

# Customising Your Home Environment

Kevin Waldron  
<http://www.zazzybob.com>

November 2, 2004

Version 0.1

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Initialisation Files</b>	<b>1</b>
2.1	Brief Outline of Shell Initialisation . . . . .	1
2.2	<code>\$HOME/.profile</code> . . . . .	2
2.2.1	Basic Customisation . . . . .	2
2.2.2	Further Customisation . . . . .	4
2.2.3	A Final Note . . . . .	5
2.3	<code>\$HOME/.kshrc</code> . . . . .	5
2.3.1	Aliases . . . . .	6
2.3.2	Functions . . . . .	10
<b>3</b>	<b>Home Directory Layout</b>	<b>13</b>
3.1	<code>\$HOME/bin</code> . . . . .	13
3.2	<code>\$HOME/docs</code> . . . . .	14
3.3	<code>\$HOME/etc</code> . . . . .	14
3.4	<code>\$HOME/sbin</code> . . . . .	14
3.5	<code>\$HOME/src</code> . . . . .	14
3.6	<code>\$HOME/tmp</code> . . . . .	14
3.7	Final Notes . . . . .	14
<b>4</b>	<b>Conclusion</b>	<b>15</b>

# 1 Introduction

I use the Korn Shell (`ksh`) for my day to day shell work, and over my many years twiddling with the world of UNIX, I have come to like my home environment to be set up a certain way on all machines I use. By this, I mean that I always want certain aliases set, certain variables exported, my `PATH` formed a certain way, and so forth. Logging into a machine where the environment isn't customised like this leads me to pound my fist on the desk in frustration and moan at whoever is sitting next to me as to why `r` isn't aliased to `fc -e -`

Therefore, I am publishing this document. It serves no other real purpose than documenting the way I like things set up. This, of course, is probably a world apart from how you like things done, but if any of the material here is of use to you, then that's fine and dandy.

Here, I will discuss my shell initialisation files, my home directory structure, and a few scripts and other things I couldn't work without.

Most of what's here, in terms of the scripts and `.kshrc` file, is available for download from the sprawling chaos that is my humble abode on the web, <http://www.zazzybob.com>, which is probably the same place you downloaded this document from.

This document deals with the Korn Shell only (well, the Public Domain Korn Shell, `pdksh`, to be precise). Therefore, users of other shells will find that their milage may well vary. I will use `ksh` in place of `pdksh` within this text for clarity, as the two shells are, for most intents and purposes, identical.

The section on Home Directory Layout should be useful no matter which shell you use.

A quick note about this document, if I say:

*“Type `FOO=bar` at the command line”*

I assume that you'll be savvy enough to press return afterwards. I don't explicitly overstate the obvious.

## 2 Initialisation Files

### 2.1 Brief Outline of Shell Initialisation

This section deals with the initialisation of the Korn Shell, and is provided to give a brief overview of what `ksh` does when it is invoked. It is not authoratitive, but should help you understand the rest of the document.

When `ksh` is invoked as a login shell, it first reads the contents of `/etc/profile`, where any system-wide variables are placed and exported. The login shell next reads your personal profile, whose location is `$HOME/.profile` where any variables you want to be made

available to this and all subshells (i.e. global environment variables) should be placed and exported.

Within your personal `.profile` file, if you have set and exported the `ENV` environment variable, the login shell will then read and execute the file specified by this variable (if it exists and is readable, obviously!). This file is also read by each sub (i.e. non-login) shell that you launch during your session. It is inside this file that you should place your aliases and functions.

The standard filename for this file is `$HOME/.kshrc`, and you would, therefore, need the following lines in your `$HOME/.profile` in order to ensure that `ksh` knows where to find it:

```
ENV=$HOME/.kshrc
export ENV
```

Or, in one line

```
export ENV=$HOME/.kshrc
```

This file will then be read by the login shell itself, and any subshells (non-interactive too - I'll get onto this later) that you spawn. You can name this file whatever you want, but for the remainder of this document, I will assume that you haven't been a pedant/show-off/buffoon and have named it `$HOME/.kshrc` like everyone else!

