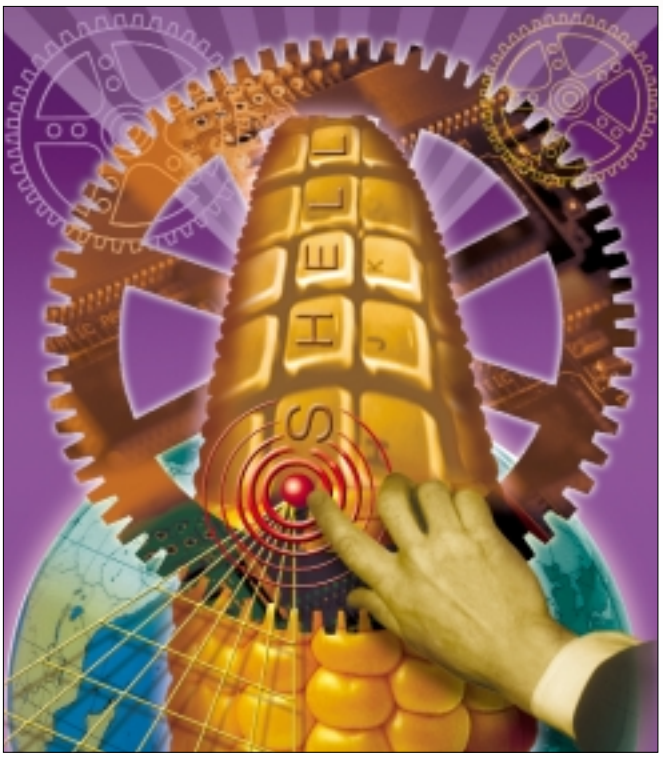
by Peter Collinson, **Hillside Systems**



BEK SHAKIROV

The Korn Shell

Well, nobody's perfect. I created an incorrect impression about the Korn shell in my article "Over revisited" (*SW Expert* August 2000). I said that the source for the most recent version of the Korn shell (`ksh93`) was not publicly available. This statement was incorrect, and I received email from a couple of people (one of them David Korn) telling me that the source has been available since 1999, pointing me to `www.kornshell.com`. The state of the art is that the program is owned by AT&T and Lucent, and is available in precompiled or source form for non-commercial and academic use. You'll find that `www.kornshell.com` links to the appropriate AT&T Web site so you can pull the relevant binaries or the source.

I thought that I might redress the balance somewhat by looking at what the Korn shell offers, and with that in mind, I began to poke about my Solaris 8 system to see exactly what Sun supplies. It seems that `/bin/ksh` on Solaris is not the most recent version of the shell. The version that you can pull from AT&T is `ksh93`, and has several enhancements that increase the power of the language. For some time, David Korn has been repositioning the shell; taking it from a program that is used to control program execution towards a powerful interpreted language. He's been aiming to create a "scripting language" that directly supports many of the programming structures and features that you will find in other UNIX super-tools, like `awk`. `ksh93` added several new programming features that has aided this growth.

However, Sun's `/bin/ksh` is an earlier version, `ksh88`. If you want to use `ksh93`, which is the version documented in most recent books, use `/usr/dt/bin/dtksh`. This program is a version of `ksh93` containing a set of native functions that give it the ability to interface directly to the X Window system. The idea is that you can write application programs for X in the `ksh` scripting language. Beware, `dtksh` is large, and you probably don't want to use it as your login shell.

If you do use `ksh` as your login shell, then what can it do for you? Most mortal users only scratch the surface of the power of any shell they use, and paradoxically, most of the usage of any shell is made by people simply typing commands and having them executed. So, I'm going to concentrate on the shell features that make your life easier.

I suppose that most of these features were inherited from `csh`, or perhaps it's more accurate to say re-implemented from the model originally supplied by the `csh` program. I suspect that porting in these `csh` facilities into a Bourne-like shell was the initial reason for David Korn's interest in hacking on the Bourne shell source to create `ksh`.

