDs). -Ed.*]

View more content in this series

Overview

This article gives you a brief introduction to some of the major features of the bash shell, and covers the following topics:

- Interacting with shells and commands using the command line
- Using valid commands and command sequences
- Defining, modifying, referencing, and exporting environment variables
- Accessing command history and editing facilities
- Invoking commands in the path and outside the path
- Using man (manual) pages to find out about commands

This article helps you prepare for Objective 103.1 in Topic 103 of the Junior Level Administration (LPIC-1) exam 101. The objective has a weight of 4. The material in this article corresponds to the April 2009 objectives for exam 101. You should always refer to the objectives for the definitive requirements.

The bash shell

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Basics of Linux system administration: Working at the console

The *bash* shell is one of several shells available for Linux. It is also called the *B*ourne-*a*gain *sh*ell, after Stephen Bourne, the creator of an earlier shell (*/bin/sh*). Bash is substantially compatible with sh, but it provides many improvements in both function and programming capability. It incorporates features from the Korn shell (ksh) and C shell (csh), and is intended to be a POSIX-compliant shell.

About this series

This series of articles helps you learn Linux system administration tasks. You can also use the material in these articles to prepare for Linux Professional Institute Certification level 1 (LPIC-1) exams.

See our series roadmap for a description of and link to each article in this series. The roadmap is in progress and reflects the latest (April 2009) objectives for the LPIC-1 exams: as we complete articles, we add them to the roadmap. In the meantime, though, you can find earlier versions of similar material, supporting previous LPIC-1 objectives prior to April 2009, in our LPI certification exam prep tutorials.

Prerequisites

To get the most from the articles in this series, you should have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this article. Sometimes different versions of a program will format output differently, so your results may not always look exactly like the listings and figures shown here.

Before we delve deeper into bash, recall that a *shell* is a program that accepts and executes commands. It also supports programming constructs, allowing complex commands to be built from smaller parts. These complex commands, or *scripts*, can be saved as files to become new commands in their own right. Indeed, many commands on a typical Linux system **are** scripts.

Shells have some *builtin* commands, such as `cd`, `break`, and `exec`. Other commands are *external*.

Shells also use three standard I/O *streams*:

- *stdin* is the *standard input stream*, which provides input to commands.
- *stdout* is the *standard output stream*, which displays output from commands.
- *stderr* is the *standard error stream*, which displays error output from commands.

Input streams provide input to programs, usually from terminal keystrokes. Output streams print text characters, usually to the terminal. The terminal was originally an ASCII typewriter or display terminal, but it is now more often a window on a graphical desktop. More detail on how to redirect these standard I/O streams will be covered in another article in this series.

For the rest of this article, we will assume you know how to get a shell prompt. If you don't, the developerWorks article "Basic tasks for new Linux developers" shows you how to do this and more.

If you are using a Linux system without a graphical desktop, or if you open a terminal window on a graphical desktop, you will be greeted by a prompt, perhaps like one shown in Listing 1.

Listing 1. Some typical user prompts

```
[db2inst1@echidna db2inst1]$
ian@lyrebird:~>
$
```

If you log in as the root user (or superuser), your prompt may look like one shown in Listing 2.

Listing 2. Superuser, or root, prompt examples

```
[root@echidna ~]#
lyrebird:~ #
#
```

The root user has considerable power, so use it with caution. When you have root privileges, most prompts include a trailing pound sign (#). Ordinary user privileges are usually delineated by a different character, commonly a dollar sign ($). Your actual prompt may look different than the examples in this article. Your prompt may include your user name, hostname, current directory, date, or time that the prompt was printed, and so on.

These articles include code examples that are cut and pasted from real Linux systems using the default prompts for those systems. Our root prompts have a trailing #, so you can distinguish them from ordinary user prompts, which have a trailing $. This convention is consistent with many books on the subject. If something doesn't appear to work for you, check the prompt in the example.

Commands and sequences

So now that you have a prompt, let's look at what you can do with it. The shell's main function is to interpret your commands so you can interact with your Linux system. On Linux (and UNIX®) systems, commands have a *command name*, and then *options* and *parameters*. Some commands have neither options nor parameters, and some have one but not the other..

If a line contains a # character, then all remaining characters on the line are ignored. So a # character may indicate a comment as well as a root prompt. Which it is should be evident from the context.

Echo

The `echo` command prints (or echos) its arguments to the terminal as shown in Listing 3.

Listing 3. Echo examples

```
[ian@echidna ~]$ echo Word
Word
[ian@echidna ~]$ echo A phrase
A phrase
[ian@echidna ~]$ echo Where     are   my   spaces?
Where are my spaces?
[ian@echidna ~]$ echo "Here     are   my   spaces." # plus comment
Here     are   my   spaces.
```

In third example of Listing 3, all the extra spaces were compressed down to single spaces in the output. To avoid this, you need to *quote* strings, using either double quotes (") or single quotes ('). Bash uses *white space*, such as blanks, tabs, and new line characters, to separate your input

line into *tokens*, which are then passed to your command. Quoting strings preserves additional white space and makes the whole string a single token. In the example above, each token after the command name is a parameter, so we have respectively 1, 2, 4, and 1 parameters.

