Bash Prompt HOWTO

Table of Contents

Table of Contents

Bash Prompt HOWTO

Giles Orr, giles@interlog.com

v0.76 31 December 1999

Creating and controlling terminal and xterm prompts is discussed, including incorporating standard escape sequences to give username, current working directory, time, etc. Further suggestions are made on how to modify xterm title bars, use external functions to provide prompt information, and how to use ANSI colours.

1.Introduction and Administrivia

1.1 Requirements

You will need Bash. The default version for most distributions is either 1.14.7, or 2.0.x. 1.14.7 was the standard for years, but is slowly being replaced. I've been using Bash 2.0.x for quite a while now, but almost all code presented here should work under 1.14.7. If I'm aware of a problem, I'll mention it. You can check your Bash version by typing `echo $BASH_VERSION` at the prompt. On my machine, it responds with `2.03.6(1)−release`.

Shell programming experience would be good, but isn't essential: the more you know, the more complex the prompts you'll be able to create. I assume a basic knowledge of shell programming and Unix utilities as I go through this tutorial. However, my own shell programming skills are limited, so I give a lot of examples and explanation that may appear unnecessary to an experienced shell programmer.

1.2 How To Use This Document

I include a lot of examples and explanatory text. Different parts will be of varying usefulness to different people. This has grown long enough that reading it straight through would be difficult − just read the sections you need, backtrack as necessary.

1.3 Translations

*Japanese:*http://www.jf.linux.or.jp/JF/JF−ftp/other−formats/Bash−Prompt/Bash−Prompt−HOWTO.html, provided by Akira Endo, akendo@t3.rim.or.jp.

German: translation is in progress by Thomas Keil, thomas@h−preissler.de.

Italian: by Daniel Dui, ddui@iee.org, available at http://www.crs4.it/~dui/linux.html.

Portugese: translation is in progress by Mário Gamito, mario.gamito@mail.telepac.pt.

Spanish: translation by Iosu Santurtún iosu@bigfoot.com at http://mipagina.euskaltel.es/iosus/linux/Bash−Prompt−HOWTO.html.

Dutch: translation is in progress by Ellen Bokhorst elboh@gironet.nl, and it will be available at http://www.nl.linux.org/doc/HOWTO.

Chinese: translation in progress by Allen Huang lancelot@tomail.com.tw. I will include a URL when I have it.

Many thanks to all of them! URLs will be included as they're available.

If you are working on a translation, please notify me − especially if it's available at a linkable URL. Thanks.

1.4 Problems

This is a list of problems I've noticed while programming prompts. Don't start reading here, and don't let this list discourage you − these are mostly quite minor details. Just check back if you run into anything odd.

- Many Bash features (such as math within $(()) among others) are compile time options. If you're

using a binary distribution such as comes with a standard Linux distribution, all such features should be compiled in. But if you're working on someone else's system, this is worth keeping in mind if something you expected to work doesn't. Some notes about this in *Learning the Bash Shell*, p.260−262.
- The terminal screen manager "screen" doesn't always get along with ANSI colours. I'm not a screen expert, unfortunately. My current version of screen (3.7.6−1, an RPM package) seems to work well in all cases, but I've seen occasions where screen reduced all prompt colours to the standard foreground colour in X terminals. This doesn't appear to be a problem on the console.
- Xdefaults files can override colours. Look in ~/.Xdefaults for lines referring to XTerm*background and XTerm*foreground (or possibly XTerm*Background and XTerm*Foreground).
- One of the prompts mentioned in this document uses the output of "jobs" − as discussed at that time, "jobs" output to a pipe is broken in Bash 2.02.
- ANSI cursor movement escape sequences aren't all implemented in all X terminals. That's discussed in its own section.
- Some nice looking pseudo−graphics can be created by using a VGA font rather than standard Linux fonts. Unfortunately, these effects look awful if you don't use a VGA font, and there's no way to detect within a term what kind of font it's using.
- Bash 2.0+ is out, and it incorporates some new features, and changes some behaviour. Things that work under 1.14.7 don't necessarily work the same under 2.0+, or vice versa.

1.5 Send Me Comments and Suggestions

This is a learning experience for me. I've come to know a fair bit about what can be done to create interesting and useful Bash Prompts, but I need your input to correct and improve this document. I've tried to check suggestions I make against different versions of Bash (2.0x and 1.14.7), but let me know of any incompatibilities you find.

The latest version of this document should always be available at http://www.interlog.com/~giles/bashprompt/. Please check this out, and feel free to e−mail me at giles@interlog.com with suggestions.

I use the Linux Documentation Project HOWTOs almost exclusively in the HTML format, so when I convert this from SGML (its native format), HTML is the only format I check thoroughly. If there are problems with other formats, I may not know about them, and I'd appreciate a note about them.

1.6 Credits

In producing this document, I have borrowed heavily from the work of the Bashprompt project at http://bash.current.nu/. Other sources used include the *xterm Title mini−HOWTO* by Ric Lister, available at http://sunsite.unc.edu/LDP/HOWTO/mini/Xterm−Title.html, *Ansi Prompts* by Keebler, available at http://www.ncal.verio.com/~keebler/ansi.html, *How to make a Bash Prompt Theme* by Stephen Webb, available at http://bash.current.nu/bash/HOWTO.html, and *X ANSI Fonts* by Stumpy, available at http://home.earthlink.net/~us5zahns/enl/ansifont.html.

Also of immense help were several conversations and e−mails from Dan, an ex−co−worker at Georgia College & State University, whose knowledge of Unix far exceeds mine. He's given me several excellent suggestions, and ideas of his have led to some interesting prompts.

Three books that have been very useful while programming prompts are *Linux in a Nutshell* by Jessica Heckman Perry (O'Reilly, 1997), *Learning the Bash Shell* by Cameron Newham and Bill Rosenblatt (O'Reilly, 2nd. ed., 1998) and *Unix Shell Programming* by Lowell Jay Arthur (Wiley, 1986. This is the first edition, the fourth came out in 1997).

1.7 Copyright and Disclaimer

This document is copyright 1998−1999 by Giles Orr. You are encouraged to redistribute it. You may not modify this document (see the section on contacting me: I incorporate most changes recommended by readers). Please contact me if you're interested in doing a translation.

This document is available for free, and, while I have done the best I can to make it accurate and up to date, I take no responsibility for any problems you may encounter resulting from the use of this document.

2.Bash and Bash Prompts

2.1 What is Bash?

Descended from the Bourne Shell, Bash is a GNU product, the "**B**ourne **A**gain **SH**ell." It's the standard command line interface on most Linux machines. It excels at interactivity, supporting command line editing, completion, and recall. It also supports configurable prompts − most people realize this, but don't know how much can be done.

2.2 What Can Tweaking Your Bash Prompt Do For You?

Most Linux systems have a default prompt in one colour (usually gray) that tells you your user name, the name of the machine you're working on, and some indication of your current working directory. This is all useful information, but you can do much more with the prompt: all sorts of information can be displayed (tty number, time, date, load, number of users, uptime ...) and the prompt can use ANSI colours, either to make it look interesting, or to make certain information stand out. You can also manipulate the title bar of an Xterm to reflect some of this information.

2.3 Why Bother?

Beyond looking cool, it's often useful to keep track of system information. One idea that I know appeals to some people is that it makes it possible to put prompts on different machines in different colours. If you have several Xterms open on several different machines, or if you tend to forget what machine you're working on and delete the wrong files (or shut down the server instead of the workstation), you'll find this a great way to remember what machine you're on.

For myself, I like the utility of having information about my machine and work environment available all the time. And I like the challenge of trying to figure out how to put the maximum amount of information into the smallest possible space while maintaining readability.

2.4 The First Step

The appearance of the prompt is governed by the shell variable PS1. Command continuations are indicated by the PS2 string, which can be modified in exactly the same ways discussed here − since controlling it is exactly the same, and it isn't as "interesting," I'll mostly be modifying the PS1 string. (There are also PS3 and PS4 strings. These are never seen by the average user − see the Bash man page if you're interested in their purpose.) To change the way the prompt looks, you change the PS1 variable. For experimentation purposes, you can enter the PS1 strings directly at the prompt, and see the results immediately (this only affects your current session, and the changes go away when you log out). If you want to make a change to the prompt permanent, look at the section below Setting the PS? Strings Permanently.

