(Not So) Stupid Shell Tricks

By Jerry Peek

Graphical environments have lots of fun add-ons, such as skins, themes, and more. If you use a terminal or other shell-based applications, it's enough to make you feel left out. This month's "Power Tools" column aims to change that.

You might call this column "Stupid Shell tricks," but that's not quite right because we'll see useful techniques!

For example, displaying a shell prompt in multiple colors may not do much for your resume, but knowing how to make important text appear in bright red can help you spot serious conditions. We'll also look at shell quoting and interpretation, escape sequences, and the usual grab-bag of curiosities and oddities that are hard to find, but good to know. Let's dig in — for fun and profit.

Prompt Preview

When a shell is ready for your next command, it prints a prompt and waits. Left-side prompts usually end with `$` on Bourne-type shells like *bash* and *ksh*, with `%` on *csh* and *zsh*, and `>` on *tcsh*. Shells running as the superuser, root, prompt with `#` to remind you of your wizardly powers.

Most shells have several left-side prompts — for unfinished command lines, for debugging output, and more. The *zsh* and *tcsh* shells can also prompt on the right side. These right-side prompts are handy for periodic notices, like the "high load" warning that you'll see later in a left-side prompt. This column won't cover any of these right-side prompts except to say that, in general, you can set them in the same way you set the primary (left-side) prompt.

Why reconfigure your shell prompt? Well, while it's fun and it keeps you from doing real work, you can make a prompt show information that you often want, like the current directory name or the current time. (The time can be useful if you look back to see what time you ran a particular command.) A prompt can also change dynamically — for instance, to remind you when you need to do something, to show that new users have logged in, and to warn of a high system or network load. Last but not least, you can "hijack" the prompt mechanism to do other things, like popping up new windows.

Storing the Prompt

Bourne-type shells store their primary prompt string in the shell variable `PS1` (the name ends with the digit `1`.) The *tcsh* shell uses `prompt` — and *zsh* uses either.

You can set these variables from the command line, experimenting until you get a prompt you like. Changes there will only affect the current shell invocation, though. To make your new prompt appear in all shells, you have to set the prompt string in a shell setup file in your home directory — or in the system-wide setup file, which is often somewhere under */etc*. You'll probably want to set it from a setup file that's read by all instances of your shell — not just login shells. For *bash* that's *.bashrc;* for *tcsh* it's *.cshrc* or *.tcshrc;* for *ksh* it's the filename stored in the *ENV* variable; and for *zsh*, check your manual page or just look around (there are several possibilities). In original Bourne shells, like *ash*, set `PS1` from your *.profile* file and use `export PS1` to pass *PS1* to subshells through the environment.

Setting the Prompt

The simplest prompt is just a character — with a single space afterward to make the command line stand out. The prompt must be a single string, so put its value inside quotes to keep the shell from breaking the string at spaces. The sidebar "Which Quote is Which?" explains what kind of quote to use.

Let's start with a basic prompt: the current directory (from the shell variable `PWD` or `cwd`) and a `$` or a `>`. These examples seem to work at first, but they have a problem that you'll see as you change directories around the filesystem:

```
bash$ PS1="$PWD $ "
/home/jpeek $ cd /usr/local
/home/jpeek $

tcsh> set prompt = "$cwd > "
/home/jpeek > cd /usr/local
/home/jpeek >
```

The current directory in the prompt isn't changing. Why? It's because the prompt string was surrounded with double quotes. As the sidebar "Which Quote is Which?" explains, the shell expands shell variables inside double quotes *before* executing the command line. So the prompt contains the value of `$PWD` *as it was at the time the* `PS1` *or* `prompt` *variable was stored!* (To confirm this, use the command *set* to see current settings of all shell variables. Its output will show that `PS1` and `prompt` both contain a literal string like `/home/jpeek`.)

How can we fix that? Some shells, like *bash*, re-evaluate the prompt string before emitting each prompt. In those shells, you can store special characters (like `$` and `'`) inside the prompt string, and each one is expanded before each prompt. As the sidebar below shows, you can preserve all special characters by putting them inside single quotes:

```
bash$ PS1='`date` $PWD$ '
Wed Aug 27 09:59:48 PDT 2003 /u/zoe$ cd /tmp
Wed Aug 27 10:00:04 PDT 2003 /tmp$
```

WHICH QUOTE IS WHICH?