The echo command has a couple of options. Normally echo will append a trailing new line character to the output. Use the `-n` option to suppress this. Use the `-e` option to enable certain backslash escaped characters to have special meaning. Some of these are shown in Table 1.

Table 1. Echo and escaped characters

Escape sequence	Function
\a	Alert (bell)
\b	Backspace
\c	Suppress trailing newline (same function as -n option)
\f	Form feed (clear the screen on a video display)
\n	New line
\r	Carriage return
\t	Horizontal tab

Escapes and line continuation

There is a small problem with using backslashes in bash. When the backslash character (\) is not quoted, it serves as an escape to signal bash itself to preserve the literal meaning of the following character. This is necessary for special shell metacharacters, which we'll cover in a moment. There is one exception to this rule: a backslash followed by a newline causes bash to swallow both characters and treat the sequence as a line continuation request. This can be handy to break long lines, particularly in shell scripts.

For the sequences described above to be properly handled by the `echo` command or one of the many other commands that use similar escaped control characters, you must include the escape sequences in quotes, or as part of a quoted string, unless you use a second backslash to have the shell preserve one for the command. Listing 4 shows some examples of the various uses of \.

Listing 4. More echo examples

```
[ian@echidna ~]$ echo -n No new line
No new line[ian@echidna ~]$ echo -e "No new line\c"
No new line[ian@echidna ~]$ echo "A line with a typed
> return"
A line with a typed
return
[ian@echidna ~]$ echo -e "A line with an escaped\nreturn"
A line with an escaped
return
[ian@echidna ~]$ echo "A line with an escaped\nreturn but no -e option"
A line with an escaped\nreturn but no -e option
[ian@echidna ~]$ echo -e Doubly escaped\\n\\tmetacharacters
Doubly escaped
        metacharacters
[ian@echidna ~]$ echo Backslash \
> followed by newline \
> serves as line continuation.
Backslash followed by newline serves as line continuation.
```

Note that bash displays a special prompt (>) when you type a line with unmatched quotes. Your input string continues onto a second line and includes the new line character.

Bash shell metacharacters and control operators

Bash has several *metacharacters*, which, when not quoted, also serve to divide input into words. Besides a blank, these are:

- |
- &
- ;
- (
-)
- <
- >

We will discuss some of these in more detail in other parts of this article. For now, note that if you want to include a metacharacter as part of your text, it must be either quoted or escaped using a backslash (\) as shown in Listing 4.

The new line and certain metacharacters or pairs of metacharacters also serve as *control operators*. These are:

- ||
- &&
- &
- ;
- ;;
- |
- (
-)

Some of these control operators allow you to create *sequences* or *lists* of commands.

The simplest command sequence is just two commands separated by a semicolon (;). Each command is executed in sequence. In any programmable environment, commands return an indication of success or failure; Linux commands usually return a zero value for success and a non-zero value in the event of failure. You can introduce some conditional processing into your list using the && and || control operators. If you separate two commands with the control operator && then the second is executed if and only if the first returns an exit value of zero. If you separate the commands with ||, then the second one is executed only if the first one returns a non-zero exit code. Listing 5 shows some command sequences using the echo command. These aren't very exciting since echo returns 0, but you will see more examples later when we have a few more commands to use.

Listing 5. Command sequences

```
[ian@echidna ~]$ echo line 1;echo line 2; echo line 3
line 1
line 2
line 3
[ian@echidna ~]$ echo line 1&&echo line 2&&echo line 3
line 1
line 2
line 3
[ian@echidna ~]$ echo line 1||echo line 2; echo line 3
line 1
line 3
```

Exit

You can terminate a shell using the `exit` command. You may optionally give an exit code as a parameter. If you are running your shell in a terminal window on a graphical desktop, your window will close. Similarly, if you have connected to a remote system using ssh or telnet (for example), your connection will end. In the bash shell, you can also hold the **Ctrl** key and press the **d** key to exit.

Let's look at another control operator. If you enclose a command or a command list in parentheses, then the command or sequence is executed in a sub shell, so the exit command exits the sub shell rather than exiting the shell you are working in. Listing 6 shows a simple example in conjunction with && and || and two different exit codes.

Listing 6. Subshells and sequences

```
[ian@echidna ~]$ (echo In subshell; exit 0) && echo OK || echo Bad exit
In subshell
OK
[ian@echidna ~]$ (echo In subshell; exit 4) && echo OK || echo Bad exit
In subshell
Bad exit
```

Stay tuned for more command sequences later in this article.

Environment variables

When you are running in a bash shell, many things constitute your *environment*, such as the form of your prompt, your home directory, your working directory, the name of your shell, files that

you have opened, functions that you have defined, and so on. Your environment includes many *variables* that may have been set by bash or by you. The bash shell also allows you to have *shell variables*, which you may *export* to your environment for use by other processes running in the shell or by other shells that you may spawn from the current shell.

Both environment variables and shell variables have a *name*. You reference the value of a variable by prefixing its name with '$'. Some of the common bash environment variables that you will encounter are shown in Table 2.