Job Control

As I've said in these pages before, the ability to perform job control was my original reason for starting to use the Berkeley releases of UNIX back in the

early 1980s. The idea is simple, but the ramifications are more complex. When you start a UNIX command from the command line:

```
$ command
```

it runs and you can do no further work until it terminates. You can make it abort its operation by typing Control-C on the keyboard which sends a special message, a *signal*, to the running process, and the running process will then die. You may not want to kill the process. You probably want the process to finish and wish that you had slapped an ampersand at the end of the command line when you typed it so that the process is running in the background. Of course, these days, you can start a new window that contains a terminal emulator, but back when you had only one window into the system from your terminal, you were generally stuck.

To stop a command temporarily, type Control-Z, which places the job that's attached to the terminal into a "stopped" state.

Job control is the solution to the problem. If you wish to stop a command temporarily, you type Control-Z. This places the job that's attached to the terminal into a "stopped" state, pausing it so you can restart it later. So, typing Control-Z into `ksh` while a command is running will look like:

```
$ command
^Z[1] + Stopped (SIGTSTP)      command
$
```

The mechanism sends a signal to the command, and the command will put itself into the "stopped" state. Notice that you've been given a new shell prompt, and can type a new command into the system. I've always felt that the use of the word "stop" is inappropriate because it implies a certain amount of finality that doesn't exist. The command is simply "paused" and can be restarted.

Once you type `^Z`, you have two options for dealing with the paused command. You can restart it in the foreground, by typing

```
$ fg
```

in which case the command takes control of the terminal, as before. Or you can restart it in the background:

```
$ bg
[1]      command&
$
```

which restarts the job, as if you'd typed an ampersand at the end of the line. The digit `1` in square brackets is a job number and can be used to refer to the jobs in the background. The `jobs` command can be used to see what's going on:

```
$ jobs
[1] + 9161     Running     command
$
```

You can move jobs in and out of the foreground and background by giving a job `id` preceded by the `%` character. I rarely do this. I mostly use job control to place some job into the background and start it running, freeing up the terminal window for some other task.

Most of the work to make job control happen is done by the shell using some special hooks in the kernel that allows the shell to manage groups of running processes. However, job control does impact on some applications. When we suspend a visual editor and restart it, we expect it to redraw the screen as if nothing had happened. A visual application needs to know that it's just been restarted, so that it can redraw its image for you to continue working.

Most shells support job control, in fact, the only one that doesn't is the Bourne shell. This is usually a source of annoyance to me when I have the system running in single user mode and need job control to manage the single terminal I am using running on the big white screen. One way around the problem is to start `ksh` or `bash` by hand by just typing the appropriate command. I always seem to forget the need to start a job control enabled shell until it's too late.

Aliasing

The C shell introduced the idea of aliasing commands, and `ksh` picked up the notion and also uses it intelligently to speed up command processing. The original idea was that there are often situations where you want to replace one command with another. I'm prone to type `dc`, for example, when I mean `cd`. I rarely use the `dc` command, and so feel happy to have the alias:

```
alias dc=cd
```

Now, when `dc` is detected as the first word of a command line, it will be replaced by substitution string `cd` and all will be well.

More complex uses are possible. For example, some people like to add the `-i` flag into the `rm` commands forcing a confirmation before any deletion is made. They will create an alias:

```
alias rm='/bin/rm -i'
```

Now whenever users type `rm`, the `rm` command is always presented with the `-i` option forcing the "Are you sure?" question. Actually, I don't feel that this actually works to stop accidental deletion of files. Users just become accustomed to confirming the deletion without looking and the file gets deleted anyway. However, different strokes for different folks.