Before we get started, it's important to remember that the PS1 string is stored in the environment like any other environment variable. If you modify it at the command line, your prompt will change. Before you make any changes, you can save your current prompt to another environment variable:

```
[giles@nikola giles]$ SAVE=$PS1
[giles@nikola giles]$
```

The simplest prompt would be a single character, such as:

```
[giles@nikola giles]$ PS1=$
$ls
bin   mail
$
```

This demonstrates the best way to experiment with basic prompts, entering them at the command line. Notice that the text entered by the user appears immediately after the prompt: I prefer to use

```
$PS1="$ "
$ ls
bin   mail
$
```

which forces a space after the prompt, making it more readable. To restore your original prompt, just call up the variable you stored:

```
$ PS1=$SAVE
[giles@nikola giles]$
```

2.5 Bash Prompt Escape Sequences

There are a lot of escape sequences offered by the Bash shell for insertion in the prompt. From the Bash 2.02 man page:

```
When executing interactively, bash  displays  the  primary
prompt  PS1  when  it  is ready to read a command, and the
secondary prompt PS2 when it needs more input to  complete
a  command.   Bash  allows these prompt strings to be cus−
tomized by inserting a number of backslash−escaped special
characters that are decoded as follows:
       \a     an ASCII bell character (07)
       \d     the  date  in  "Weekday  Month  Date" format
              (e.g., "Tue May 26")
       \e     an ASCII escape character (033)
       \h     the hostname up to the first `.'
       \H     the hostname
       \n     newline
       \r     carriage return
       \s     the name of the shell, the  basename  of  $0
              (the portion following the final slash)
       \t     the current time in 24−hour HH:MM:SS format
       \T     the current time in 12−hour HH:MM:SS format
       \@     the current time in 12−hour am/pm format
       \u     the username of the current user
       \v     the version of bash (e.g., 2.00)
       \V     the  release  of  bash, version + patchlevel
              (e.g., 2.00.0)
       \w     the current working directory
       \W     the basename of the current  working  direc−
              tory
       \!     the history number of this command
       \#     the command number of this command
       \$     if  the effective UID is 0, a #, otherwise a
              $
       \nnn   the character  corresponding  to  the  octal
              number nnn
       \\     a backslash
```

```
        \[     begin a sequence of non−printing characters,
               which could be used to embed a terminal con−
               trol sequence into the prompt
        \]     end a sequence of non−printing characters
```

Continuing where we left off:

```
[giles@nikola giles]$ PS1="\u@\h \W> "
giles@nikola giles> ls
bin   mail
giles@nikola giles>
```

This is similar to the default on most Linux distributions. I wanted a slightly different appearance, so I changed this to:

```
giles@nikola giles> PS1="[\t][\u@\h:\w]\$ "
[21:52:01][giles@nikola:~]$ ls
bin   mail
[21:52:15][giles@nikola:~]$
```

2.6 Setting the PS? Strings Permanently

Various people and distributions set their PS? strings in different places. The most common places are /etc/profile, /etc/bashrc, ~/.bash_profile, and ~/.bashrc . Johan Kullstam (johan19@idt.net) writes:

> the PS1 string should be set in .bashrc. this is because non−interactive bashes go out of their way to unset PS1. the bash man page tells how the presence or absence of PS1 is a good way of knowing whether one is in an interactive vs non−interactive (ie script) bash session.
>
> the way i realized this is that startx is a bash script. what this means is, startx will wipe out your prompt. when you set PS1 in .profile (or .bash_profile), login at console, fire up X via startx, your PS1 gets nuked in the process leaving you with the default prompt.
>
> one workaround is to launch xterms and rxvts with the −ls option to force them to read .profile. but any time a shell is called via a non−interactive shell−script middleman PS1 is lost. system(3) uses sh −c which if sh is bash will kill PS1. a better way is to place the PS1 definition in .bashrc. this is read every time bash starts and is where interactive things − eg PS1 should go.

therefore it should be stressed that PS1=..blah.. should be in .bashrc and not .profile.

I tried to duplicate the problem he explains, and encountered a different one: my PROMPT_COMMAND variable (which will be introduced later) was blown away. My knowledge in this area is somewhat shaky, so I'm going to go with what Johan says.

3.Bash Programming and Shell Scripts

3.1 Variables

I'm not going to try to explain all the details of Bash scripting in a section of this HOWTO, just the details pertaining to prompts. If you want to know more about shell programming and Bash in general, I highly recommend *Learning the Bash Shell* by Cameron Newham and Bill Rosenblatt (O'Reilly, 1998). Oddly, my copy of this book is quite frayed. Again, I'm going to assume that you know a fair bit about Bash already. You can skip this section if you're only looking for the basics, but remember it and refer back if you proceed much farther.

Variables in Bash are assigned much as they are in any programming language:

```
testvar=5
foo=zen
bar="bash prompt"
```

Quotes are only needed in an assignment if a space (or special character, discussed shortly) is a part of the variable.

Variables are referenced slightly differently than they are assigned:

```
> echo $testvar
5
> echo $foo
zen
> echo ${bar}
bash prompt
> echo $NotAssigned
```

```
>
```

A variable can be referred to as **$bar** or **${bar}**. The braces are useful when it is unclear what is being referenced: if I write **$barley** do I mean **${bar}ley** or **${barley}**? Note also that referencing a value that hasn't been assigned doesn't generate an error, instead returning nothing.

3.2 Quotes and Special Characters

If you wish to include a special character in a variable, you will have to quote it differently:

```
> newvar=$testvar
> echo $newvar
5
> newvar="$testvar"
> echo $newvar
5
> newvar='$testvar'
> echo $newvar
$testvar
> newvar=\$testvar
> echo $newvar
$testvar
>
```

The dollar sign isn't the only character that's special to the Bash shell, but it's a simple example. An interesting step we can take to make use of assigning a variable name to another variable name is to use **eval** to dereference the stored variable name:

```
> echo $testvar
5
> echo $newvar
$testvar
> eval echo $newvar
5
>
```

Normally, the shell does only one round of substitutions on the expression it is evaluating: if you say **echo $newvar** the shell will only go so far as to determine that **$newvar** is equal to the text string **$testvar**, it won't evaluate what **$testvar** is equal to. **eval** forces that evaluation.

3.3 Command Substitution

In almost all cases in this document, I use the $(<command>) convention for command substitution: that is,

```
$(date +%H%M)
```

means "substitute the output from the **date +%H%M** command here." This works in Bash 2.0+. In some older versions of Bash, prior to 1.14.7, you may need to use backquotes (`` `date +%H%M` ``). Backquotes can be used in Bash 2.0+, but are being phased out in favor of $(), which nests better. If you're using an earlier version of Bash, you can usually substitute backquotes where you see $(). If the command substitution is escaped (ie. \$(command)), then use backslashes to escape BOTH your backquotes (ie. \'command\').

3.4 Non−Printing Characters in Prompts

Many of the changes that can be made to Bash prompts that are discussed in this HOWTO use non−printing characters. Changing the colour of the prompt text, changing an Xterm title bar, and moving the cursor position all require non−printing characters.

If I want a very simple prompt consisting of a greater−than sign and a space:

```
[giles@nikola giles]$ PS1='> '
>
```

This is just a two character prompt. If I modify it so that it's a bright yellow greater−than sign (colours are discussed in their own section):

```
> PS1='\033[1;33m>\033[0m '
>
```

This works fine − until you type in a large command line. Because the prompt still only consists of two printing characters (a greater−than sign and a space) but the shell thinks that this prompt is eleven characters long (I think it counts '\033' , '[1' and '[0' as one character each). You can see this by typing a really long command line − you will find that the shell wraps the text before it gets to the edge of the terminal, and in most cases wraps it badly. This is because it's confused about the actual length of the prompt.

So use this instead:

```
> PS1='\[\033[1;33m\]>\[\033[0m\] '
```

This is more complex, but it works. Command lines wrap properly. What's been done is to enclose the '\033[1;33m' that starts the yellow colour in '\[' and '\]' which tells the shell "everything between these escaped square brackets, including the brackets themselves, is a non−printing character." The same is done with the '\033[0m' that ends the colour.

3.5 Sourcing a File

When a file is sourced (by typing either **source filename** or **. filename** at the command line), the lines of code in the file are executed as if they were printed at the command line. This is particularly useful with complex prompts, to allow them to be stored in files and called up by sourcing the file they are in.

In examples, you will find that I often include **#!/bin/bash** at the beginning of files including functions. This is **not** necessary if you are sourcing a file, just as it isn't necessary to **chmod +x** a file that is going to be sourced. I do this because it makes Vim (my editor of choice, no flames please − you use what you like) think I'm editing a shell script and turn on colour syntax highlighting.

3.6 Functions, Aliases, and the Environment

As mentioned earlier, PS1, PS2, PS3, PS4, and PROMPT_COMMAND are all stored in the Bash environment. For those of us coming from a DOS background, the idea of tossing big hunks of code into the environment is horrifying, because that DOS environment was small, and didn't exactly grow well. There are probably practical limits to what you can and should put in the environment, but I don't know what they are, and we're probably talking a couple of orders of magnitude larger than what DOS users are used to. As Dan put it:

"In my interactive shell I have 62 aliases and 25 functions. My rule of thumb is that if I need something solely for interactive use and can handily write it in bash I make it a shell function (assuming it can't be easily expressed as an alias). If these people are worried about memory they don't need to be using bash. Bash is one of the largest programs I run on my linux box (outside of Oracle). Run top sometime and press 'M' to sort by memory − see how close bash is to the top of the list. Heck, it's bigger than sendmail! Tell 'em to go get ash or something."

I guess he was using console only the day he tried that: running X and X apps, I have a lot of stuff larger than Bash. But the idea is the same: the environment is something to be used, and don't worry about overfilling it.