Table 2.Some common bash environment variables

Name	Function
USER	The name of the logged-in user
UID	The numeric user id of the logged-in user
HOME	The user's home directory
PWD	The current working directory
SHELL	The name of the shell
$	The process id (or *PID*of the running bash shell (or other) process
PPID	The process id of the process that started this process (that is, the id of the parent process)
?	The exit code of the last command

Listing 7 shows what you might see in some of these common bash variables.

Listing 7. Environment and shell variables

```
[ian@echidna ~]$ echo $USER $UID
ian 500
[ian@echidna ~]$ echo $SHELL $HOME $PWD
/bin/bash /home/ian /home/ian
[ian@echidna ~]$ (exit 0);echo $?;(exit 4);echo $?
0
4
[ian@echidna ~]$ echo $$ $PPID
2559 2558
```

Not using bash?

The bash shell is the default shell on most Linux distributions. If you are not running under the bash shell, you may want to consider one of the following ways to practice with the bash shell.

- Use the
  `chsh -s /bin/bash`
  command to change your default shell. The default will take effect next time you log in.
- Use the
  `su - $USER -s /bin/bash`
  command to create another process as a child of your current shell. The new process will be a login shell using bash.
- Create an id with a default of a bash shell to use for LPI exam preparation.

You may create or *set* a shell variable by typing a name followed immediately by an equal sign (=). If the variable exists, you will modify it to assign the new value. Variables are case sensitive, so var1 and VAR1 are different variables. By convention, variables, particularly exported variables, are upper case, but this is not a requirement. Technically, $$ and $? are shell *parameters* rather than variables. They may only be referenced; you cannot assign a value to them.

When you create a shell variable, you will often want to *export* it to the environment so it will be available to other processes that you start from this shell. Variables that you export are **not** available to a parent shell. You use the `export` command to export a variable name. As a shortcut in bash, you can assign a value and export a variable in one step.

To illustrate assignment and exporting, let's run the bash command while in the bash shell and then run the Korn shell (ksh) from the new bash shell. We will use the `ps` command to display information about the command that is running. We'll learn more about `ps` in another article in this series. (See Resources for the series roadmap).

Listing 8. More environment and shell variables

```
[ian@echidna ~]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
 2559  2558 -bash
[ian@echidna ~]$ bash
[ian@echidna ~]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
 2811  2559 bash
[ian@echidna ~]$ VAR1=var1
[ian@echidna ~]$ VAR2=var2
[ian@echidna ~]$ export VAR2
[ian@echidna ~]$ export VAR3=var3
[ian@echidna ~]$ echo $VAR1 $VAR2 $VAR3
var1 var2 var3
[ian@echidna ~]$ echo $VAR1 $VAR2 $VAR3 $SHELL
var1 var2 var3 /bin/bash
[ian@echidna ~]$ ksh
$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
 2840  2811 ksh
$ export VAR4=var4
$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
var2 var3 var4 /bin/bash
$ exit
[ian@echidna ~]$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
var1 var2 var3 /bin/bash
[ian@echidna ~]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
 2811  2559 bash
[ian@echidna ~]$ exit
exit
[ian@echidna ~]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
 2559  2558 -bash
[ian@echidna ~]$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
/bin/bash
```

Notes:

1. At the start of this sequence, the bash shell had PID 2559.

2. The second bash shell has PID 2811, and its parent is PID 2559, the original bash shell.
3. We created VAR1, VAR2, and VAR3 in the second bash shell, but only exported VAR2 and VAR3.
4. In the Korn shell, we created VAR4. The `echo` command displayed values only for VAR2, VAR3, and VAR4, confirming that VAR1 was not exported. Were you surprised to see that the value of the SHELL variable had not changed, even though the prompt had changed? You cannot always rely on SHELL to tell you what shell you are running under, but the `ps` command does tell you the actual command. Note that `ps` puts a hyphen (-) in front of the first bash shell to indicate that this is the *login shell*.
5. Back in the second bash shell, we can see VAR1, VAR2, and VAR3.
6. And finally, when we return to the original shell, none of our new variables still exist.

The earlier discussion of quoting mentioned that you could use either single or double quotes. There is an important difference between them. The shell expands shell variables that are between double quotes ($quot;), but expansion is not done when single quotes (') are used. In the previous example, we started another shell within our shell and we got a new process id. Using the `-c` option, you can pass a command to the other shell, which will execute the command and return. If you pass a quoted string as a command, your outer shell will strip the quotes and pass the string. If double quotes are used, variable expansion occurs **before** the string is passed, so the results may not be as you expect. The shell and command will run in another process so they will have another PID. Listing 9 illustrates these concepts. The PID of the top-level bash shell is highlighted.