I said that ksh uses the aliasing system to speed up command processing. When you have many directories in your PATH variable, it can take time to discover the location of a command. Korn shell retains the discovered file name as an alias whose key is the original command name. Korn shell calls this a "tracked alias." It automatically adds an alias such as

```
ls=/usr/bin/ls
```

into the alias list when the shell discovers the location of the command, in this case ls. Subsequent use of ls will use the alias system and invoke the command directly without searching for it. The alias system is acting as a command cache. To turn this feature on you need to say:

```
set -o trackall
```

You can see all the aliases that have been added by typing:

```
$ alias -t
cat=/usr/bin/cat
ls=/usr/bin/ls
```

Of course, with any aliasing system, it's a good idea to be able to remove any aliases, and you can do this with unalias.

It is possible to provide arguments to csh aliases, although the substitution mechanism is not very good. Aliases with optional arguments are not needed by ksh because you can provide shell functions that perform complex tasks. Those functions behave like external commands and can have arguments passed via the command line. Functions are enhanced in ksh93, where you can define variables that are only active within the function body, which is better programming practice. Earlier version of ksh followed the Bourne shell practice where all variables in functions have global scope, meaning that you have to be careful when you create a new script by raiding other scripts for functions.

Finally, if you want to have private commands, you can always use a regular command file stashed somewhere like a bin directory in your home. Incidentally, ksh implemented the tilde notation used in csh to mean the user's home directory, so a private bin directory can be referred to as ~/bin.

I think in the last few years, I've tended to put little command files and symbolic links that rename commands into my bin directory, mostly because their use is then independent from any shell I happen to be using.

You'll find that ksh provides a couple of really useful commands that can be used to find out what the command you are typing actually is. The first of these is the type command, which tells you how a word will be interpreted when used as a command:

```
$ type cd ls title
cd is a shell builtin
ls is a tracked alias for /usr/bin/ls
title is a function
```

The second is whence, which is useful for finding out the location of a command you are using:

```
$ whence cd ls ps
cd
/usr/bin/ls
/home/pc/bin/ps
```

which shows that I use my version of ps, which is a symlink to /usr/ucb/ps. Incidentally, whence -v is the type command. It's unclear why there are two shell builtin commands rather than one.

Starting the Shell

With aliases, private functions and the need to establish settings to make the shell do what we want, there has to be a way of running a private start-up script when the shell is started. The Bourne shell reads commands from /etc/profile and then from $HOME/.profile when a login session is started. The idea was that subsequent shell invocations would inherit settings of global variables from the process environment and didn't need to re-run the profile script. The author of csh realized there was a need for a tailoring script to be run whenever a shell was started and used ~/.cshrc to fulfill this function. The .login file is run when the user first logged in.

The Korn shell adopts a midway approach. It uses the profile files as before, but if the environment contains a definition of the ENV variable, then it is assumed to contain the name of a file that contains ksh commands to be run whenever the shell is started. The file can define aliases or functions. The whole approach of using a per-invocation start-up file works well in the windowing environment, where we login once and then start zillions of shells, each in their own window.

Another task for the various set up files is to establish what might be called "look and feel tailoring." For ages, I've set my shell prompt to be the name of the directory I am currently "in" and this is easy as pie in ksh. First, ksh maintains the current working directory in an environment variable PWD. If you are content with the complete path to the current directory appearing in the prompt, then add:

```
PS1="${PWD}$ "
```

and your main shell prompt will contain the whole pathname, a dollar and a space. The curly braces around the variable name are used to make sure that the shell parses the variable name correctly. I could have written:

```
PS1="$PWD$ "
```

The code looks confusing. The second dollar is printed as part of the prompt but looks as if it should be joined to PWD. I think the first form above is more readable.

However, I find printing out the whole pathname in the prompt is long-winded, especially when dealing with deep file hierarchies. All you'll need is the name of the directory you are in, and you will keep the path to the directory in your brain.