I risk censure by Unix gurus when I say this (for the crime of over−simplification), but functions are basically small shell scripts that are loaded into the environment for the purpose of efficiency. Quoting Dan again: "Shell functions are about as efficient as they can be. It is the approximate equivalent of sourcing a bash/bourne shell script save that no file I/O need be done as the function is already in memory. The shell functions are typically loaded from [.bashrc or .bash_profile] depending on whether you want them only in the initial shell or in subshells as well. Contrast this with running a shell script: Your shell forks, the child does an exec, potentially the path is searched, the kernel opens the file and examines enough bytes to determine how to run the file, in the case of a shell script a shell must be started with the name of the script as its argument, the shell then opens the file, reads it and executes the statements. Compared to a shell function, everything other than executing the statements can be considered unnecessary overhead."

Aliases are simple to create:

```
alias d="ls --color=tty --classify"
alias v="d --format=long"
alias rm="rm -i"
```

Any arguments you pass to the alias are passed to the command line of the aliased command (ls in the first two cases). Note that aliases can be nested, and they can be used to make a normal unix command behave in a different way. (I agree with the argument that you shouldn't use the latter kind of aliases − if you get in the habit of relying on "rm *" to ask you if you're sure, you may lose important files on a system that doesn't use your alias.)

Functions are used for more complex program structures. As a general rule, use an alias for anything that can be done in one line. Functions differ from shell scripts in that they are loaded into the environment so that they work more quickly. As a general rule again, you would want to keep functions relatively small, and any shell script that gets relatively large should remain a shell script rather than turning it into a function. Your decision to load something as a function is also going to depend on how often you use it. If you use a small shell script infrequently, leave it as a shell script. If you use it often, turn it into a function.

To modify the behaviour of **ls**, you could do something like the following:

```
function lf
{
    ls --color=tty --classify $*
    echo "$(ls -l $* | wc -l) files"
}
```

This could readily be set as an alias, but for the sake of example, we'll make it a function. If you type the text shown into a text file and then source that file, the function will be in your environment, and be immediately available at the command line without the overhead of a shell script mentioned previously. The usefulness of this becomes more obvious if you consider adding more functionality to the above function, such as using an if statement to execute some special code when links are found in the listing.

4.External Commands

4.1 PROMPT_COMMAND

Bash provides another environment variable called **PROMPT_COMMAND**. The contents of this variable are executed as a regular Bash command just before Bash displays a prompt.

```
[21:55:01][giles@nikola:~] PS1="[\u@\h:\w]\$ "
[giles@nikola:~] PROMPT_COMMAND="date +%H%M"
2155
[giles@nikola:~] d
bin   mail
2156
[giles@nikola:~]
```

What happened above was that I changed PS1 to no longer include the `\t` escape sequence, so the time was no longer a part of the prompt. Then I used `date +%H%M` to display the time in a format I like better. But it appears on a different line than the prompt. Tidying this up using `echo −n ...` as shown below works with Bash 2.0+, but appears not to work with Bash 1.14.7: apparently the prompt is drawn in a different way, and the following method results in overlapping text.

```
2156
[giles@nikola:~] PROMPT_COMMAND="echo −n [$(date +%H%M)]"
[2156][giles@nikola:~]$
[2156][giles@nikola:~]$ d
bin   mail
[2157][giles@nikola:~]$ unset PROMPT_COMMAND
[giles@nikola:~]
```

`echo −n ...` controls the output of the `date` command and supresses the trailing newline, allowing the prompt to appear all on one line. At the end, I used the `unset` command to remove the PROMPT_COMMAND environment variable.

4.2 External Commands in the Prompt

You can use the output of regular Linux commands directly in the prompt as well. Obviously, you don't want to insert a lot of material, or it will create a large prompt. You also want to use a **fast** command, because it's going to be executed every time your prompt appears on the screen, and delays in the appearance of your prompt while you're working can be very annoying. (Unlike the previous example that this closely resembles, this does work with Bash 1.14.7.)

```
[21:58:33][giles@nikola:~]$ PS1="[\$(date +%H%M)][\u@\h:\w]\$ "
[2159][giles@nikola:~]$ ls
bin   mail
[2200][giles@nikola:~]$
```

It's important to notice the backslash before the dollar sign of the command substitution. Without it, the external command is executed exactly once: when the PS1 string is read into the environment. For this prompt, that would mean that it would display the same time no matter how long the prompt was used. The backslash protects the contents of $() from immediate shell interpretation, so "date" is called every time a prompt is generated.

Linux comes with a lot of small utility programs like **date**, **grep**, or **wc** that allow you to manipulate data. If you find yourself trying to create complex combinations of these programs within a prompt, it may be easier to make an alias, function, or shell script of your own, and call it from the prompt. Escape sequences are often required in bash shell scripts to ensure that shell variables are expanded at the correct time (as seen above with the date command): this is raised to another level within the prompt PS1 line, and avoiding it by creating functions is a good idea.

An example of a small shell script used within a prompt is given below:

```
#!/bin/bash
#     lsbytesum - sum the number of bytes in a directory listing
TotalBytes=0
for Bytes in $(ls -l | grep "^-" | cut -c30-41)
do
    let TotalBytes=$TotalBytes+$Bytes
done
TotalMeg=$(echo -e "scale=3 \n$TotalBytes/1048576 \nquit" | bc)
echo -n "$TotalMeg"
```

I have at times kept this both as a function, or as a shell script in my ~/bin directory, which is on my path. Used in a prompt:

```
[2158][giles@nikola:~]$ PS1="[\u@\h:\w (\$(lsbytesum) Mb)]\$ "
[giles@nikola:~ (0 Mb)]$ cd /bin
[giles@nikola:/bin (4.498 Mb)]$
```

4.3 What to Put in Your Prompt

You'll find I put username, machine name, time, and current directory name in most of my prompts. With the exception of the time, these are very standard items to find in a prompt, and time is probably the next most common addition. But what you include is entirely a matter of personal taste. Here are examples from people I know to help give you ideas.

Dan's prompt is minimal but very effective, particularly for the way he works.

```
[giles@nikola:~]$ cur_tty=$(tty | sed −e "s/.*tty\(.*\)/\1/")
[giles@nikola:~]$ echo $cur_tty
p4
[giles@nikola:~]$ PS1="\!,$cur_tty,\$?\$ "
1095,p4,0$
```

Dan doesn't like that having the current working directory can resize the prompt drastically as you move through the directory tree, so he keeps track of that in his head (or types "pwd"). He learned Unix with csh and tcsh, so he uses his command history extensively (something many of us weened on Bash do not do), so the first item in the prompt is the history number. The second item is the significant characters of the tty (the output of "tty" is cropped with sed), an item that can be useful to "screen" users. The third item is the exit value of the last command/pipeline (note that this is rendered useless by any command executed within the prompt − you could work around that by capturing it to a variable and playing it back, though). Finally, the "\$" is a dollar sign for a regular user, and switches to a hash mark ("#") if the user is root.

Torben Fjerdingstad (tfj@fjerdingstad.dk) wrote to tell me that he often suspends jobs and then forgets about them. He uses his prompt to remind himself of suspended jobs:

```
[giles@nikola:~]$ function jobcount {
> jobs|wc −l| awk '{print $1}'
> }
[giles@nikola:~]$ export PS1='\W[`jobcount`]# '
giles[0]# man ls &
[1] 4150

[1]+  Stopped (tty output)    man ls
giles[1]#
```

Torben uses **awk** to trim the whitespace from the output of wc, while I would have used sed or tr − not because they're better, but because I'm more familiar with them. There are probably other ways as well. Torben also surrounds his PS1 string in single quotes, which prevent Bash from immediately interpreting the backquotes, so he doesn't have to escape them as I have mentioned.

NOTE: There is a known bug in Bash 2.02 that causes the **jobs** command (a shell builtin) to return nothing to a pipe. If you try the above under Bash 2.02, you will always get a "0" back regardless of how many jobs you have suspended. This problem is fixed in 2.03.

5.Xterm Title Bar Manipulations

Non−printing escape sequences can be used to produce interesting effects in prompts. To use these escape sequences, you need to enclose them in **\[** and **\]** (as discussed in Non−Printing Characters in Prompts), telling Bash to ignore this material while calculating the size of the prompt. Failing to include these delimiters results in line editing code placing the cursor incorrectly because it doesn't know the actual size of the prompt. Escape sequences must also be preceded by **\033[** in Bash prior to version 2, or by either **\033[** or **\e[** in later versions.

If you try to change the title bar of your Xterm with your prompt when you're at the console, you'll produce garbage in your prompt. To avoid this, test the TERM environment variable to tell if your prompt is going to be in an Xterm.

```
function proml
{
case $TERM in
    xterm*)
        local TITLEBAR='\[\033]0;\u@\h:\w\007\]'
        ;;
    *)
        local TITLEBAR=''
        ;;
esac

PS1="${TITLEBAR}\
[\$(date +%H%M)]\
[\u@\h:\w]\
\$ "
PS2='> '
PS4='+ '
}
```

This is a function that can be incorporated into ~/.bashrc. The function name could then be called to execute the function. The function, like the PS1 string, is stored in the environment. Once the PS1 string is set by the function, you can remove the function from the environment with `unset proml`. Since the prompt can't change from being in an Xterm to being at the console, the TERM variable isn't tested every time the prompt is generated. I used continuation markers (backslashes) in the definition of the prompt, to allow it to be continued on multiple lines. This improves readability, making it easier to modify and debug.