Listing 9. Quoting and shell variables

```
[ian@echidna ~]$ echo "$SHELL" '$SHELL' "$$" '$$'
/bin/bash $SHELL 2559 $$
[ian@echidna ~]$ bash -c "echo Expand in parent $$ $PPID"
Expand in parent 2559 2558
[ian@echidna ~]$ bash -c 'echo Expand in child $$ $PPID'
Expand in child 2845 2559
```

So far, all our variable references have terminated with white space, so it has been clear where the variable name ends. In fact, variable names may be composed only of letters, numbers or the underscore character. The shell knows that a variable name ends where another character is found. Sometimes you need to use variables in expressions where the meaning is ambiguous. In such cases, you can use curly braces to delineate a variable name as shown in Listing 10.

Listing 10. Using curly braces with variable names

```
[ian@echidna ~]$ echo "-$HOME/abc-"
-/home/ian/abc-
[ian@echidna ~]$ echo "-$HOME_abc-"
--
[ian@echidna ~]$ echo "-${HOME}_abc-"
-/home/ian_abc-
```

Env

The `env` command without any options or parameters displays the current environment variables. You can also use it to execute a command in a custom environment. The `-i` (or just `-`) option

clears the current environment before running the command, while the `-u` option unsets environment variables that you do not wish to pass.

Listing 11 shows partial output of the `env` command without any parameters and then three examples of invoking different shells without the parent environment. Look carefully at these before we discuss them.

Note: If your system does not have the ksh (Korn) or tcsh shells installed, you will need to install them to do these exercises yourself.

Listing 11. The env command

```
[ian@echidna ~]$ env
HOSTNAME=echidna
SELINUX_ROLE_REQUESTED=
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=9.27.206.68 1316 22
SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib/qt-3.3
QTINC=/usr/lib/qt-3.3/include
SSH_TTY=/dev/pts/3
USER=ian
...
_=/bin/env
OLDPWD=/etc
[ian@echidna ~]$ env -i bash -c 'echo $SHELL; env'
/bin/bash
PWD=/home/ian
SHLVL=1
_=/bin/env
[ian@echidna ~]$ env -i ksh -c 'echo $SHELL; env'
/bin/sh
_=/bin/env
PWD=/home/ian
_AST_FEATURES=UNIVERSE - ucb
[ian@echidna ~]$ env -i tcsh -c 'echo $SHELL; env'
SHELL: Undefined variable.
```

Notice that bash has set the SHELL variable, but not exported it to the environment, although there are three other variables that bash has created in the environment. In the ksh example, we have two environment variables, but our attempt to echo the value of the SHELL variable gives a blank line. Finally, tcsh has not created any environment variables and produces an error at our attempt to reference the value of SHELL.

Unset and set

Listing 11 showed different behavior in how shells handle variables and environments. While this article focuses on bash, it is good to know that not all shells behave the same way. Furthermore, shells behave differently according to whether they are a *login shell* or not. For now, we will just say that a login shell is the shell you get when you log in to a system; you can start other shells to behave as login shells if you wish. The three shells started above using `env -i` were not login shells. Try passing the `-l` option to the shell command itself to see what differences you would get with a login shell.

So let's consider our attempt to display the SHELL variable's value in these non-login shells:

1. When bash started, it set the SHELL variable, but it did not automatically export it to the environment.
2. When ksh started, it did not set the SHELL variable. However, referencing an undefined environment variable is equivalent to referencing one with an empty value.
3. When tcsh started, it did not set the SHELL variable. In this case, the default behavior is different than ksh (and bash) in that an error is reported when we attempt to use a variable that does not exist.

You can use the `unset` command to unset a variable and remove it from the shell variable list. If the variable was exported to the environment, this will also remove it from the environment. You can use the `set` command to control many facets of the way bash (or other shells) work. Set is a shell builtin, so the various options are shell specific. In bash, the `-u` option causes bash to report an error with undefined variables rather than treat them as defined but empty. You can turn on the various options to `set` with a `-` and turn them off with a `+`. You can display currently set options using `echo $-`.

Listing 12. Unset and set

```
[ian@echidna ~]$ echo $-
himBH
[ian@echidna ~]$ echo $VAR1

[ian@echidna ~]$ set -u;echo $-
himuBH
[ian@echidna ~]$ echo $VAR1
-bash: VAR1: unbound variable
[ian@echidna ~]$ VAR1=v1
[ian@echidna ~]$ VAR1=v1;echo $VAR1
v1
[ian@echidna ~]$ unset VAR1;echo $VAR1
-bash: VAR1: unbound variable
[ian@echidna ~]$ set +u;echo $VAR1;echo $-

himBH
```

If you use the `set` command without any options, it displays all your shell variables and their values (if any). There is also another command, `declare`, which you can use to create, export, and display values of shell variables. You can explore the many remaining `set` options and the `declare` command using the man pages. We will discuss man pages later in this article.

Exec

One final command to cover is `exec`. You can use the `exec` command to run another program that **replaces** the current shell. Listing 13 starts a child bash shell and then uses `exec` to replace it with a Korn shell. Upon exit from the Korn shell, you are back at the original bash shell (PID 2852, in this example).

Listing 13. Using exec

```
[ian@echidna ~]$ echo $$
2852
[ian@echidna ~]$ bash
[ian@echidna ~]$ echo $$
5114
[ian@echidna ~]$ exec ksh
$ echo $$
5114
$ exit
[ian@echidna ~]$ echo $$
2852
```

System information with uname

The `uname` command prints information about your system and its kernel. Listing 14 shows the various options for `uname` and the resulting information; each option is defined in Table 3.

