

Speaking UNIX: Just a few clicks

Edit the command line like an expert

Skill Level: Intermediate

[Adam Cormany \(acormany@yahoo.com\)](mailto:acormany@yahoo.com)

National Data Center Manager
Scientific Games Corporation

01 Jul 2008

The way you interface with a computer is changing constantly. Operating systems that once started as a command line-only interface have moved to a graphical front end. But moving away from what made the operating system great isn't always a step in the right direction. The IBM® AIX® operating system has kept to what's important: stability, functionality, robustness. And it has done it by keeping a strong command-line interface (CLI). If you never learned to use the CLI or need a refresher on its basics, read on.

The way you interface with a computer is changing constantly. Operating systems that once started as a command line-only interface have moved to a graphical front end. Sometimes, however, moving away from the building blocks that made the operating system isn't necessarily a step in the right direction. More often than not, moving toward a graphical user interface (GUI) means losing functionality; in addition, users become less inclined to learn more of the computer they're working with. Thankfully the AIX operating system—like other UNIX® and Linux® systems—has kept to what's important: the stability, functionality, and robustness of a computer's operating system.

The various UNIX and Linux vendors have kept a strong grasp of the importance behind the CLI of an operating system. But for reasons of automation, making computing easier for users, or something else, users have either forgotten or never learned the ins and outs of the CLI. This article sheds some light on the CLI for those users who haven't touched it much or for those who may need a little nudge to remember why it's so important to administration, development, and general UNIX computing.

What is the command line?

When working on computers, it's important to understand what you're actually working on. If you've ever worked on UNIX or Linux, it's a fair bet that you've heard the term *shell* or the *command line*. The terms can be used synonymously and refer to the actual UNIX shell the user is running. The term *shell* in UNIX refers to the interface you use when typing commands or performing functions.

When a user logs in to a UNIX system through the console or over a network, a definable shell (in `/etc/passwd`) is evoked, the user's environment is set up through configuration files (explained later in this article), and the user is ready to perform actions in the shell. When the user is typing a command on the command line—that is, the shell he or she is using—the user only sees *stdin*, or standard in—that is, input that the user or a program provides. When the user clicks **Enter** or **Return**, the *stdin* is sent through the shell to execute, and the user may receive *stdout*, or standard out, as well as *stderr*, or standard error, depending on how the output is redirected (for example, to the user's display, a file, a printer). The term *stdout* is the output data that the program executed returns, and *stderr* refers to errors that the program encountered or returned. The user doesn't see all the low-level code executions to handle single or multiple commands but rather a very simplistic input, output, and error. Because of this, the program the user evoked when logging in has been rightfully called a *shell*, because it hides all the operating system's low-level calls.

The history of the shell

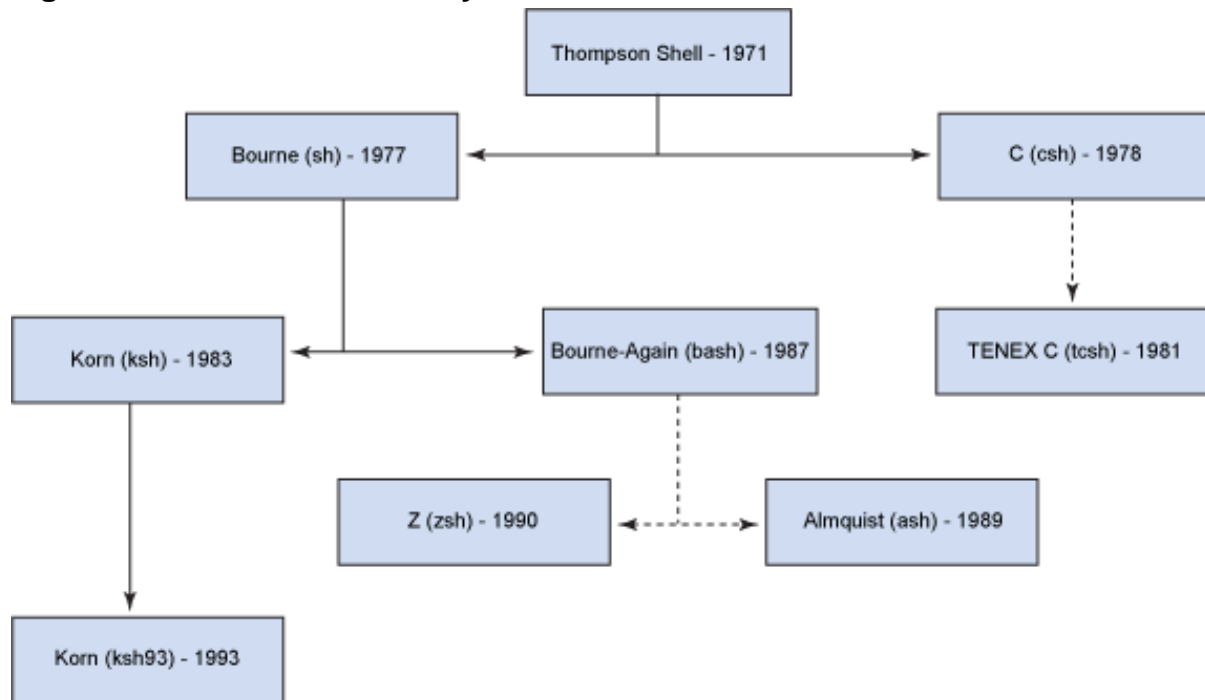
The UNIX shell has been around for more than 35 years now—through evolution and enhancements—and is still going strong! It all began in 1971, when Ken Thompson of AT&T Bell Laboratories created the first UNIX shell named (appropriately) the *Thompson shell*. Fundamentals of the Thompson shell, such as redirection of data, exists in shells used today, although the shell lacked some important built-in functions that UNIX users use every day, such as pipes (`|`), the ability to write shell scripts, and `if` conditional statements.