I define this as a function because this is how the Bashprompt package (discussed later in this document: The Bash Prompt Package) deals with prompts: it's not the only way to do it, but it works well. As the prompts you use become more complex, it becomes more and more cumbersome to type them in at the prompt, and more practical to make them into some sort of text file. In this case, to test this at the prompt, save the above as a text file called "proml". You can work with it as follows:

```
[giles@nikola:/bin (4.498 Mb)]$ cd          -> Go where you want to save the prompt
[giles@nikola:~ (0 Mb)]$ vi proml           -> Edit the prompt file
...                                         -> Enter the text given above
[giles@nikola:~ (0 Mb)]$ source proml       -> Read the prompt function
[giles@nikola:~ (0 Mb)]$ proml              -> Execute the prompt function
```

The first step in creating this prompt is to test if the shell we're starting is an xterm or not: if it is, the shell variable (${TITLEBAR}) is defined. It consists of the appropriate escape sequences, and **\u@\h:\w**, which puts <user>@<machine>:<working directory> in the Xterm title bar. This is particularly useful with minimized Xterms, making them more rapidly identifiable. The other material in this prompt should be familiar from previous prompts we've created.

The only drawback to manipulating the Xterm title bar like this occurs when you log into a system on which you haven't set up the title bar hack: the Xterm will continue to show the information from the previous system that had the title bar hack in place.

6.ANSI Escape Sequences: Colours and Cursor Movement

6.1 Colours

As mentioned before, non−printing escape sequences have to be enclosed in **\[\033[** and **\]**. For colour escape sequences, they should also be followed by a lowercase **m**.

If you try out the following prompts in an xterm and find that you aren't seeing the colours named, check out your ~/.Xdefaults file (and possibly its bretheren) for lines like "XTerm*Foreground: BlanchedAlmond". This can be commented out by placing an exclamation mark ("!") in front of it. Of course, this will also be dependent on what terminal emulator you're using. This is the likeliest place that your term foreground colours would be overridden.

To include blue text in the prompt:

```
PS1="\[\033[34m\][\$(date +%H%M)][\u@\h:\w]$ "
```

The problem with this prompt is that the blue colour that starts with the 34 colour code is never switched back to the regular colour, so any text you type after the prompt is still in the colour of the prompt. This is also a dark shade of blue, so combining it with the **bold** code might help:

```
PS1="\[\033[1;34m\][\$(date +%H%M)][\u@\h:\w]$\[\033[0m\] "
```

The prompt is now in light blue, and it ends by switching the colour back to nothing (whatever foreground colour you had previously).

Here are the rest of the colour equivalences:

Black	0;30	Dark Gray	1;30
Blue	0;34	Light Blue	1;34
Green	0;32	Light Green	1;32
Cyan	0;36	Light Cyan	1;36
Red	0;31	Light Red	1;31
Purple	0;35	Light Purple	1;35
Brown	0;33	Yellow	1;33
Light Gray	0;37	White	1;37

Daniel Dui (ddui@iee.org) points out that to be strictly accurate, we must mention that the list above is for colours at the console. In an xterm, the code **1;31** isn't "Light Red," but "Bold Red." This is true of all the colours.

You can also set background colours by using 44 for Blue background, 41 for a Red background, etc. There are no bold background colours. Combinations can be used, like Light Red text on a Blue background: **\[\033[44;1;31m\]**, although setting the colours separately seems to work better (ie. **\[\033[44m\]\[\033[1;31m\]**). Other codes available include 4: Underscore, 5: Blink, 7: Inverse, and 8: Concealed.

Aside: Many people (myself included) object strongly to the "blink" attribute. Fortunately, it doesn't work in any terminal emulators that I'm aware of − but it will still work on the console. And, if you were wondering (as I did) "What use is a 'Concealed' attribute?!" − I saw it used in an example shell script (not a prompt) to allow someone to type in a password without it being echoed to the screen.

Based on a prompt called "elite2" in the Bashprompt package (which I have modified to work better on a standard console, rather than with the special xterm fonts required to view the original properly), this is a prompt I've used a lot:

```
function elite
{

local GRAY="\[\033[1;30m\]"
local LIGHT_GRAY="\[\033[0;37m\]"
local CYAN="\[\033[0;36m\]"
local LIGHT_CYAN="\[\033[1;36m\]"

case $TERM in
    xterm*)
        local TITLEBAR='\[\033]0;\u@\h:\w\007\]'
        ;;
    *)
        local TITLEBAR=""
        ;;
esac

local GRAD1=$(tty|cut -d/ -f3)
PS1="$TITLEBAR\
$GRAY-$CYAN-$LIGHT_CYAN(\
$CYAN\u$GRAY@$CYAN\h\
$LIGHT_CYAN)$CYAN-$LIGHT_CYAN(\
$CYAN\#$GRAY/$CYAN$GRAD1\
$LIGHT_CYAN)$CYAN-$LIGHT_CYAN(\
$CYAN\$(date +%H%M)$GRAY/$CYAN\$(date +%d-%b-%y)\
$LIGHT_CYAN)$CYAN-$GRAY-\
$LIGHT_GRAY\n\
$GRAY-$CYAN-$LIGHT_CYAN(\
$CYAN\$$GRAY:$CYAN\w\
$LIGHT_CYAN)$CYAN-$GRAY-$LIGHT_GRAY "
PS2="$LIGHT_CYAN-$CYAN-$GRAY-$LIGHT_GRAY "
}
```

I define the colours as temporary shell variables in the name of readability. It's easier to work with. The "GRAD1" variable is a check to determine what terminal you're on. Like the test to determine if you're working in an Xterm, it only needs to be done once. The prompt you see look like this, except in colour:

```
       --(giles@nikola)-(75/ttyp7)-(1908/12-Oct-98)--
       --($:~/tmp)--
```

To help myself remember what colours are available, I wrote the following script which echoes all the colours to screen:

```
#!/bin/bash
#
#   This file echoes a bunch of colour codes to the terminal to demonstrate
#   what's available.  Each line is one colour on black and gray
#   backgrounds, with the code in the middle.  Verified to work on white,
#   black, and green BGs (2 Dec 98).
#
```

```
echo "  On Light Gray:        On Black:"
echo −e "\033[47m\033[1;37m  White        \033[0m\
 1;37m \
\033[40m\033[1;37m  White        \033[0m"
echo −e "\033[47m\033[37m  Light Gray   \033[0m\
   37m \
\033[40m\033[37m  Light Gray   \033[0m"
echo −e "\033[47m\033[1;30m  Gray         \033[0m\
 1;30m \
\033[40m\033[1;30m  Gray         \033[0m"
echo −e "\033[47m\033[30m  Black        \033[0m\
   30m \
\033[40m\033[30m  Black        \033[0m"
echo −e "\033[47m\033[31m  Red          \033[0m\
   31m \
\033[40m\033[31m  Red          \033[0m"
echo −e "\033[47m\033[1;31m  Light Red    \033[0m\
 1;31m \
\033[40m\033[1;31m  Light Red    \033[0m"
echo −e "\033[47m\033[32m  Green        \033[0m\
   32m \
\033[40m\033[32m  Green        \033[0m"
echo −e "\033[47m\033[1;32m  Light Green  \033[0m\
 1;32m \
\033[40m\033[1;32m  Light Green  \033[0m"
echo −e "\033[47m\033[33m  Brown        \033[0m\
   33m \
\033[40m\033[33m  Brown        \033[0m"
echo −e "\033[47m\033[1;33m  Yellow       \033[0m\
 1;33m \
\033[40m\033[1;33m  Yellow       \033[0m"
echo −e "\033[47m\033[34m  Blue         \033[0m\
   34m \
\033[40m\033[34m  Blue         \033[0m"
echo −e "\033[47m\033[1;34m  Light Blue   \033[0m\
 1;34m \
\033[40m\033[1;34m  Light Blue   \033[0m"
echo −e "\033[47m\033[35m  Purple       \033[0m\
   35m \
\033[40m\033[35m  Purple       \033[0m"
echo −e "\033[47m\033[1;35m  Pink         \033[0m\
 1;35m \
\033[40m\033[1;35m  Pink         \033[0m"
echo −e "\033[47m\033[36m  Cyan         \033[0m\
   36m \
\033[40m\033[36m  Cyan         \033[0m"
echo −e "\033[47m\033[1;36m  Light Cyan   \033[0m\
 1;36m \
\033[40m\033[1;36m  Light Cyan   \033[0m"
```

6.2 Cursor Movement

ANSI escape sequences allow you to move the cursor around the screen at will. This is more useful for full screen user interfaces generated by shell scripts, but can also be used in prompts. The movement escape sequences are as follows:

```
        - Position the Cursor:
          \033[<L>;<C>H
             Or
          \033[<L>;<C>f
          puts the cursor at line L and column C.
        - Move the cursor up N lines:
          \033[<N>A
        - Move the cursor down N lines:
          \033[<N>B
        - Move the cursor forward N columns:
          \033[<N>C
        - Move the cursor backward N columns:
          \033[<N>D

        - Clear the screen, move to (0,0):
          \033[2J
        - Erase to end of line:
          \033[K

        - Save cursor position:
          \033[s
        - Restore cursor position:
          \033[u
```

The latter two codes are NOT honoured by many terminal emulators. The only ones that I'm aware of that do are xterm and nxterm − even though the majority of terminal emulators are based on xterm code. As far as I can tell, rxvt, kvt, xiterm, and Eterm do not support them. They are supported on the console.