Listing 14. The uname command

```
[ian@echidna ~]$ uname
Linux
[ian@echidna ~]$ uname -s
Linux
[ian@echidna ~]$ uname -n
echidna.raleigh.ibm.com
[ian@echidna ~]$ uname -r
2.6.29.6-217.2.3.fc11.i686.PAE
[ian@echidna ~]$ uname -v
#1 SMP Wed Jul 29 16:05:22 EDT 2009
[ian@echidna ~]$ uname -m
i686
[ian@echidna ~]$ uname -o
GNU/Linux
[ian@echidna ~]$ uname -a
Linux echidna.raleigh.ibm.com 2.6.29.6-217.2.3.fc11.i686.PAE
#1 SMP Wed Jul 29 16:05:22 EDT 2009 i686 i686 i386 GNU/Linux
```

Table 3. Options for uname

Option	Description
-s	Print the kernel name. This is the default if no option is specified.
-n	Print the nodename or hostname.
-r	Print the release of the kernel. This option is often used with module-handling commands.
-v	Print the version of the kernel.
-m	Print the machine's hardware (CPU) name.
-o	Print the operating system name.
-a	Print all of the above information.

Listing 14 is from an Fedora 11 system running on an Intel® CPU. The `uname` command is available on most UNIX® and UNIX-like systems as well as Linux. The information printed will vary

by Linux distribution and version as well as by the type of machine you are running on. Listing 15 shows the output from an AMD Athlon 64 system running Ubuntu 9.04.

Listing 15. Using uname with another system

```
ian@attic4:~$ uname -a
Linux attic4 2.6.28-14-generic #47-Ubuntu SMP Sat Jul 25 01:19:55 UTC 2009 x86_64
GNU/Linux
```

Command history

If you are typing in commands as you read, you may notice that you often use a command many times, either exactly the same, or with slight changes. The good news is that the bash shell can maintain a *history* of your commands. By default, history is on. You can turn it off using the command `set +o history` and turn it back on using `set -o history`. An environment variable called HISTSIZE tells bash how many history lines to keep. A number of other settings control how history works and is managed. See the bash man pages for full details.

Some of the commands that you can use with the history facility are:

history
: Displays the entire history

history *N*
: Displays the last *N* lines of your history

history -d *N*
: Deletes line *N* form your history; you might do this if the line contains a password, for example

!!
: Your most recent command

!*N*
: The *N*th history command

!-*N*
: The command that is *N* commands back in the history (!-1 is equivalent to !!)

!#
: The current command you are typing

!*string*
: The most recent command that starts with *string*

!?*string***?**
: The most recent command that contains*string*

You can also use a colon (:) followed by certain values to access or modify part or a command from your history. Listing 16 illustrates some of the history capabilities.

Listing 16. Manipulating history

```
[ian@echidna ~]$ echo $$
2852
[ian@echidna ~]$ env -i bash -c 'echo $$'
9649
[ian@echidna ~]$ !!
env -i bash -c 'echo $$'
10073
```

```
[ian@echidna ~]$ !ec
echo $$
2852
[ian@echidna ~]$ !en:s/$$/$PPID/
env -i bash -c 'echo $PPID'
2852
[ian@echidna ~]$ history 6
  595  echo $$
  596  env -i bash -c 'echo $$'
  597  env -i bash -c 'echo $$'
  598  echo $$
  599  env -i bash -c 'echo $PPID'
  600  history 6
[ian@echidna ~]$ history -d598
```

The commands in Listing 16 do the following:

1. Echo the current shell's PID
2. Run an echo command in a new shell and echo that shell's PID
3. Rerun the last command
4. Rerun the last command starting with 'ec'; this reruns the first command in this example
5. Rerun the last command starting with 'en', but substitute '$PPID' for '$$', so the parent PID is displayed instead
6. Display the last 6 commands of the history
7. Delete history entry 598, the last echo command

You can also edit the history interactively. The bash shell uses the readline library to manage command editing and history. By default, the keys and key combinations used to move through the history or edit lines are similar to those used in the GNU Emacs editor. Emacs keystroke combinations are usually expressed as **C-x** or **M-x**, where **x** is a regular key and **C** and **M** are the *Control* and *Meta* keys, respectively. On a typical PC system, the **Ctrl** key serves as the Emacs Control key, and the **Alt** key serves as the Meta key. Table 3 summarizes some of the history editing functions available. Besides the key combinations shown in Table 3, cursor movement keys such as the right, left, up, and down arrows, and the Home and End keys are usually set to work in a logical way. Additional functions as well as how to customize these options using a readline init file (usually inputrc in your home directory) can be found in the man pages.