As a result, the Thompson shell was replaced with the Bourne shell, or *sh*, in 1977. The Bourne shell, created by Stephen Bourne of AT&T Bell Laboratories, became the default shell for UNIX version 7 (V7). The shell took a huge leap into the future for UNIX. Now, users could write shell scripts; store and export information in variables; control file descriptors; control signal handling, `for` loops, and `case` statements; and so much more. Even though the Bourne shell was created more than 30 years ago, it is still widely used by many current UNIX systems and is the default shell for the superuser—root—on many UNIX systems today.

Over the past three decades, there have been changes and improvements to the

UNIX shell. As a result, several different shells have been created. [Figure 1](#) illustrates the family tree of a few of the UNIX shells. This figure is by no means complete, but it shows the major shells from which other, minor shells have been derived.

Figure 1. The UNIX shell family tree



The Korn shell

The Korn shell, or *ksh*, was originally developed by David Korn of AT&T Bell Laboratories in 1982. The shell, like many other shells, is backwards compatible with the Bourne shell (*sh*) and has evolved into a robust, stable, and very reliable shell in its more than 25 years of existence. IBM uses the Korn shell as its default shell in AIX. Two versions of Korn shell are available, and AIX contains both.

The first—and default shell for normal users in AIX—is the standard *ksh* shell. The Korn shell conforms to the Portable Operating System Interface for Computer Environments (POSIX), which is an international standard for operating systems.

The second Korn shell available in AIX is the enhanced Korn shell, called *ksh93*. In addition to all the great features of the standard Korn shell, the enhanced Korn shell contains such features as:

- Arithmetic enhancements
- Compound variables

- Compound assignments
- Associative arrays
- Variable name references
- Parameter expansions
- Discipline functions
- Function environments
- PATH search rules
- Shell history
- Additional built-in commands

For a complete list of enhancements and differences between ksh and ksh93, see [Resources](#).

Setting up the command-line environment with ksh

Before looking at editing the command line with ksh, you must set up your environment. Setting up the Korn shell to your liking is relatively simple: While logged in under ksh, view your current settings by using the `-o` switch with the `set` command:

```
# set -o

Current option settings are:
allexport      off
bgnice        on
emacs         off
errexit       off
gmacs         off
ignoreeof     on
interactive   on
keyword       off
markdirs      off
monitor       on
noexec        off
noclobber     off
noglob        off
nolog         off
notify        off
nounset       off
privileged    off
restricted    off
trackall      off
verbose       off
vi            off
viraw         on
xtrace        off
```

Here's a brief explanation of each setting. (You can also find this explanation by running `man set`.)