Try putting in the following line of code at the prompt (it's a little clearer what it does if the prompt is several lines down the terminal when you put this in): `echo -en "\033[7A\033[1;35m BASH \033[7B\033[6D"` This should move the cursor seven lines up screen, print the word " BASH ", and then return to where it started to produce a normal prompt. This isn't a prompt: it's just a demonstration of moving the cursor on screen, using colour to emphasize what has been done.

Save this in a file called "clock":

```
#!/bin/bash

function prompt_command {
let prompt_x=$COLUMNS-5
}

PROMPT_COMMAND=prompt_command

function clock {
local       BLUE="\[\033[0;34m\]"
local        RED="\[\033[0;31m\]"
local  LIGHT_RED="\[\033[1;31m\]"
local      WHITE="\[\033[1;37m\]"
local  NO_COLOUR="\[\033[0m\]"
case $TERM in
    xterm*)
        TITLEBAR='\[\033]0;\u@\h:\w\007\]'
        ;;
    *)
        TITLEBAR=""
        ;;
esac

PS1="${TITLEBAR}\
\[\033[s\033[1;\$(echo -n \${prompt_x})H\]\
$BLUE[$LIGHT_RED\$(date +%H%M)$BLUE]\[\033[u\033[1A\]
$BLUE[$LIGHT_RED\u@\h:\w$BLUE]\
$WHITE\$$NO_COLOUR "
PS2='> '
PS4='+ '
}
```

This prompt is fairly plain, except that it keeps a 24 hour clock in the upper right corner of the terminal (even if the terminal is resized). This will NOT work on the terminal emulators that I mentioned that don't accept the save and restore cursor position codes. If you try to run this prompt in any of those terminal emulators, the clock will appear correctly, but the prompt will be trapped on the second line of the terminal.

See also The Elegant Useless Clock Prompt for a more extensive use of these codes.

6.3 Moving the Cursor With tput

As with so many things in Unix, there is more than one way to achieve the same ends. A utility called "tput" can also be used to move the cursor around the screen, or get back information about the status of the terminal. "tput" for cursor positioning is less flexible than ANSI escape sequences: you can only move the cursor to an absolute position, you can't move it relative to its current position. I don't use "tput," so I'm not going to explain it in detail. Type "man tput" and you'll know as much as I do.

7.Special Characters: Octal Escape Sequences

Outside of the characters that you can type on your keyboard, there are a lot of other characters you can print on your screen. I've created a script to allow you to check out what the font you're using has available for you. The main command you need to use to utilise these characters is "echo −e". The "−e" switch tells echo to enable interpretation of backslash−escaped characters. What you see when you look at octal 200−400 will be very different with a VGA font from what you will see with a standard Linux font. Be warned that some of these escape sequences have odd effects on your terminal, and I haven't tried to prevent them from doing whatever they do. The linedraw and block characters (which many of us became familiar with with Word Perfect) that are used heavily by the Bashprompt project are between octal 260 and 337.

```
#!/bin/bash
```

```
#   Script: escgen

function usage {
   echo -e "\033[1;34mescgen\033[0m <lower_octal_value> [<higher_octal_value>]"
   echo "   Octal escape sequence generator: print all octal escape sequences"
   echo "   between the lower value and the upper value.  If a second value"
   echo "   isn't supplied, print eight characters."
   echo "   1998 - Giles Orr, no warranty."
   exit 1
}

if [ "$#" -eq "0" ]
then
   echo -e "\033[1;31mPlease supply one or two values.\033[0m"
   usage
fi
let lower_val=${1}
if [ "$#" -eq "1" ]
then
   #   If they don't supply a closing value, give them eight characters.
   upper_val=$(echo -e "obase=8 \n ibase=8 \n $lower_val+10 \n quit" | bc)
else
   let upper_val=${2}
fi
if [ "$#" -gt "2" ]
then
   echo -e "\033[1;31mPlease supply two values.\033[0m"
   echo
   usage
fi
if [ "${lower_val}" -gt "${upper_val}" ]
then
   echo -e "\033[1;31m${lower_val} is larger than ${upper_val}."
   echo
   usage
fi
if [ "${upper_val}" -gt "777" ]
   then
   echo -e "\033[1;31mValues cannot exceed 777.\033[0m"
   echo
   usage
fi

let i=$lower_val
let line_count=1
let limit=$upper_val
while [ "$i" -lt "$limit" ]
do
   octal_escape="\\$i"
   echo -en "$i:'$octal_escape' "
   if [ "$line_count" -gt "7" ]
   then
      echo
      #   Put a hard return in.
      let line_count=0
   fi
   let i=$(echo -e "obase=8 \n ibase=8 \n $i+1 \n quit" | bc)
   let line_count=$line_count+1
done
echo
```

You can also use **xfd** to display all the characters in an X font, with the command "xfd −fn <fontname>". Clicking on any given character will give you lots of information about that character, including its octal value. The script given above will be useful on the console, and if you aren't sure of the current font name.

8.The Bash Prompt Package

8.1 Availability

The Bash Prompt package is available at http://bash.current.nu, and is the work of several people, co−ordinated by Rob Current (aka BadLandZ). The package is an early beta, but offers a simple way of using multiple prompts (or themes), allowing you to set prompts for login shells, and for subshells (ie. putting PS1 strings in ~/.bash_profile and ~/.bashrc). Most of the themes use the extended VGA character set, so they look bad unless they're used with VGA fonts (which aren't the default on most systems).

8.2 Xterm Fonts

To use some of the most attractive prompts in the Bash Prompt package, you need to get and install fonts that support the character sets expected by the prompts. These are "VGA Fonts," which support different character sets than regular Xterm fonts. Standard Xterm fonts support an extended alphabet, including a lot of letters with accents. In VGA fonts, this material is replaced by graphical characters − blocks, dots, lines. I asked for an explanation of this difference, and Sérgio Vale e Pace (space@gold.com.br) wrote me:

> I love computer history so here goes:
>
> When IBM designed the first PC they needed some character codes to use, so they got the ASCII character table (128 numbers, letters, and some punctuation) and to fill a byte addressed table they added 128 more characters. Since the PC was designed to be a home computer, they fill the remaining 128 characters with dots, lines, points, etc, to be able to do borders, and grayscale effects (remember that we are talking about 2 color graphics).
>
> Time passes, PCs become a standard, IBM creates more powerful systems and the VGA standard is born, along with 256 colour graphics, and IBM continues to include their IBM−ASCII characters table.
>
> More time passes, IBM has lost their leadership in the PC market, and the OS authors dicover that there are other languages in the world that use non−english characters, so they

add international alphabet support in their systems. Since we now have bright and colorful screens, we can trash the dots, lines, etc. and use their space for accented characters and some greek letters, which you'll see in Linux.

8.3 Changing the Xterm Font

Getting and installing these fonts is a somewhat involved process. First, retrieve the font(s). Next, ensure they're .pcf or .pcf.gz files. If they're .bdf files, investigate the "bdftopcf" command (ie. read the man page). Drop the .pcf or .pcf.gz files into the /usr/X11R6/lib/X11/fonts/misc dir (this is the correct directory for RedHat 5.1 and Slackware 3.4, it may be different on other distributions). "cd" to that directory, and run the "mkfontdir" command. Then run "xset fp rehash". Sometimes it's a good idea to go into the fonts.alias file in the same directory, and create shorter alias names for the fonts.

To use the new fonts, you start your Xterm program of choice with the appropriate command to your Xterm, which can be found either in the man page or by using the "−−help" parameter on the command line. Popular terms would be used as follows:

```
xterm −font <fontname>
```

OR

```
xterm −fn <fontname> −fb <fontname−bold>
Eterm −F <fontname>
rxvt −fn <fontname>
```

VGA fonts are available from *Stumpy's ANSI Fonts* page at http://home.earthlink.net/~us5zahns/enl/ansifont.html (which I have borrowed from extensively while writing this).

9.Loading a Different Prompt

9.1 Loading a Different Prompt, Later

The explanations in this HOWTO have shown how to make PS1 environment variables, or how to incorporate those PS1 and PS2 strings into functions that could be called by ~/.bashrc or as a theme by the bashprompt package.