Table 3. History editing with emacs commands

Command	Common PC key	Description
C-f	Right arrow	Move one space to the right
C-b	Left arrow	Move one space to the left
C-p	Up arrow	Move one command earlier in history
C-n	Down arrow	Move one command later in history
C-r		Incremental reverse search. Type a letter or letters to search backwards for a string. Press C-r again to search for the next previous occurrence of the same string.
M-f	Alt-f	Move to beginning of next word; GUI environments usually take this key combination to open the **File** menu of the window

M-b	Alt-b	Move to beginning of previous word
C-a	Home	Move to beginning of line
C-e	End	Move to end of line
Backspace	Backspace	Delete the character preceding the cursor
C-d	Del	Delete the character under the cursor (Del and Backspace functions may be configured with opposite meanings)
C-k	Ctrl-k	Delete (kill) to end of line and save removed text for later use
M-d	Alt-d	Delete (kill) to end of word and save removed text for later use
C-y	Ctrl-y	Yank back text removed with a kill command

If you prefer to manipulate the history using a vi-like editing mode, use the command `set -o vi` to switch to vi mode. Switch back to emacs mode using `set -o emacs`. When you retrieve a command in vi mode, you are initially in vi's insert mode. The vi editor is covered in another article in this series. (See Resources for the series roadmap).

Paths - where's my command?

Some bash commands are builtin, while others are external. Let's now look at external commands and how to run them, and how to tell if a command is internal.

Where does the shell find commands?

External commands are just files in your file system. Basic file management is covered in another article in this series. (See Resources for the series roadmap). On Linux and UNIX systems, all files are accessed as part of a single large tree that is rooted at /. In our examples so far, our current directory has been the user's home directory. Non-root users usually have a home directory within the /home directory, such as /home/ian, in my case. Root's home is usually /root. If you type a command name, then bash looks for that command on your *path*, which is a colon-separated list of directories in the PATH environment variable.

If you want to know what command will be executed if you type a particular string, use the `which` or `type` command. Listing 17 shows my default path along with the locations of several commands.

Listing 17. Finding command locations

```
[ian@echidna ~]$ echo $PATH
/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/usr/lib/ccache:/usr/local/bin:/bin:/usr/b
in:/home/ian/bin
[ian@echidna ~]$ which bash env zip xclock echo set ls
alias ls='ls --color=auto'
        /bin/ls
/bin/bash
/bin/env
/usr/bin/zip
/usr/bin/xclock
/bin/echo
/usr/bin/which: no set in (/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/usr/lib/ccache
:/usr/local/bin:/bin:/usr/bin:/home/ian/bin)
[ian@echidna ~]$ type bash env zip xclock echo set ls
bash is hashed (/bin/bash)
```

```
env is hashed (/bin/env)
zip is /usr/bin/zip
xclock is /usr/bin/xclock
echo is a shell builtin
set is a shell builtin
ls is aliased to `ls --color=auto'
```

Note that the directories in the path mostly end in /bin. This is a common convention, but not a requirement as you can see from /usr/lib/ccache. The `which` command reported that the `ls` command is an *alias* and that the `set` command could not be found. In this case, we interpret that to mean that it does not exist or that it is a builtin. The `type` command reports that the `ls` command is an *alias*, but it identifies the `set` command as a shell builtin. It also reports that there is a builtin `echo` command as well as the one in /bin that was found by `which`. The commands also produce output in different orders.

We saw that the `ls` command, used for listing directory contents, is an *alias*. Aliases are a handy way to configure some commands to use different sets of defaults or to provide an alternate name for a command. In our example, the `--color=tty` option causes directory listings to be color coded according to the type of file or directory. Try running `dircolors --print-database` to see how the color codings are controlled and which colors are used for what kind of file.

Each of these commands has additional options. Depending on your need, you may use either command. I tend to use `which` when I am reasonably sure I'll find an executable and I just need its full path specification. I find that `type` gives me more precise information, which I sometimes need in a shell script.

Running other commands

We saw in Listing 17 that executable files have a full path starting with /, the root directory. For example, the xclock program is really /usr/bin/xclock, a file located in the /usr/bin directory. On older systems you might find this in the /usr/X11R6/bin directory instead. If a command is **not** in your PATH specification, you may still run it by specifying a path as well as a command name. There are two types of paths that you can use:

- *Absolute* paths are those starting with /, such as those we saw in Listing 17 (/bin/bash, /bin/env, etc).
- *Relative* paths are relative to your *current working directory*, as reported by the `pwd` command. These commands do not start with /, but do contain at least one /.

You may use absolute paths regardless of your current working directory, but you will probably use relative paths only when a command is close to your current directory. Suppose you are developing a new version of the classic "Hello World!" program in a subdirectory of your home directory called mytestbin. You might use the relative path to run your command as `mytestbin/hello`. There are two special names you can use in a path; a single dot (.) refers to the current directory, and a pair of dots (..) refers to the parent of the current directory. Because your home directory is usually not on your PATH (and generally should not be), you will need to explicitly provide a path for any executable that you want to run from your home directory. For example, if you had a copy of your hello program in your home directory, you could run it using the command `./hello`. You can use both . and .. as part of an absolute path, although a single . is not very useful

in such a case. You can also use a tilde (~) to mean your own home directory and ~*username* to mean the home directory of the user named *username*. Some examples are shown in Listing 18.