- **allexport** : Export all defined subsequent variables automatically.
- **bgnice** : Run all processes in the background at a lower priority.
- **emacs** : When editing the command-line text entered, use the emacs-style inline editor.
- **errexit** : If a command has an exit status of anything but 0 (zero), execute the ERR trap (if it is set and exists).
- **gmacs** : When editing the command-line text entered, use the gmacs-style inline editor.
- **ignoreeof** : Ignore end-of-file characters, and do not exit the shell. If the user wants to exit, the user must type the `exit` command or press Control-D 11 times.
- **keyword** : Rather than placing only the arguments that precede a command, this option places all arguments in the environment for a command, which can be viewed with the `set` command.
- **markdirs** : Place a forward slash (/) on the end of all directories that are from a file name substitution.
- **monitor** : Run all processes in the background, as a separate process, and inform the user when the process has finished by printing a line to stdout.
- **noexec** : Do not execute the commands. Instead, just check for syntax errors.
Note: This parameter isn't used if attempted in interactive shells.
- **noclobber** : This flag prohibits existing files from being truncated when output is redirected to it. If this option is used, truncating can still occur if a greater-than symbol and a pipe (> |) are used, instead.
- **noglob** : File name substitution is disabled.
- **nolog** : Function definitions will not be stored in the history file if this option is used.
- **nounset** : If substituting, all unset parameters will be returned as an error.
- **restricted** : Run a restricted shell. Users cannot change directories; change their SHELL, ENV, or PATH variables; execute a command that contains a forward slash (/) in the pathname; or redirect output.
- **trackall** : Each command, when initially run, will be a tracked alias.
- **verbose** : Display all input lines to stdout as the shell reads them.

- **vi** : When editing the command-line text entered, use the vi-style inline editor.
- **viraw** : As each character is typed, execute it as if it were typed in the vi editor.
- **xtrace** : Display all commands and arguments as they are being executed to stdout.

To turn options on with the built-in command set, use the `-o` switch. If you change your mind, you can turn off the options you set by using the `+o` switch, instead.

The main option I focus on in this article is the inline editor switch. Depending on the individual, some favor one file editor over another, be it vi, emacs, or gmacs. The Korn shell accommodates all three. However, I focus on the vi inline editor. Setting the inline editor option to `vi` is easy. Simply enter the option into the command you used to view all the current settings:

```
# set -o vi
```

That's it! To verify the setting, you can look at your current settings again:

```
# set -o

Current option settings are:
allexport      off
bgnice        on
emacs         off
errexit       off
gmacs         off
ignoreeof     on
interactive   on
keyword       off
markdirs      off
monitor       on
noexec        off
noclobber     off
noglob        off
nolog         off
notify        off
nounset       off
privileged    off
restricted    off
trackall      off
verbose       off
vi           on
viraw         on
xtrace        off
```

Using the Korn shell vi inline editor

Now that your shell has been configured to use the vi inline editor, it's time to test it

out.

Modifying text on the command line

When you type on the command line now, think of it as you're now in insert mode in the vi editor. If you make a mistake or need to add something to the command to execute, simply click the **Esc** key to exit insert mode and switch back to command mode.

For example, the present working directory you're in has the contents:

```
# ls
fileA   fileAA  fileAAA  fileAB   fileABA  fileABB
fileB   fileBAA fileBB   fileBBB
```

You want to find files that begin with *fileAA* and remove them:

```
# find . -name "fileAB*" -exec rm {} \;
```

Before executing the line you typed, you notice that you made a mistake and accidentally typed *fileAB* instead of *fileAA*! No need to worry. Simply exit insert mode to switch into command mode, move the cursor to the incorrect letter, and replace it—all using vi commands. To break down the command sequence, while still in the insert mode of the inline editor:

1. Click **Esc** to switch to command mode.
2. Move the cursor left to highlight the **B** in the string "fileAB*" using vi-style movement commands. (The H key moves left.)
Note: If you're accustomed to using the arrow keys in vi, it's wise to learn the actual letters on the keyboard to move the cursor, as the TERM type may differ and you may not achieve the desired results with the arrow keys:
 - **h**: Left
 - **l**: Right
 - **k**: Up
 - **j**: Down
3. Replace the *B* with *A* using the vi-style "replace single character" commands (that is, click **R**, and then type **A**).

When you've reviewed your work and agree that this is what you want, click **Enter** to execute the command:

```
# find . -name "fileAA*" -exec rm {} \;
# ls
fileA    fileAB   fileABA  fileABB  fileB    fileBAA
fileBB   fileBBB
```

File name completion

Another useful operation of the vi inline editor in the Korn shell is file name completion. When executing commands, there are often times when a file you're using as an argument for stdin or stdout (or stderr) is being written to a file. File names can become long, there may be several files with similar names, or you simply can't remember the full file name. This is where file name completion comes to the rescue. If, when typing the file name, you get halfway through, simply click the **Esc** key, then the backslash (\) key. It's convenient and saves a lot of time!