Surrounding a string of characters with quote marks — `'`, the *single quote* mark, or `"`, the *double quote* — tells the shell to *disable the special meaning of some or all of the characters between the quotes.* In a nutshell, that's what quoting does!

An opening single quote disables all special characters until the next single quote character is found. (There is one exception: in *tcsh,* single quotes don't disable `!`, the history substitution character.) Double quotes match in the same way as single quotes, but they allow most substitutions: variables (like `$USER`), commands (like `` `date` `` or `$(date)`, and so on. Both types of quotes disable the word-separator characters SPACE, TAB and NEWLINE — which keeps the shell from breaking the quoted string into separate arguments. (That's important when you set a prompt because a prompt must be a single long string, even if it contains spaces or newlines.) Let's see some quoting examples using *tcsh:*

```
% echo "I am $USER."
I am jpeek.
% echo 'I am $USER.'
I am $USER.
% echo "It's `date`, $USER."
It's Wed Aug 27 08:17:24 PDT 2003, jpeek.
% echo 'It's `date`, $USER.'
Unmatched '.
% echo "It'"'s `date`, $USER.'
It's `date`, $USER.
%
```

The last few examples show that double quotes can surround single quotes (and disable their special meaning), but single quotes can't surround other single quotes. Why? After the first quote is found, the next quote of the same kind, reading from left to right, makes a pair. (In other words: quotes of the same type do not nest.)

The last example passes a literal single quote character (in the word *It's*) to the *echo* command, while also surrounding `` `date` `` and `$USER` with single quote characters to disable the `` ` `` and `$` characters. How? We surrounded the first three characters (`It'`) with double quotes and the rest with single quotes. Also notice that there are no unquoted spaces; this doesn't matter to *echo,* but it's important when you set a prompt.

These terse examples show almost everything about quoting. The rest is practice, checking the rules (from this sidebar) and trying more examples.

But *tcsh* and *csh* would prompt with a literal "`$cwd >`" because they don't re-interpret the `prompt` variable (and expand special characters like `$` and `` ` ``) before emitting each prompt.

The standard fix is to define an alias named something like *setprompt* that sets the `prompt` variable to a literal string. Then, each time you want to change the text in the prompt — say, after *cd, pushd,* or *popd* — call the *setprompt* alias. You could define aliases like these:

```
alias setprompt 'set prompt="$cwd > "'
alias cd '""cd \!* && setprompt'
alias pushd '""pushd \!* && setprompt'
alias popd '""popd \!* && setprompt'
```

You can also use the *tcsh* variable named `precmd` to reset the `prompt` variable before each prompt, as we'll see later. But luckily, there's an easier way to handle most cases.

You usually don't need to go through these quoting and alias nightmares because most shells also understand special *format strings* that you can store in the prompt string. Those strings expand into common things you'd want to put in your prompt, like the current directory. Your shell's manual page should list its format strings.

Using Prompt Format Strings

A basic prompt might simply show the current directory, but the same techniques can also make prompts with a lot of information. Adding lots of stuff to a prompt can take a lot of room across the screen, leaving less room for command lines. You can solve that with a *multiline* prompt that puts extra stuff on separate lines, leaving the final line almost empty.

So let's go wild and make a fairly complex three-line prompt. The pattern is shown below:

```
username@hostname time directory
historynumber X
```

The prompt starts with a blank line to separate it from the previous command's output. The *`X`* should be `$` on *bash,* `>` on *tcsh,* and `#` on any shell run as root. For example:

```
jpeek@kumquat 15:39:36 ~/articles
103$ cd /usr/local

jpeek@kumquat 15:40:01 /usr/local
104$
```

That's simple to program on *bash.* This does the trick:

```
PS1='\n\u@\h \t \w\n\!\$ '
```

The first `\n` in `PS1` makes a leading blank line, and the second `\n` separates the two lines of text. The final `\$` outputs `$` for non-superusers or `#` if you're root.