Listing 18. Absolute and relative paths

```
[ian@echidna ~]$ /bin/echo Use echo command rather than builtin
Use echo command rather than builtin
[ian@echidna ~]$ /usr/../bin/echo Include parent dir in path
Include parent dir in path
[ian@echidna ~]$ /bin/./././echo Add a couple of useless path components
Add a couple of useless path components
[ian@echidna ~]$ pwd # See where we are
/home/ian
[ian@echidna ~]$ ../../bin/echo Use a relative path to echo
Use a relative path to echo
[ian@echidna ~]$ myprogs/hello # Use a relative path with no dots
-bash: myprogs/hello: No such file or directory
[ian@echidna ~]$ mytestbin/hello # Use a relative path with no dots
Hello world!
[ian@echidna ~]$ ./hello
Hello world!
[ian@echidna ~]$ ~/mytestbin/hello # run hello using ~
Hello world!
[ian@echidna ~]$ ../hello # Try running hello from parent
-bash: ../hello: No such file or directory
```

Changing your working directory

Just as you can execute programs from various directories in the system, so too can you change your current working directory using the `cd` command. The argument to `cd` must be the absolute or relative path to a directory. As for commands, you can use ., .., ~, and ~*username* in paths. If you use `cd` with no parameters, the change will be to your home directory. A single hyphen (-) as a parameter means to change to the previous working directory. Your home directory is stored in the HOME environment variable, and the previous directory is stored in the OLDPWD variable, so `cd` alone is equivalent to `cd $HOME` and `cd -` is equivalent to `cd $OLDPWD`. Usually we say *change directory* instead of the full *change current working directory*.

As for commands, there is also an environment variable, CDPATH, which contains a colon-separated set of directories that should be searched (in addition to the current working directory) when resolving relative paths. If resolution used a path from CDPATH, then `cd` will print the full path of the resulting directory as output. Normally, a successful directory change results in no output other than a new, and possibly changed, prompt. Some examples are shown in Listing 19.

Listing 19. Changing directories

```
[ian@echidna ~]$ cd /;pwd
/
[ian@echidna /]$ cd /usr/local;pwd
/usr/local
[ian@echidna local]$ cd ;pwd
/home/ian
[ian@echidna ~]$ cd -;pwd
/usr/local
/usr/local
[ian@echidna local]$ cd ~ian/..;pwd
/home
[ian@echidna home]$ cd ~;pwd
/home/ian
[ian@echidna ~]$ export CDPATH=~
[ian@echidna ~]$ cd /;pwd
/
[ian@echidna /]$ cd mytestbin
/home/ian/mytestbin
```

Manual (man) pages

Our final topic in this article tells you how to get documentation for Linux commands through manual pages and other sources of documentation.

Manual pages and sections

The primary (and traditional) source of documentation is the *manual pages*, which you can access using the `man` command. Figure 1 illustrates the manual page for the `man` command itself. Use the command `man man` to display this information.

Figure 1. Man page for the man command

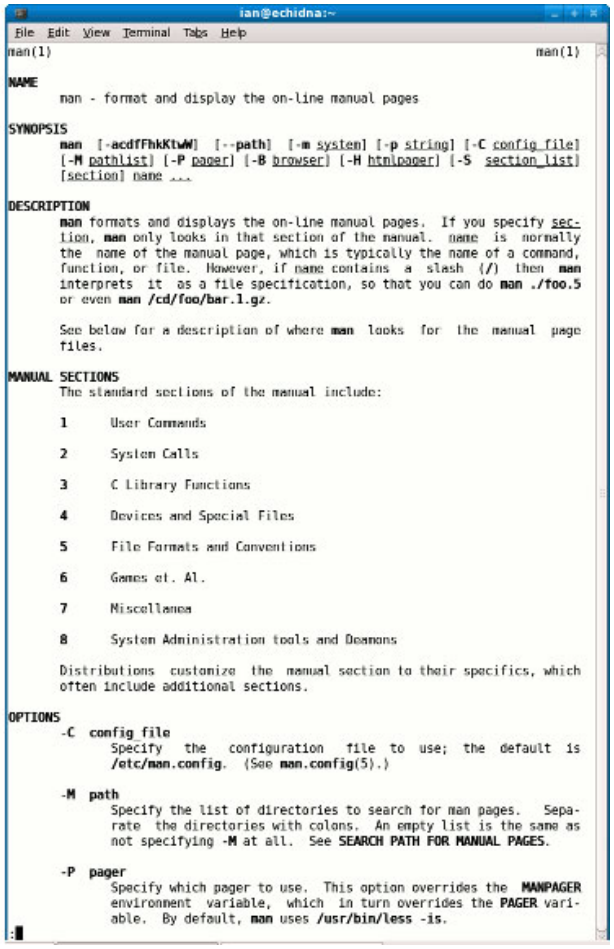

Figure 1 shows some typical items in man pages:

- A heading with the name of the command followed by its section number in parentheses
- The name of the command and any related commands that are described on the same man page
- A synopsis of the options and parameters applicable to the command
- A short description of the command
- Detailed information on each of the options

You might find other sections on usage, how to report bugs, author information, and a list of related commands. For example, the man page for `man` tells us that related commands (and their manual sections) are:
apropos(1), whatis(1), less(1), groff(1), and man.conf(5).