For example, I want to view the /etc/filesystems file on AIX, but I forgot the full file name. I know it's in /etc, and I know the file begins with *file*, but that's it. I simply type **view /etc/file** and click **Esc-**, and voilà! ksh has completed the line for me. The command line now reads `view /etc/filesystems`.

The same can be done on a directory structure, because they are really just file names, too.

Viewing and modifying command history

How many times do you type the same command over and over while monitoring a process or performing some other function on your UNIX system? Rather than constant retyping, the Korn shell has a built-in command history for your review. If you also have your inline editor set to vi, ksh allows you to pull the history of commands executed by that user—sometimes only for that session, depending on how you've configured your system—and modify the commands as you would any other text typed on the command line.

If you've defined a file name in the variable HISTFILE, ksh allows users to pull from their history and modify the commands or simply re-execute the original command. For example, here are the last 10 occurrences of a sample \$HISTFILE:

```
# tail -10 $HISTFILE
ls
cd ~cormany/testdir/dirA
./fileA 1>fileA.out 2>fileA.errors
pwd
ps -fu cormany
df -k .
ps -fu cormany
```



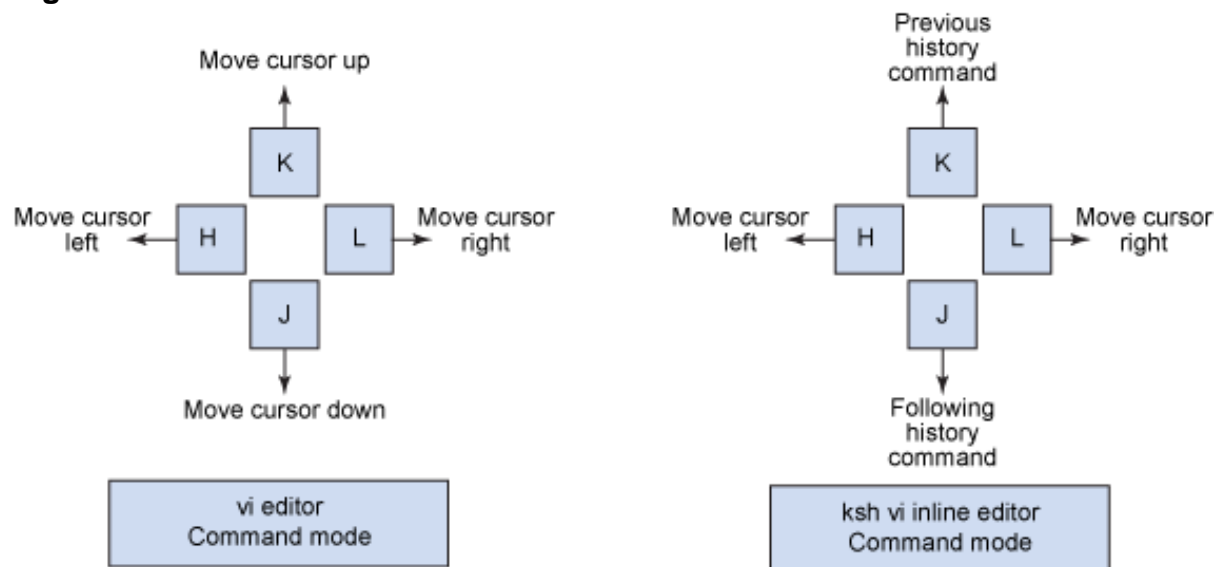
```
find . -name "fileA.out" -ls
find . -name "fileA.errors" -ls
tail -10 $HISTFILE
```

While on the command line, simply click **Esc** to enter command mode in the vi inline editor, and then click **K** to pull the last command executed. Because you're still in command mode, you can continue clicking **K** to move up the history of commands executed or **J** to move down the list.

To help simplify the command-mode cursor movement, when you click **Esc** at a command prompt, think of your `$HISTFILE` being loaded as a normal file in vi. In the vi editor, the **K** key moves up one line, while the **J** key moves down a line. If clicking **Esc-J** and using the sample `$HISTFILE`, visualize editing the `$HISTFILE` and the cursor beginning at the bottom of the file. The line would be `tail -10 $HISTFILE`. If you clicked **J** again, you would move up a single line in the `$HISTFILE` you're editing, which would be `find . -name "fileA.errors" -ls`.

Figure 2 provides a small "cheat sheet" comparing regular vi command-mode cursor movement against the ksh vi inline editor command-mode movement.