Using the bashprompt package, you would type `bashprompt -i` to see a list of available themes. To set the prompt in future login shells (primarily the console, but also telnet and Xterms, depending on how your Xterms are set up), you would type `bashprompt -l themename`. bashprompt then modifies your ~/.bash_profile to call the requested theme when it starts. To set the prompt in future subshells (usually Xterms, rxvt, etc.), you type `bashprompt -s themename`, and bashprompt modifies your ~/.bashrc file to call the appropriate theme at startup.

See also Setting the PS? Strings Permanently for Johan Kullstam's note regarding the importance of putting the PS? strings in ~/.bashrc .

9.2 Loading a Different Prompt, Immediately

You can change the prompt in your current terminal (using the example "elite" function above) by typing "`source elite`" followed by "`elite`" (assuming that the elite function file is the working directory). This is somewhat cumbersome, and leaves you with an extra function (elite) in your environment space − if you want to clean up the environment, you would have to type "`unset elite`" as well. This would seem like an ideal candidate for a small shell script, but a script doesn't work here because the script cannot change the environment of your current shell: it can only change the environment of the subshell it runs in. As soon as the script stops, the subshell goes away, and the changes the script made to the environment are gone. What **can** change environment variables of your current shell are environment functions. The bashprompt package puts a function called "callbashprompt" into your environment, and, while they don't document it, it can be called to load any bashprompt theme on the fly. It looks in the theme directory it installed (the theme you're calling has to be there), sources the function you asked for, loads the function, and then unsets the function, thus keeping your environment uncluttered. "callbashprompt" wasn't intended to be used this way, and has no error checking, but if you keep that in mind, it works quite well.

10.Loading Prompt Colours Dynamically

10.1 A "Proof of Concept" Example

This is a "proof of concept" more than an attractive prompt: changing colours within the prompt dynamically. In this example, the colour of the host name changes depending on the load (as a warning).

```
#!/bin/bash
#   "hostloadcolour" − 17 October 98, by Giles
#
#   The idea here is to change the colour of the host name in the prompt,
#   depending on a threshold load value.

# THRESHOLD_LOAD is the value of the one minute load (multiplied
# by one hundred) at which you want
# the prompt to change from COLOUR_LOW to COLOUR_HIGH
THRESHOLD_LOAD=200
COLOUR_LOW='1;34'
          # light blue
COLOUR_HIGH='1;31'
           # light red

function prompt_command {
ONE=$(uptime | sed −e "s/.*load average: \(.*\...\), \(.*\...\), \(.*\...\)/\1/" −e "s/ //g")
#   Apparently, "scale" in bc doesn't apply to multiplication, but does
#   apply to division.
ONEHUNDRED=$(echo −e "scale=0 \n $ONE/0.01 \nquit \n" | bc)
if [ $ONEHUNDRED −gt $THRESHOLD_LOAD ]
then
    HOST_COLOUR=$COLOUR_HIGH
        # Light Red
else
    HOST_COLOUR=$COLOUR_LOW
        # Light Blue
fi
}

function hostloadcolour {

PROMPT_COMMAND=prompt_command
PS1="[$(date +%H%M)][\u@\[\033[\$(echo −n \$HOST_COLOUR)m\]\h\[\033[0m\]:\w]$ "
}
```

Using your favorite editor, save this to a file named "hostloadcolour". If you have the Bashprompt package installed, this will work as a theme. If you don't, type `source hostloadcolour` and then `hostloadcolour`. Either way, "prompt_command" becomes a function in your environment. If you examine the code, you will notice that the colours ($COLOUR_HIGH and $COLOUR_LOW) are set using only a partial colour code, ie. "1;34" instead of "\[\033[1;34m\]", which I would have preferred. I have been unable to get it to work with the complete code. Please let me know if you manage this.

11.Prompt Code Snippets

This section shows how to put various pieces of information into the Bash prompt. There are an infinite number of things that could be put in your prompt. Feel free to send me examples, I will try to include what I think will be most widely used. If you have an alternate way to retrieve a piece of information here, and feel your method is more efficient, please contact me. It's easy to write bad code, I do it often, but it's great to write elegant code, and a pleasure to read it. I manage it every once in a while, and would love to have more of it to put in here.

To incorporate shell code in prompts, it has to be escaped. Usually, this will mean putting it inside `\$(<command>)` so that the output of `command` is substituted each time the prompt is generated.

11.1 Built−in Escape Sequences

See Bash Prompt Escape Sequences for a complete list of built−in escape sequences. This list is taken directly from the Bash man page, so you can also look there.

11.2 Date and Time

If you don't like the built−ins for date and time, extracting the same information from the `date` command is relatively easy. Examples already seen in this HOWTO include `date +%H%M`, which will put in the hour in 24 hour format, and the minute. `date "+%A, %d %B %Y"` will give something like "Sunday, 06 June 1999". For a full list of the interpreted sequences, type `date −−help` or `man date`.

11.3 Counting Files in the Current Directory

To determine how many files there are in the current directory, put in `ls −l | wc −l`. This uses `wc` wordcount to do a count of the number of lines (−l) in the output of `ls −l`. It doesn't count dotfiles. If you want to count only files and NOT include symbolic links (just an example of what else you could do), you could use `ls −l | grep −v ^l | wc −l`. Here, `grep` checks for any line beginning with "l" (indicating a link), and discards that line (−v).

11.4 Total Bytes in the Current Directory

If you want to know how much space the contents of the current directory take up, you can use something like the following:

```
# The sed command replaces all the spaces with only one space.
# cut −d" " −f5 : −d determines a delimiter, which means that (in
# this case) a space begins a new column.
# −f says to take out a certain column, in this case the fifth one

let TotalBytes=0

for Bytes in $(ls −l | grep "^−" | sed −e "s/ \+/ /g" | cut −d" " −f5)
do
   let TotalBytes=$TotalBytes+$Bytes
done

# The if...fi's give a more specific output in byte, kilobyte, megabyte,
# and gigabyte

if [ $TotalBytes −lt 1024 ]; then
   TotalSize=$(echo −e "scale=3 \n$TotalBytes \nquit" | bc)
else if [ $TotalBytes −lt 1048576 ]; then
   TotalSize=$(echo −e "scale=3 \n$TotalBytes/1024 \nquit" | bc)
else if [ $TotalBytes −lt 1073741824 ]; then
   TotalSize=$(echo −e "scale=3 \n$TotalBytes/1048576 \nquit" | bc)
else
   TotalSize=$(echo −e "scale=3 \n$TotalBytes/1073741824 \nquit" | bc)
fi
fi
fi
```

Code courtesy of Sam Schmit (id@pt.lu) and his uncle Jean−Paul, who ironed out a fairly major bug in my original code, and just generally cleaned it up.

11.5 Checking the Current TTY

The `tty` command returns the filename of the terminal connected to standard input. This comes in two formats on the Linux systems I have used, either "/dev/tty4" or "/dev/pts/2". I have taken to using a more general solution to this: `tty | sed −e "s:/dev/::"`, which removes the leading "/dev/". Older systems (in my experience, RedHat through 5.2) returned only filenames in the "/dev/tty4" format, so I used `tty | sed −e "s/.*tty\(.*\)/\1/"`.

An alternative method: `ps aux | grep $$ | awk '{ print $7 }'`.

11.6 Suspended Job Count

To find out how many suspended jobs you have, use `jobs | wc −l | awk '{print $1}'`. `awk` is used to trim the output, which would otherwise include blank spaces that waste space in a prompt. If you start netscape from an xterm, this will also be counted. If you want to avoid that, and only count stopped jobs, use `jobs −s` instead. Type `help jobs` for more info on jobs. `jobs` will always return nothing to a pipe in version 2.02 of Bash: this problem is not present in any other version.

11.7 Uptime and Load

Current load is taken from the `uptime` command. What I use at the moment is `uptime | sed -e "s/.*load average: \(.*\...\), .*\..., .*\.../\1/" -e "s/ //g"` which is clunky in the extreme, but works. Replacements welcome. `uptime` can also be used in a very similar manner to find out how long the machine has been up (obviously) or how many users are logged in, and the data could be massaged with `sed` to look the way you want it to.

11.8 Number of Processes

`ps ax | wc -l | tr -d " "`*OR*`ps ax | wc -l | awk '{print $1}'`*OR*`ps ax | wc -l | sed -e "s: ::g"`. In each case, `tr` or `awk` or `sed` is used to remove the undesirable whitespace.

11.9 Controlling the Width of $PWD

Unix allows long file names, which can lead to the value of $PWD being very long. Some people (notably the default RedHat prompt) choose to use the basename of the current working directory (ie. "giles" if $PWD="/home/giles"). I like more info than that, but it's often desirable to limit the length of the directory name, and it makes the most sense to truncate on the left.

```
#   How many characters of the $PWD should be kept
local pwd_length=30
if [ $(echo -n $PWD | wc -c | tr -d " ") -gt $pwd_length ]
then
   newPWD="...$(echo -n $PWD | sed -e "s/.*\(.\{$pwd_length\}\)/\1/")"
else
   newPWD="$(echo -n $PWD)"
fi
```

The above code can be executed as part of PROMPT_COMMAND, and the environment variable generated (*newPWD*) can then be included in the prompt.