It's trickier in *tcsh* because you have to embed a newline character in the `prompt` variable — which, in *tcsh*, requires a backslash (`\`) before the literal newline:

```
set prompt = '\
%n@%m %P %~\
%h%# '
```

Z Shell syntax looks a lot like *tcsh* — although, as usual, *zsh* has many more features. The Korn shell doesn't support these prompt format strings, but you can simulate them by using shell and environment variables. For a little shell programming adventure, *Listing One* shows the example shown immediately above rewritten for *ksh*.

Because *ksh* evaluates the stored prompt string (which is set at the end of the code), and `PS1` contains a call to the `shortdir()` function (which is defined earlier), that means `shortdir()` is called each time a prompt is printed. `shortdir()` outputs the current directory pathname; if you're under your home directory, its name is replaced by a `~` character. `shortdir()` also uses the *ksh* operator `${variable#pattern}`, which edits the value of `$PWD` (the absolute pathname of the current directory) to remove the leading value of `$HOME` and the trailing slash (`/`).

As we store the prompt string, the first and last parts are inside double quotes, so the values of (`$USER`, `$hostbase`, and `$promptend`) are expanded as the prompt is stored. The rest of the prompt — the calls to *date* and `shortdir()`, and the history number `!` — isn't interpreted until just before each prompt is printed. (Instead of surrounding some parts with double quotes, you can simply put single quotes around everything — and *ksh* would interpret all special characters before every prompt. But there are times when you want to set a particular value once and not have it re-interpreted before every prompt; that's where this quote-switching technique is useful.) You may notice that *ksh* doesn't require a backslash before literal newlines inside quotes, and calling `date +%T` makes *date* output only the time of day.

To keep things simple from here on, we'll only show *bash*. For other shells, please compare the *bash* examples to your shell's *man* page — and use aliases like *setprompt*, or setups like *Listing One*, as needed.

LISTING ONE: A multi-line prompt for *ksh*

```
# Get short hostname (up to first '.'):
hostname=$(hostname)
hostbase=${hostname%%.*}
# The shortdir function shortens $PWD. It
# converts $HOME to ~ and $HOME/dir to ~/dir.
# It writes the result to standard output.
shortdir() {
  case "$PWD" in
  $HOME) echo '~' ;;
  ${HOME}*) echo "~/${PWD#$HOME/}";;
  "") /bin/pwd ;;
  *) echo "$PWD" ;;
  esac
}
# Root prompt ends with '#', others with '$':
if [ $(id -u) -eq 0 ]
  then promptend='#'
  else promptend='$'
fi
# Set the three-line prompt:
PS1="
$USER@$hostbase"' $(date +%T) $(shortdir)'"
!$promptend "
```

Beyond Prompt Format Strings

If you're new to shells, the Korn shell example at the end of the previous section may seem to stretch the limits of what's reasonable. But a shell is actually a programming language interpreter, where the language is Linux command lines. There's no reason not to use the shell's programmability on its interactive features — like prompts. So let's keep on digging!

Built-in format strings can't put everything you might want into a prompt. For instance, do you use the shell's directory stack (described in the Power Tip at the end of the February, 2003 column, available online at http://www.linux-mag.com/2003-02/power_01.html)? You might want to put the stack list in your prompt, so you don't have to check it constantly. That's as simple as replacing the `\w` in the *bash* prompt string with `$(dirs)`, which uses command substitution (the `$(cmd)` operator) to run the *dirs* command and replace the command with its output:

```
PS1='\n\u@\h \t $(dirs)\n\!\$ '
```

Let's make another version where each item on the stack is numbered. (This is especially handy for commands like `pushd +2`, which changes to the second directory on the stack.) To simplify things a little, let's get rid of the time — which gives prompts that look like this:

```
jpeek@kumquat 0=/usr/proj/foo 1=~ 2=~/bin
107$
```

Even if you don't want a prompt like that, the way to make it is worth a look! See *Listing Two*. We've changed `PS1` by calling the *do_dirs* function instead of the *dirs* command (and, as we said, removing the `\t` that gave the current