Figure 2. A vi command-mode cheat sheet



Command line versus shell script

There are times for shell scripts and there are times for command line use. If a task is to be performed on a routine basis or the task requested is complex, requiring data manipulation, rather than asking users always to type the commands, a shell script becomes useful. Other times, when it's a single occurrence and something relatively simple, the command line can do the trick nicely.

For example, take this directory listing:

```
# ls
fileA.tar.gz  fileAA.tar.gz  fileB.tar.gz  fileBB.tar.gz
```

If you simply want to uncompress the files, recompress them with bzip2, and transfer them to ATC-AIX2, rather than typing a shell script, you could do it on the command line. Think of a shell script as several command-line entries typed at once, because that is what it really is, in a sense. When typing commands on a command line, it's just like typing them into a script, and then executing the script.

You want to loop through the files in the directory that end with `gz`, uncompress them, recompress with bzip2, and then use the `scp` command on the files to the destination server of ATC-AIX. A loop works on the command line as nicely as it does in a script. When beginning a `loop...if` conditional statement, `case` switch statement, or other code block statements, the ksh you're running will simply move the cursor to the next line, but the prompt will change to `$PS2`. When the code block has been completed, the code block will be executed and return the user to a `$PS1` prompt.

In other words:

- **\$PS1 prompt:** Waiting for the next command
- **\$PS1 prompt:** Code block starts
- **\$PS2 prompt:** Code block continues
- **\$PS2 prompt:** Code block continues
- **\$PS2 prompt:** Code block ends
- Code block executes
- **\$PS1 prompt:** Waiting for the next command

The default value to variable `PS2` is `>`. Going back to the previous function of uncompress and then recompress, you would simply type the following at a ksh command line:

```
# for _FNAME in `ls -l *.gz`
> do
> gzip -d ${_FNAME}
> bzip2 ${_FNAME%*.gz}
> scp ${_FNAME%*.gz}.bz2 cormany@ATC-AIX2:/home/cormany
> done
```

When you click **Enter** after completing the code block (that is, for a loop terminating

with `done`), the loop will begin. The loop typed on the command line searches for all files in the current working directory ending with `.gz`, uncompress them, recompresses them with `bzip2`, and transfers them to the directory `/home/cormany` on ATC-AIX2. It's as simple as that.

Conclusion

After reading this article, you should now be able to use the Korn shell in ways you may not have known before. Mastering the command line can simplify your work and help you better understand how to make the shell and command line work for you rather than you working harder for it.

Resources

Learn

- [Speaking UNIX](#): Check out other parts in this series.
- [Wikipedia's AIX entry](#): Read Wikipedia's excellent entry on the AIX operating system for more information about its background and development.
- [Wikipedia's UNIX shells entry](#): Read Wikipedia for more information about UNIX shells.
- [Wikipedia's Korn shell entry](#): Read Wikipedia's excellent entry on the Korn shell.
- [C/C++ and shell standard streams](#): Read Wikipedia's entry on standard streams.
- [The Korn shell](#): Learn more about the Korn shell on IBM's Commands Reference (man) page.
- [The enhanced Korn shell](#): Learn more about the enhanced Korn shell (ksh93) on IBM's Commands Reference (man) page.
- [kdh93](#): Learn more about the enhanced Korn shell on the Combined IBM System Information Center.
- [The AIX and UNIX developerWorks zone](#) provides a wealth of information relating to all aspects of IBM® AIX® systems administration and expanding your UNIX skills.
- [New to AIX and UNIX?](#) Visit the New to AIX and UNIX page to learn more.
- [developerWorks technical events and webcasts](#): Stay current with developerWorks technical events and webcasts.
- [AIX](#): Visit this collaborative environment for technical information related to AIX.
- [Podcasts](#): Tune in and catch up with IBM technical experts.

Get products and technologies

- [IBM trial software](#): Build your next development project with software for download directly from developerWorks.

Discuss

- Participate in the AIX and UNIX forums:
 - [AIX Forum](#)
 - [AIX Forum for developers](#)
 - [Cluster Systems Management](#)

- [IBM Support Assistant Forum](#)
- [Performance Tools Forum](#)
- [Virtualization Forum](#)
- [More AIX and UNIX forums](#)

About the author

Adam Cormany

Adam Cormany is an UNIX systems engineer and has worked with AIX, Solaris, and Red Hat Linux administration for more than 10 years. He is an IBM eServer® Certified Specialist in pSeries® AIX System Administration. In addition to administration, Adam has extensive knowledge of shell scripting in BASH, CSH, and KSH, as well as programming in C, PHP, and Perl.

Trademarks

IBM and AIX are registered trademarks of International Business Machines in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.