I would like to stress the following point. The `LOGIN` shell reads (if the files exist and are readable) `/etc/profile`, `$HOME/.profile` and whichever file is pointed to by the `ENV` variable (if set), in that order. Non-login shells (i.e. subshells) will read only the file that is pointed to by `ENV`. This is all very `ksh` specific. You should read your own shells manpages.

## 2.2 `$HOME/.profile`

### 2.2.1 Basic Customisation

The `.profile` file can be used to set any environment variables that you want to be made available to all subshells. You do this by setting the variable and then exporting it. First and foremost, with `ksh`, we want to set the `ENV` variable, so that the shell knows where to find our `.kshrc` file that contains our aliases and functions.

Therefore, the first line in my `.profile` file is

```
ENV=$HOME/.kshrc
```

Now `ksh` will be able to read the file specified by `ENV` (well, after it has been exported, to be pedantic) and execute the commands contained therein.

Next, I add any other search directories I may require to my `PATH` variable. These include `sbin` directories, non-standard paths to executables (for example, the Java Development Kit), and also (rather naughtily), the current directory, “.”

I am well aware that any security concious readers will be throwing up their hands in horror and berating me for having the current directory in my `PATH`. However, I am not `root` here, I am a standard user, and it saves me prefixing everything with `./` which in turn saves me two keystrokes! Obviously, the current directory should never be in `root`'s `PATH`!

As you'll know, `PATH` is searched from left to right. We want to add a few more directories to our `PATH`, and we also want the original `PATH` defined by the system (usually in `/etc/profile`) to be searched first. How do we "save" the value of the current `PATH` and just append our own customisations to the end? Easy - using basic shell programming. At the end of the day, `.profile` and `.kshrc` are just shell scripts, nothing more, nothing less. Therefore, the same syntax is used. Let's see my `PATH` customisation to clarify this.

```
PATH=$PATH:/usr/lib/SunJava2-1.4.1/bin:/home/kevin/bin:/sbin:/usr/sbin:.
```

As you can see, I set `PATH` to be equal to the current `PATH`, plus a colon delimited list of any subsequent directories to add to the search path. You will need to adjust the order of the directories here to suit your requirements/paranoia. I add the current directory “.” last. My reasoning for this? Well, say you leave your workstation to go to the loo, and foolishly forget to run `xlock`. Somebody grabs the opportunity to mount a floppy, copy a malicious executable into your current directory (say, your `HOME` directory), with the same name as a system executable (`ifconfig` for example). They then `umount` the floppy and leave your workstation (after clearing the terminal screen). You then sit down at your workstation. You've got the current directory “.” in your `PATH`, and like a fool you've got it BEFORE `/sbin`. You want to check the IP address of the system, you type `ifconfig eth0` at the command line, and *whoops!* you've run the malicious file in the current directory and, even though you're not `root` (let's hope not, anyway) it's done a recursive delete of your home directory. Oh dear. Where are those backup tapes?

I know this is a simple and silly example, although it does serve to illustrate the issue. No right-minded user/administrator would make such a mistake. But be warned about the power of having the current directory, “.” in your `PATH`!

The next two variables that I set are the `EDITOR` and `VISUAL` variables. With `ksh`, these variables define which command line editing mode will be used with interactive shells. These variables are also used by certain applications that require the use of a default editor. Technically, `EDITOR` is the default editor (traditionally a line editor), and `VISUAL` is the default full-screen editor (a visual editor). These days, gone is the need to use `ed` and `ex` and other

line-editors (although I still keep a knowledge of them - you never know when you might need to make a quick edit over an old slow modem link). When determining which command line editing mode to use, the `VISUAL` variable is used first by `ksh`, and if it is not set, then `EDITOR` is used instead. If the variable is set to one of `vi`, `emacs` or `gmacs`, then the command-line editing style of that particular editor will be used by your interactive shells. Now, I wouldn't touch `emacs` with a 10-foot editing pole, and am very much a "*vi-lover*", so that's what I use for the value in both of these variables.

```
EDITOR=vi
VISUAL=vi
```

In order for `ksh` to remember the commands you've typed, and for its history capabilities to work, you need to store the path to your history file (I use `$HOME/.ksh_history`) in the `HISTFILE` variable. So, a simple assignment is all that's necessary

```
HISTFILE=$HOME/.ksh_history
```

There are many more variables that you can set which alter/enhance the way that `ksh` works. `man ksh` will tell you more. You can set `HISTSZ` which specifies how many commands are kept in the history, but the default of 128 is enough for me, so I don't need to set it.