There are eight common manual page sections. Manual pages are usually installed when you install a package, so if you do not have a package installed, you probably won't have a manual page for it. Similarly, some of your manual sections may be empty or nearly empty. The common manual sections, with some example contents are:

1. User commands (env, ls, echo, mkdir, tty)

2. System calls or kernel functions (link, sethostname, mkdir)
3. Library routines (acosh, asctime, btree, locale, XML::Parser)
4. Device related information (isdn_audio, mouse, tty, zero)
5. File format descriptions (keymaps, motd, wvdial.conf)
6. Games (note that many games are now graphical and have graphical help outside the man page system)
7. Miscellaneous (arp, boot, regex, unix utf8)
8. System administration (debugfs, fdisk, fsck, mount, renice, rpm)

Other sections that you might find include *9* for Linux kernel documentation, *n* for new documentation, *o* for old documentation, and *l* for local documentation.

Some entries appear in multiple sections. Our examples show mkdir in sections 1 and 2, and tty in sections 1 and 4. You can specify a particular section, for example, `man 4 tty` or `man 2 mkdir`, or you can specify the `-a` option to list all applicable manual sections.

You may have noticed in the figure that `man` has many options for you to explore on your own. For now, let's take a quick look at some of the "See also" commands related to `man`.

See also

Two important commands related to `man` are `whatis` and `apropos`. The `whatis` command searches man pages for the name you give and displays the name information from the appropriate manual pages. The `apropos` command does a keyword search of manual pages and lists ones containing your keyword. Listing 20 illustrates these commands.

Listing 20. Whatis and apropos examples

```
[ian@echidna ~]$ whatis man
man []               (1)  - format and display the on-line manual pages
man []               (1p)  - display system documentation
man []               (7)  - macros to format man pages
man []               (7)  - pages - conventions for writing Linux man pages
man.config []        (5)  - configuration data for man
man-pages           (rpm) - Man (manual) pages from the Linux Documentation Project
man                 (rpm) - A set of documentation tools: man, apropos and whatis
[ian@echidna ~]$ whatis mkdir
mkdir []             (1)  - make directories
mkdir []             (1p)  - make directories
mkdir []             (2)  - create a directory
mkdir []             (3p)  - make a directory
[ian@echidna ~]$ apropos mkdir
mkdir []             (1)  - make directories
mkdir []             (1p)  - make directories
mkdir []             (2)  - create a directory
mkdir []             (3p)  - make a directory
mkdirat []           (2)  - create a directory relative to a directory file descriptor
mkdirhier []         (1)  - makes a directory hierarchy
```

By the way, if you cannot find the manual page for man.conf, try running `man man.conf**ig**` instead.

The `man` command pages output onto your display using a paging program. On most Linux systems, this is likely to be the `less` program. Another choice might be the older `more` program. If

you wish to print the page, specify the `-t` option to format the page for printing using the `groff` or `troff` program.

The less pager has several commands that help you search for strings within the displayed output. Use `man less` to find out more about */* (search forwards), *?* (search backwards), and *n* (repeat last search), among many other commands.

Other documentation sources

In addition to manual pages accessible from a command line, the Free Software Foundation has created a number of *info* files that are processed with the *info* program. These provide extensive navigation facilities including the ability to jump to other sections. Try `man info` or `info info` for more information. Not all commands are documented with info, so you will find yourself using both man and info if you become an info user.

There are also some graphical interfaces to man pages, such as `xman` (from the XFree86 Project) and `yelp` (the Gnome 2.0 help browser).

If you can't find help for a command, try running the command with the `--help` option. This may provide the command's help, or it may tell you how to get the help you need.

Resources

Learn

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- Use the roadmap for LPIC-1 to find the developerWorks articles you need to help you prepare for LPIC-1 certification based on the April 2009 objectives.
- See the April 2009 objectives for LPI exam 101 and LPI exam 102. You should always refer to the objectives for the definitive requirements.
- Review the entire LPI exam prep series on developerWorks to learn Linux fundamentals and prepare for system administrator certification based on earlier objectives prior to April 2009.
- At the LPIC Program, find task lists, sample questions, and detailed objectives for the three levels of the Linux Professional Institute's Linux system administration certification.
- In "Basic tasks for new Linux developers" (developerWorks, March 2005), learn how to open a terminal window or shell prompt and much more.
- The Linux Documentation Project has a variety of useful documents, especially its HOWTOs.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the My developerWorks community; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

Ian Shields

Ian Shields works on a multitude of Linux projects for the developerWorks Linux zone. He is a Senior Programmer at IBM at the Research Triangle Park, NC. He joined IBM in Canberra, Australia, as a Systems Engineer in 1973, and has since worked on communications systems and pervasive computing in Montreal, Canada, and RTP, NC. He has several patents and has published several papers. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University. Learn more about Ian in Ian's profile on developerWorks Community.