11.10 Laptop Power

Again, this isn't elegant, but it works (most of the time). If you have a laptop with APM installed, try `power=$(apm | sed -e "s/.*: \([1-9][0-9]*\)%/\1/" | tr -d " ")` executed from PROMPT_COMMAND to create an environment variable you can add to your prompt. This will indicate percentage power remaining.

11.11 Having the Prompt Ignored on Cut and Paste

This one is weird but cool. Rory Toma (rory@corp.webtv.net) wrote to suggest a prompt like this: `: rory@demon ;` . How is this useful? Well, if you type a command after the prompt (odd idea, that), you can triple click on that line (in Linux, anyway) to highlight the whole line, then paste that line in front of another prompt, and the stuff between the ":" and the """ is ignored, like so:

```
: rory@demon ; uptime
  5:15pm  up 6 days, 23:04,  2 users,  load average: 0.00, 0.00, 0.00
: rory@demon ; : rory@demon ; uptime
  5:15pm  up 6 days, 23:04,  2 users,  load average: 0.00, 0.00, 0.00
```

The prompt is a no−op, and if your PS2 is set to a space, multiple lines can be cut and pasted as well.

11.12 Setting the Window Title and Icon Title Separately

A suggestion from Charles Lepple (clepple@negativezero.org) on setting the window title of the Xterm and the title of the corresponding icon separately (first check out the earlier section Xterm Title Bar Manipulations). He uses this under WindowMaker because the title that's appropriate for an Xterm is usually too long for a 64x64 icon. "\[\e]1;icon−title\007\e]2;main−title\007\]". He says to set this in the prompt command because "I tried putting the string in PS1, but it causes flickering under some window managers because it results in setting the prompt multiple times when you are editing a multi−line command (at least under bash 1.4.x −− and I was too lazy to fully explore the reasons behind it)." I had no trouble with it in the PS1 string, but didn't use any multi−line commands. He also points out that it works under xterm, xwsh, and dtterm, but not gnome−terminal (which uses only the main title). I also found it to work with rxvt, but not kterm.

12.Example Prompts

http://www.interlog.com/~giles/bashprompt/prompts. Web pages also allow me to include pictures, which I can't include in a standard HOWTO. All of the examples given here except Bradley Alexander's "Prompts Depending on Connection Types" can also be seen on the web.

12.2 A "Lightweight" Prompt

```
function proml {
local BLUE="\[\033[0;34m\]"
local RED="\[\033[0;31m\]"
local LIGHT_RED="\[\033[1;31m\]"
local WHITE="\[\033[1;37m\]"
local NO_COLOUR="\[\033[0m\]"
case $TERM in
    xterm*)
        TITLEBAR='\[\033]0;\u@\h:\w\007\]'
        ;;
    *)
        TITLEBAR=""
        ;;
esac

PS1="${TITLEBAR}\
$BLUE[$RED\$(date +%H%M)$BLUE]\
$BLUE[$LIGHT_RED\u@\h:\w$BLUE]\
$WHITE\$$NO_COLOUR "
PS2='> '
PS4='+ '
}
```

12.3 Elite from Bashprompt Themes

Note that this requires a VGA font.

```
# Created by KrON from windowmaker on IRC
# Changed by Spidey 08/06
function elite {
PS1="\[\033[31m\]\332\304\[\033[34m\](\[\033[31m\]\u\[\033[34m\]@\[\033[31m\]\h\
\[\033[34m\])\[\033[31m\]-\[\033[34m\](\[\033[31m\]\$(date +%I:%M%P)\
\[\033[34m\]-:-\[\033[31m\]\$(date +%m)\[\033[34m\033[31m\]/\$(date +%d)\
\[\033[34m\])\[\033[31m\]\304-\[\033[34m]\\371\[\033[31m\]-\371\371\
\[\033[34m\]\372\n\[\033[31m\]\300\304\[\033[34m\](\[\033[31m\]\W\[\033[34m\])\
\[\033[31m\]\304\371\[\033[34m\]\372\[\033[00m\]"
PS2="> "
}
```

12.4 A "Power User" Prompt

I actually do use this prompt, but it results in noticeable delays in the appearance of the prompt on a single−user PII−400, so I wouldn't recommend using it on a multi−user P−100 or anything ... Look at it for ideas, rather than as a practical prompt.

```
#!/bin/bash
#--------------------------------------------------------------------
#       POWER USER PROMPT "pprom2"
#--------------------------------------------------------------------
#
#   Created August 98, Last Modified 9 November 98 by Giles
#
#   Problem: when load is going down, it says "1.35down-.08", get rid
#   of the negative

function prompt_command
{
#   Create TotalMeg variable: sum of visible file sizes in current directory
local TotalBytes=0
for Bytes in $(ls -l | grep "^-" | cut -c30-41)
do
    let TotalBytes=$TotalBytes+$Bytes
done
TotalMeg=$(echo -e "scale=3 \nx=$TotalBytes/1048576\n if (x<1) {print \"0\"} \n print x \nquit" |

#      This is used to calculate the differential in load values
#      provided by the "uptime" command.  "uptime" gives load
#      averages at 1, 5, and 15 minute marks.
#
local one=$(uptime | sed -e "s/.*load average: \(.*\...\), \(.*\...\), \(.*\...\)/\1/" -e "s/ //g
local five=$(uptime | sed -e "s/.*load average: \(.*\...\), \(.*\...\), \(.*\...\).*/\2/" -e "s/
local diff1_5=$(echo -e "scale = scale ($one) \nx=$one - $five\n if (x>0) {print \"up\"} else {pr
loaddiff="$(echo -n "${one}${diff1_5}")"

#   Count visible files:
let files=$(ls -l | grep "^-" | wc -l | tr -d " ")
let hiddenfiles=$(ls -l -d .* | grep "^-" | wc -l | tr -d " ")
let executables=$(ls -l | grep ^-..x | wc -l | tr -d " ")
let directories=$(ls -l | grep "^d" | wc -l | tr -d " ")
let hiddendirectories=$(ls -l -d .* | grep "^d" | wc -l | tr -d " ")-2
let linktemp=$(ls -l | grep "^l" | wc -l | tr -d " ")
if [ "$linktemp" -eq "0" ]
then
    links=""
else
    links=" ${linktemp}l"
fi
unset linktemp
let devicetemp=$(ls -l | grep "^[bc]" | wc -l | tr -d " ")
if [ "$devicetemp" -eq "0" ]
then
    devices=""
else
    devices=" ${devicetemp}bc"
fi
unset devicetemp

}

PROMPT_COMMAND=prompt_command

function pprom2 {

local        BLUE="\[\033[0;34m\]"
local  LIGHT_GRAY="\[\033[0;37m\]"
local LIGHT_GREEN="\[\033[1;32m\]"
local  LIGHT_BLUE="\[\033[1;34m\]"
local  LIGHT_CYAN="\[\033[1;36m\]"
local      YELLOW="\[\033[1;33m\]"
local       WHITE="\[\033[1;37m\]"
local         RED="\[\033[0;31m\]"
local   NO_COLOUR="\[\033[0m\]"

case $TERM in
    xterm*)
        TITLEBAR='\[\033]0;\u@\h:\w\007\]'
        ;;
    *)
        TITLEBAR=""
        ;;
esac

PS1="$TITLEBAR\
$BLUE[$RED\$(date +%H%M)$BLUE]\
$BLUE[$RED\u@\h$BLUE]\
$BLUE[\
$LIGHT_GRAY\${files}.\${hiddenfiles}−\
$LIGHT_GREEN\${executables}x \
$LIGHT_GRAY(\${TotalMeg}Mb) \
$LIGHT_BLUE\${directories}.\
\${hiddendirectories}d\
$LIGHT_CYAN\${links}\
$YELLOW\${devices}\
$BLUE]\
$BLUE[${WHITE}\${loaddiff}$BLUE]\
$BLUE[\
$WHITE\$(ps ax | wc −l | sed −e \"s: ::g\")proc\
$BLUE]\
\n\
$BLUE[$RED\$PWD$BLUE]\
$WHITE\$\
\
$NO_COLOUR "
PS2='> '
PS4='+ '
}
```