I have now set my "essential" variables, and thusly can export them with a single `export` command

```
export ENV PATH EDITOR VISUAL HISTFILE
```

Now, all of these variables will not only be made available to the login shell, but to all subshells too.

## 2.2.2 Further Customisation

I also define several other variables within my `.profile` file, some of which are needed by certain tools and applications (for example, Python has its `PYTHONPATH` variable, among others). I won't cover application specifics here, as you'll need to read the documentation supplied with the specific tool/package to decide whether any environment variables need to be set.

I do, however, define several variables that I use regularly. For example, I have a backup copy of my website stored on my local filesystem. To save keep typing the path each time, I have set a `WEB` variable

```
export WEB=/files/zazzybob.com
```

Now, when I want to, for example, view a file in my web directory, I can simply type `more $WEB/index.html` and save those keystrokes! You can do this for other things too. Let's suppose that you're always using a certain application, `/some/nonstandard/location/foo_munge`. Now, you could type in that long path each time you wanted to run the program, or you could add the directory path to your `PATH` variable. I would just set a variable thusly

```
export FOO=/some/nonstandard/location/foo_munge
```

Then, you just need to type `$FOO` on the command line, and the variable will be expanded, unleashing the wonderful `foo_munge` upon your machine!

### 2.2.3 A Final Note

I think that I've covered enough here for you to be able to customise your `.profile` to your liking.

I should also mention that because `.profile` is only read in by your login-shell, you should log out, then back in again in order to apply any changes you have made. Some shells support a `-l` option which causes them to behave like a login shell - this can be useful for testing your changes. Why don't you just source the file? If you were to type `. $HOME/.profile` at the command line your `PATH` variable would be set again, for starters, re-appending your customisations and in effect doubling them up. Nonsense. Don't do it. This kind of "apply-changes-now" chicanery is fine for `.kshrc` but not for `.profile`.

## 2.3 \$HOME/.kshrc

As discussed previously, `.kshrc` is read by each subshell, whether it is interactive or not. What do I mean by this? Well, suppose you've got a script, `foo.ksh`. You run that script, and it executes in a subshell, therefore executing all the commands in your `.kshrc`. With a large and complex `.kshrc` file, this will cause your scripts to slow down as more and more is executed every time a subshell is forked. Bad news.

However - when we launch an interactive shell, by typing `ksh` at the command line, or firing up an `xterm` under X, for example, we DO want the commands in `.kshrc` to be executed, so that the features become available to us in that shell. So, we need a way to distinguish between interactive and non-interactive shells. We can then check this at the top of the `.kshrc` file and exit if the shell is non-interactive, thus preventing execution of unnecessary commands and improving performance.

Luckily, the Bourne Shell and its variants have a builtin way of checking this - with the `$-` variable. This variable holds a concatenated list of the current shell options which have been set. For an interactive shell, the "i" option will be set, so typing `echo $-` at the command line will yield "ims", or similar. The "m" enables job control, and the "s" ensures that default input is read from stdin. The value of `$-` will probably be different on your system, but we're only interested in the "i" option here. Some of these options are set automatically, depending on how the shell was launched, and others can be set by the user using the `set` command. `set -x` will enable tracing which is good for debugging scripts. This is beyond the scope of this document however, and we are going off at a tangent. View the manpage for your shell if you want to know more.

Anyone with a modicum of shell programming experience will now see that a simple way to stop the `.kshrc` in its tracks for non-interactive shells is to include a `case` statement right at the beginning of the `.kshrc` file which checks the value of the `$-` variable, and exits if it doesn't contain an "i".