LISTING TWO: Building a *bash* prompt with a shell function

```
do_dirs() {
 local n dir
 dirs -v |
 while read n dir
 do
 echo -n " $n=$dir"
 done
}

PS1='\n\u@\h$(do_dirs)\n\!\$ '
```

time). The `do_dirs()` function calls *dirs* with the `-v` option, which outputs a numbered list of directories in the stack, one per line. We pipe that output to a *while* loop. The loop-controlling *read* command reads the `dirs -v` output, line by line from the pipe, assigning the number to the shell variable *n* and the directory path to the variable *dir.* In the body of the loop, for each directory, we output a space character, followed by the number and the path, with an equal sign between. The `-n` option tells *echo* not to output a newline — so the entire output of `do_dirs()` is a single line, like `0=/usr/proj/foo 1=~ 2=~/bin`, which appears in the middle of the prompt string.

You might wonder why we used a semi-bizarre shell function instead of, say, writing this in Perl. The main reason is that every command we used in *Listing Two* is built in to the shell, which means the prompt will be set almost instantaneously — even on a busy machine. Starting a new process (for Perl) before each prompt could make your shell seem sluggish.

The shell function also uses a *while* loop with its standard input redirected. This is a very handy technique in Bourne-type shells. As the *bash* manpage explains, *while* is a *compound command*. Most compound commands can be preceded with a "feeder" command and a pipe, as we've done in `do_dirs()`. That means all commands within the compound command will take their standard input from the pipe. In our *while* loop, the only command that reads its *stdin* is the *read* command. Each time we call *read,* it reads a single line from its standard input.

So this *while* loop reads the `dirs -v` output line-by-line.

Another way to redirect the input of a compound command is with the shell's < (less-than) operator, which redirects *stdin* from a file. Put that operator, followed by a filename, at the end — in a *while* loop, that's just after the word `done`:

```
while read variables
...
done < filename
```

Escape Sequences, Pre-prompt Commands

You can also put color, boldface/bright, or blinking text in your prompt. For instance, you could add the word *root* in reverse video to the superuser prompt.

To do this, surround the "special" text with escape sequences that control the terminal's character mode. (There's an introduction to escape sequences online at http://www.linuxmagazine.com/downloads/2003-09/power/escape_sequences.html.)

Your shell needs to know how wide the prompt is — so, for instance, you can't "backspace over" the prompt while you edit the command line. Because escape sequences don't output visible characters (because they don't take room on the screen), you need to tell the shell not to "count" those characters. In *bash*, do that by surrounding each escape sequence with `\[` and `\]`. In *zsh* and *tcsh*, surround an escape sequence with `%{` and `%}` (though you don't need to do this with *zsh*/*tcsh* operators like `%S` that set standout, boldfacing, and underlining modes in the prompt).

Maybe you have accounts on multiple workstations. One of them — the master server — is special. You want to make that host have a special prompt with the hostname in reverse video. You could put the following code into your *.bashrc* file.

```
if [ "$HOSTNAME" == "server" ]
  then PS1='\[\e[7m\]\h\[\e[0m\] \$ '
  else PS1='\h \$ '
fi
```

If the hostname is *server*, we set the special `PS1` string. It starts with a "start escape sequence" marker, then an ESC character (which, in a *bash* version 2 prompt, is written `\e`), then the reverse-video escape sequence and the "end escape sequence" marker. After the hostname (which *bash* expands from `\h`), we use another escape sequence to cancel reverse video. The prompt ends with `\$`, as explained earlier.