12.5 Prompt Depending on Connection Type

Bradley M Alexander (storm@tux.org) had the excellent idea of reminding his users what kind of connection they were using to his machine(s), so he colour−codes prompts dependent on connection type. Here's the bashrc he supplied to me:

```
# /etc/bashrc

# System wide functions and aliases
# Environment stuff goes in /etc/profile

# For some unknown reason bash refuses to inherit
# PS1 in some circumstances that I can't figure out.
# Putting PS1 here ensures that it gets loaded every time.

# Set up prompts. Color code them for logins. Red for root, white for
# user logins, green for ssh sessions, cyan for telnet,
# magenta with red "(ssh)" for ssh + su, magenta for telnet.
THIS_TTY=tty`ps aux | grep $$ | grep bash | awk '{ print $7 }'`
SESS_SRC=`who | grep $THIS_TTY | awk '{ print $6 }'`

SSH_FLAG=0
SSH_IP=`echo $SSH_CLIENT | awk '{ print $1 }'`
if [ $SSH_IP ] ; then
  SSH_FLAG=1
fi
SSH2_IP=`echo $SSH2_CLIENT | awk '{ print $1 }'`
if [ $SSH2_IP ] ; then
  SSH_FLAG=1
fi
if [ $SSH_FLAG −eq 1 ] ; then
  CONN=ssh
elif [ −z $SESS_SRC ] ; then
  CONN=lcl
elif [ $SESS_SRC = "(:0.0)" −o $SESS_SRC = "" ] ; then
  CONN=lcl
else
  CONN=tel
fi

# Okay...Now who we be?
if [ `/usr/bin/whoami` = "root" ] ; then
  USR=priv
else
  USR=nopriv
fi

#Set some prompts...
if [ $CONN = lcl −a $USR = nopriv ] ; then
  PS1="[\u \W]\\$ "
elif [ $CONN = lcl −a $USR = priv ] ; then
  PS1="\[\033[01;31m\][\w]\\$\[\033[00m\] "
elif [ $CONN = tel −a $USR = nopriv ] ; then
  PS1="\[\033[01;34m\][\u@\h \W]\\$\[\033[00m\] "
elif [ $CONN = tel −a $USR = priv ] ; then
  PS1="\[\033[01;30;45m\][\u@\h \W]\\$\[\033[00m\] "
elif [ $CONN = ssh −a $USR = nopriv ] ; then
  PS1="\[\033[01;32m\][\u@\h \W]\\$\[\033[00m\] "
elif [ $CONN = ssh −a $USR = priv ] ; then
  PS1="\[\033[01;35m\][\u@\h \W]\\$\[\033[00m\] "
fi

# PS1="[\u@\h \W]\\$ "
export PS1
alias which="type −path"
alias dir="ls −lF −−color"
alias dirs="ls −lFS −−color"
alias h=history
```

12.6 A Prompt the Width of Your Term

A friend complained that he didn't like having a prompt that kept changing size because it had $PWD in it, so I wrote this prompt that adjusts its size to exactly the width of your term, with the working directory on the top line of two.

```
#!/bin/bash

#   termwide prompt
#      by Giles − created 2 November 98
#
#   The idea here is to have the upper line of this two line prompt
#   always be the width of your term.  Do this by calculating the
#   width of the text elements, and putting in fill as appropriate
#   or left−truncating $PWD.
#

function prompt_command {

TERMWIDTH=${COLUMNS}

#   Calculate the width of the prompt:

hostnam=$(echo −n $HOSTNAME | sed −e "s/[\.].*//")
#   "whoami" and "pwd" include a trailing newline
usernam=$(whoami)
let usersize=$(echo −n $usernam | wc −c | tr −d " ")
newPWD="${PWD}"
let pwdsize=$(echo −n ${newPWD} | wc −c | tr −d " ")
#   Add all the accessories below ...
let promptsize=$(echo −n "−−(${usernam}@${hostnam})−−−(${PWD})−−" \
                 | wc −c | tr −d " ")
let fillsize=${TERMWIDTH}−${promptsize}
fill=""
while [ "$fillsize" −gt "0" ]
do
   fill="${fill}−"
   let fillsize=${fillsize}−1
done

if [ "$fillsize" −lt "0" ]
then
   let cut=3−${fillsize}
   newPWD="...$(echo −n $PWD | sed −e "s/\(^.\{$cut\}\)\(.*\)/\2/")"
fi
}

PROMPT_COMMAND=prompt_command

function termwide {

local GRAY="\[\033[1;30m\]"
local LIGHT_GRAY="\[\033[0;37m\]"
local WHITE="\[\033[1;37m\]"
local NO_COLOUR="\[\033[0m\]"

local LIGHT_BLUE="\[\033[1;34m\]"
local YELLOW="\[\033[1;33m\]"
case $TERM in
    xterm*)
        TITLEBAR='\[\033]0;\u@\h:\w\007\]'
        ;;
    *)
        TITLEBAR=""
        ;;
esac

PS1="$TITLEBAR\
$YELLOW-$LIGHT_BLUE-(\
$YELLOW\${usernam}$LIGHT_BLUE@$YELLOW\${hostnam}\
${LIGHT_BLUE})-${YELLOW}-\${fill}${LIGHT_BLUE}-(\
$YELLOW\${newPWD}\
$LIGHT_BLUE)-$YELLOW-\
\n\
$YELLOW-$LIGHT_BLUE-(\
$YELLOW\$(date +%H%M)$LIGHT_BLUE:$YELLOW\$(date \"+%a,%d %b %y\")\
$LIGHT_BLUE:$WHITE\$$LIGHT_BLUE)-\
$YELLOW-\
$NO_COLOUR "

PS2="$LIGHT_BLUE-$YELLOW-$YELLOW-$NO_COLOUR "

}
```

12.7 The Elegant Useless Clock Prompt

This is one of the more attractive (and useless) prompts I've made. Because many X terminal emulators don't implement cursor position save and restore, the alternative when putting a clock in the upper right corner is to anchor the cursor at the bottom of the terminal. This builds on the idea of the "termwide" prompt above, drawing a line up the right side of the screen from the prompt to the clock. A VGA font is required.

Note: There is an odd substitution in here, that may not print properly being translated from SGML to other formats: I had to substitute the screen character for \304 − I would normally have just included the sequence "\304", but it was necessary to make this substitution in this case.

```
#!/bin/bash

#   This prompt requires a VGA font.  The prompt is anchored at the bottom
#   of the terminal, fills the width of the terminal, and draws a line up
#   the right side of the terminal to attach itself to a clock in the upper
#   right corner of the terminal.

function prompt_command {
#   Calculate the width of the prompt:
hostnam=$(echo -n $HOSTNAME | sed -e "s/[\.].*//")
#   "whoami" and "pwd" include a trailing newline
usernam=$(whoami)
newPWD="${PWD}"
#   Add all the accessories below ...
let promptsize=$(echo -n "--(${usernam}@${hostnam})---(${PWD})-----" \
                 | wc -c | tr -d " ")
#   Figure out how much to add between user@host and PWD (or how much to
#   remove from PWD)
let fillsize=${COLUMNS}-${promptsize}
fill=""
#   Make the filler if prompt isn't as wide as the terminal:
while [ "$fillsize" -gt "0" ]
do
   fill="${fill}Ä"
   # The A with the umlaut over it (it will appear as a long dash if
   # you're using a VGA font) is \304, but I cut and pasted it in
   # because Bash will only do one substitution - which in this case is
   # putting $fill in the prompt.
   let fillsize=${fillsize}-1
done
#   Right-truncate PWD if the prompt is going to be wider than the terminal:
if [ "$fillsize" -lt "0" ]
then
   let cutt=3-${fillsize}
   newPWD="...$(echo -n $PWD | sed -e "s/\(^.\{$cutt\}\)\(.*\)/\2/")"
fi
#
#   Create the clock and the bar that runs up the right side of the term
#
local LIGHT_BLUE="\033[1;34m"
local     YELLOW="\033[1;33m"
#   Position the cursor to print the clock:
echo -en "\033[2;$((${COLUMNS}-9))H"
echo -en "$LIGHT_BLUE($YELLOW$(date +%H%M)$LIGHT_BLUE)\304$YELLOW\304\304\277"
local i=${LINES}
echo -en "\033[2;${COLUMNS}H"
#   Print vertical dashes down the side of the terminal:
while [ $i -ge 4 ]
do
   echo -en "\033[$(($i-1));${COLUMNS}H\263"
   let i=$i-1
done

let prompt_line=${LINES}-1
#   This is needed because doing \${LINES} inside a Bash mathematical
#   expression (ie. $(())) doesn't seem to work.
}

PROMPT_COMMAND=prompt_command

function clock3 {
local LIGHT_BLUE="\[\033[1;34m\]"
local     YELLOW="\[\033[1;33m\]"
local      WHITE="\[\033[1;37m\]"
local LIGHT_GRAY="\[\033[0;37m\]"
local  NO_COLOUR="\[\033[0m\]"

case $TERM in
    xterm*)
        TITLEBAR='\[\033]0;\u@\h:\w\007\]'
        ;;
    *)
        TITLEBAR=""
        ;;
esac

PS1="$TITLEBAR\
\[\033[\${prompt_line};0H\]
$YELLOW\332$LIGHT_BLUE\304(\
$YELLOW\${usernam}$LIGHT_BLUE@$YELLOW\${hostnam}\
${LIGHT_BLUE})\304${YELLOW}\304\${fill}${LIGHT_BLUE}\304(\
$YELLOW\${newPWD}\
$LIGHT_BLUE)\304$YELLOW\304\304\304\331\
\n\
$YELLOW\300$LIGHT_BLUE\304(\
$YELLOW\$(date \"+%a,%d %b %y\")\
$LIGHT_BLUE:$WHITE\$$LIGHT_BLUE)\304\
$YELLOW\304\
$LIGHT_GRAY "

PS2="$LIGHT_BLUE\304$YELLOW\304$YELLOW\304$NO_COLOUR "

}
```