For those without said modicum - see below:

```
# Check that this is an interactive shell - if not, exit
case $- in
  *i*) ;;
  * ) return 0 ;;
esac
```

Include this at the top of your `.kshrc` file, and execution will promptly cease if the current shell is not interactive. If, however, the shell is interactive, the `*i*` case will be true, and the rest of the file will be executed.

I know I've laboured the point and banged on about this one, but this is one of the finest shell initialisation tricks I know.

Some shells (notably `bash`) seem to default to using damned `emacs` line editing commands. I am a `vi` user, so to have the land of CTRL and ALT thrust upon me is rather annoying. Thankfully, because of the way I set up my `EDITOR` and `VISUAL` environment variables in `$HOME/.profile`, `vi` editing mode is assumed by default.

For reference, you can turn of `emacs` mode and enable `vi` mode by inserting the following two lines into your `.kshrc`. However, I find it more logical to set the correct variables in `$HOME/.profile`

```
set +o emacs
set -o vi
```

There is one set command that I cannot work without

```
set -o vi-tabcomplete
```

Now, as far as I am aware, this is a Public Domain Korn Shell (`pdksh`) only feature. This enables TAB file name and command name completion. Every saved keystroke is useful! If you don't have this feature, you can use `ESC-\` (Escape then Backslash) under `vi` editing mode to facilitate filename completion too.

### 2.3.1 Aliases

Yet another boon to using `ksh` is that several common aliases are pre-defined. One of these is the `r` command. `r` is aliased to `fc -e -` which allows us to recall previous commands with ease. For example, suppose we've just compiled something

```
gcc -Wall my_file.c -o my_file $(gtk-config --cflags --libs)
```

The compiler freaks out and informs us that our code is like swiss cheese. We go back and edit the file, and want to recompile. Now we could write makefiles, etc, to streamline this compilation. However, we're lazy. We just type

```
r gcc
```

and the last `gcc` command issued is substituted onto the command line and executed. Beautiful.

There are a handful of other aliases built into `ksh`, but this document is about customisation, not utilisation, so read the manual page if you want the details. We want to start defining some useful aliases of our own.

## Keeping it logical

Now, I don't practice what I preach here (such is the hackish nature of my working!), but I would recommend that you break down your alias definitions within `.kshrc` into a logical structure. For example list all of your `ls` related aliases together, all of your `vi` related aliases together, and so on. Also, put commented headers within the file to distinguish between the sections. As your `.kshrc` grows and sprawls, you'll be thankful for this!

In the following sections, I will discuss and define some of the aliases I have defined within my `.kshrc` file. Most of these aliases are of sense and use only to me, but I will present them here lest they be of use to anyone.

## ls-type aliases

I primarily use Linux systems, and these systems come with GNU (<http://www.gnu.org>) versions of the standard UNIX utilities. These utilities are usually enhanced greatly, whilst still retaining all of the standard capability of (and remaining totally compatibly with) the original UNIX command set.

One such command with greatly enhanced functionality is the humble `ls` command. There are two options to this command (known as *GNU long-style options* that I use frequently. These are the `--color` and `--time-style` options.

The `--color` option allows us to view (on ANSI-compliant terminals) the output from the `ls` command in a nice easy-to-digest colour format. I use `--color=auto`. The `auto` part tells `ls` not to use colour formatting if I'm piping to another process or redirecting to a file. You can use `none` which disables colour completely, or `always` which will cause `ls` to always

use colour. Always using colour is not good for redirecting to text files, for example, as you'll see the control characters. And imagine the disaster when you pipe the output of `ls --color=always` to another command! This is why I use "intelligent" colour output - with `--color=auto`.

The `--time-style` option allows us to specify a date format to be used for `ls -l` listings. I am British, therefore I want a British date format. Using normal date formatting characters (see `man date` for more information), I can set this option thusly `--time-style=+%d-%m-%Y\ %H:%M`. Using the `alias` command, I can then redefine `ls` to behave the way I would like. Notice that I am backslash-escaping the space in the formatting sequence.

```
alias ls="ls --color=auto --time-style=+%d-%m-%Y\ %H:%M"
```

Before I discuss other aliases, I should touch upon what's happening here. This is basically saying "Alias the `ls` command, so that if `ls` typed on the command line, it's automatically replaced by `ls --color=auto --time-style=+%d-%m-%Y\ %H:%M` before the command is itself executed." - In a nutshell - this is how aliasing works. If we were to type

```
ls -la /bin
```

at the prompt, what we are actually executing is

```
ls --color=auto --time-style=+%d-%m-%Y\ %H:%M -la /bin
```

after expansion.