Here's another escape-sequence example. You might want

LISTING THREE: Red text in *bash* prompt when load reaches 5

```
setprompt() {
 local load etc
 read load etc < /proc/loadavg
 if [ ${load%.*} -ge 5 ]
 then PS1="\n\u@\h \w \[\e[31m\]HIGH LOAD
$load\[\e[30m\]\n\!\$ "
 else PS1="\n\u@\h \w\n\!\$ "
 fi
}

PROMPT_COMMAND=setprompt
```

LISTING FOUR: *bash* pre-prompt function with a timer

```
wrister() {
  message="Time for a break!"
  interval=600 # number of seconds between mes-
sages
  : ${lastseconds=0} # set lastseconds to 0 if
not set yet
  if let "$SECONDS - $lastseconds > $interval"
  then
    if [ -n "$DISPLAY" ]
    then
      xmessage -nearmouse -bg blue "$message"
    else
      # Ring bell, output blue $message, then
change to black:
    echo -e "\a\e[34m${message}\e[30m"
    read dummy
    fi
  lastseconds=$SECONDS
  fi
}
PROMPT_COMMAND=wrister
```

part of your prompt to be highlighted when something happens. *Listing Three* (pg. 37) shows how to add the red words HIGH LOAD, and the one-minute load average, to your *bash* prompt when the system load average is 5 or more.

The `setprompt()` function declares two local variables (used only within the function). Next it reads the system's one-minute load average from */proc/loadavg* into the variable named `load`. (The rest of the contents are read into the variable `etc`, which we don't use — but is required so *read* will split the input line.) The *bash* operator `${`*`variable%pattern`*`}` strips the string matched by `pattern` from the end of the shell *variable*. Here we're removing the dot, and everything after it, from the load average; this removes the decimal point and digits to leave an integer. The script tests the integer and if it's greater than or equal to 5, it sets a prompt that has the red words HIGH LOAD, plus the complete load average number, at the end of its second line. If the load average is under 5, the code sets a normal prompt.

By storing the name of this function in the *bash* `PROMPT_COMMAND` variable, the function runs *before* each shell prompt is printed. (In *zsh* and *tcsh*, you'd use the `precmd` variable instead.)

Although it's called `PROMPT_COMMAND`, that variable doesn't need to be used only to set the prompt. It can run any command you want (though, of course, you should use discretion: do you really want to rebuild the kernel before every prompt?). *Listing Four* shows a shell function named `wrister()` that reminds you every ten minutes to give your wrists a break from typing. (This is worthwhile only if you're using the shell a lot; the time interval is only checked before a shell prompt is printed.) The function remembers the value of the *bash* variable `SECONDS`, which counts up from 0 each time you start a shell running, and compares it to the current `SECONDS`. If you're using X (if the `DISPLAY` environment variable is set), it pops up an *xmessage* window; otherwise, it rings the terminal bell, prints a message in blue and waits for you to press the RETURN key.

The shell won't print its usual prompt until `wrister()` exits. If you don't want that, you could omit the `read dummy` — and also put *xmessage* into the background, like this:

```
(xmessage -nearmouse -bg blue "$message" &)
```

The subshell operators `()` with the ampersand `&` inside "disown" (detach) the background *xmessage* job. That prevents shell job-status notices (like `[1] Done xmessage ...`) after *xmessage* exits.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.

POWER TIP: KEEPING A PROCESS ALIVE

A Linux *daemon* is a process that runs on its own — typically detached from any shell, as we saw in the final *xmessage* example. You can start daemons from one of the system */etc/rc** files that are read when the system reboots — assuming you have superuser access, that is. But that won't restart daemons that exit for some other reason — such as exhaustion — after your Linux system has gone a year between reboots.

An easy answer is a *cron* job that tries to restart a daemon every so often, like once an hour or once a day. (If the system has just rebooted, the *cron* job will start the daemon.) This trick depends on the daemon having a "sanity check" to prevent itself from running twice: a lockfile, scanning *ps* output for its name and/or the PID number saved the last time it started, etc.

For instance, here's a *crontab* entry that restarts the SETI daemon seven minutes past midnight and noon:

```
7 0,12 * * * (cd seti && exec ./setiathome &)
```

The `&&` operator aborts if *cd* fails. The *exec* replaces *cron's* shell process with the *setiathome* process, which means the shell doesn't have to hang around, doing nothing except using some memory and a slot in the system's process table. Some shells will do the *exec* automatically when they see there are no more commands to run. You can find out what your shell does by omitting the *exec,* then running `ps lx` after *cron* starts the daemon; see whether the daemon has a PPID (parent process ID) of 1. If the PPID is 1 (the system's *init* process), there's no shell waiting for the daemon to finish.