Last month I wrote about `basename` and `dirname`, which are two programs designed to split file paths into constituent bits. They are not needed in `ksh`. We can use one of the pattern matching and extraction mechanisms to alter the data that's contained in `PWD` before its printed.

```
PS1="${PWD##*/}$ "
```

This syntax probably looks enigmatic. However, all that has changed is that some magic characters have been added inside the `${PWD}` variable expansion that we used above. The magic characters are used to indicate that some processing should be done on the contents of the variable before it is used in this context. In fact, the Bourne shell supported several operators on variables and `ksh` added a few more.

To get a handle on what's going on here, let's take a simple example of this type of variable expansion. The simplest form fulfills the need of establishing default values for variables.

```
A=${A:-"Default Value"}
```

sets the variable `A` to the contents of `A`, unless `A` is empty or doesn't exist, in which case, `A` is set to the string `Default Value`. Note the structure of the conditional test, which contains a variable name `A`, a two-letter separator (`:-`) and a value. With this in mind, look at the `PS1` setting above. It has a variable name to test `PWD`, an operator (##) and a parameter which is a pattern `*/`. The operator makes `ksh` "remove the largest prefix pattern." The pattern is a star that matches anything, followed by a `/`. The effect is to remove all but the last directory name from `PWD`, so `/export/home/pc` becomes `pc` and the prompt will be output as `pc$` (with an extra space at the end, of course).

Line Editing

As a dreadful typist, I like to have some way of putting text into the machine in a way that can be easily altered without deleting the characters I already typed. The Korn shell treats the input line as an editable one line text window using either `emacs` style key-strokes or `vi` commands. You select your preferred style by saying:

```
set -o vi
```

or

```
set -o emacs
```

in your start-up file. You can also use `gmacs` mode, which is the same as `emacs`, but the Control-T (transpose character) action works in a slightly different way. If you are not a `vi` user, you will find the `emacs` mode is somewhat easier to learn. Actually, you will find that learning the `emacs` keystrokes is always useful because they will work in any text input box that you use for the X windows system.

Sadly, `ksh` doesn't permit the use of the arrow keys to move

about the line, so you should learn some of the emacs default keystrokes to use the editing facilities. Most of the simpler functions are achieved by using control keys. ^B moves back one character (mnemonic: B is Back), ^F moves on one character (mnemonic: F is Forward), ^E moves to the end of the line (mnemonic: E is End), and ^A moves to the start of the line (mnemonic: I'm baffled). Your normal delete character key, (usually Delete or backspace) will delete characters before the cursor, and ^D deletes the character *under* the cursor. That's about all you need to know to get started. There are many other commands, usually two character command sequences. Each sequence is usually the Escape character followed by another letter. Escape followed by F will move forward one word, for example, and Escape followed by B will move back one word. I offer no prizes of guessing how to delete the last word. The command sequence Escape followed by a letter is called a Meta command, and shown as Meta-F (say), or simply M-F in emacs-speak.

If you use the go up to previous line command, ^P, then ksh will scroll back up through your command history allowing you to re-use previous command lines. Control-N takes you down through the history, going forward to the last command executed. As you go back up the history list, each line is displayed after the current prompt, so pressing return will activate the command, and of course, you can use the line editing commands to change the command details before you commit to using it. There are also ways of searching the history list looking for particular commands that you've previously entered.

Tied in with the input line editing is the ability to perform filename completion. When typing a filename, you type the first few letters and you can get the shell to complete it for you. You need to type a unique subset of the name and press Escape-Escape. The shell will not advance the completion if it detects the stem it's holding is not unique in the directory. To see what is causing the holdup, press Escape-= to obtain a list of alternatives. You then need to type enough text to select the filename that you want, before pressing Escape-Escape again. I find the command name and filename completion mechanism supported by bash is much easier to use, and this is one reason why I tend to talk to the machine using bash.

Finally

I've really run out of space without touching on many of the aspects of ksh that make life easier for writing scripts. Well, that is understandable–people write whole books on the topic. I have tried to concentrate on the aspects of the program that affect the user at the terminal, rather than those that are aimed at supporting the programmer. To find out more about ksh, I suggest you visit http://www.kornshell.com.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* pc@cpg.com.