I also alias some other common `ls` variations. Take a look at the following

```
alias ll='ls -l'
alias lla='ls -la'
alias llt='ls -lt'
alias ltr='ls -ltr'
```

It should be very obvious what's going on here, so I won't dwell. I think that even though these are simple aliases, they serve to illustrate the power of a good alias. I am not a happy man when I use a system that does not have `ll` set up!

## cd-type aliases

I find that another very good use for aliases is to set up shortcuts to common directory changes. For example, I have the following `cd` based aliases in my `.kshrc`

```
alias cb='cd $HOME/bin; pwd'
alias cdocs='cd $HOME/docs; pwd'
alias cproj='cd /files/projects; pwd'
alias cw='cd /files/zazzybob.com; pwd'
alias cj='cd /files/websh/proof; pwd'
alias ct='cd /files/latex; pwd'
```

Again, I am not going to dwell too much on what's happening above. You will see that I have single quoted commands (') with variables inside (\$HOME). Won't this prevent evaluation? No. When I type `cb` at the command line, it is expanded to `cd $HOME/bin; pwd`, losing the quotes. The shell evaluates the command before execution, which expands the value of \$HOME before the `cd` command itself is ever called.

Also, note how I call `pwd` after each `cd` command - this is just for clarity so that I can see where I'm cd-ing too.

## vi-type aliases

I usually set up some aliases for common files that I edit using `vi` - **THE** editor (I'm not biased, am I?!).

```
alias vh='su -c "vi /etc/hosts"'
# under some systems (HP-UX for one)
# alias vh='su - root -c "vi /etc/hosts"'
alias vk='vi $HOME/.kshrc'
alias vp='vi $HOME/.profile'
alias vs='vi /admin/var/log/sys.maint.log'
```

One handy alias that I'd like to discuss is the `vh` alias above. If you're managing a large network, and constantly find yourself `su`-ing to `root` in order to edit the `/etc/hosts` file, this is a good shortcut to use. Some \*nix systems implement their `su` command differently, as you can see from the above variations of the alias. Now, when you type `vh` at the command line, you'll be prompted for `root`'s password, and then (assuming you enter the correct password) the file will open in `vi` for editing. Once you save and exit `vi`, your session as `root` will end and you'll assume your usual identity. This is a very useful technique, and can be used in a variety of situations. However, be aware that some old versions of `su` do not support the `-c` option.

## Other aliases

I now go on to fill my `.kshrc` with a variety of useful aliases - most of which are specific to applications that I have installed, and need not be discussed here. I have, however, defined a handful of aliases that should be useful for most people, and I will describe these below.

```
alias ftz='ftp username@ftp.hostingsite.com'
```

The above alias allows me to connect to my hosting providers FTP site with ease. Obviously, I could script the complete FTP transfer, but my FTP requirements change with each

connection. Therefore, I just save myself the work of establishing the connection, and leave the actual transfer interactive.

```
alias mntcd='mount /media/cdrom'  
alias umntcd='su -c "umount /media/cdrom"'
```

These CD-ROM aliases are perhaps my most frequently used. These will depend on your exact mount points and mount options (consult your `/etc/fstab` for more information). I have my main system set up so that anybody can mount the CD-ROM, but only root can unmount it, hence the alias definitions above.

```
alias shutdn='su -c "shutdown -h now; exit" && exit'
```

Another frequently used alias. This will `su` to root, execute the `shutdown` command, and logout of the current terminal.

Some examples of other aliases follow. These will conclude my discussion of aliases within `.kshrc`.

```
alias p='pwd'  
alias c='clear'  
alias pc='echo "There are `ps aux --no-headers | wc -l` processes"'  
alias nla='nl -ba'  
alias sk='. $HOME/.kshrc'  
alias tk='top -u kevin'  
alias dut='du -s'
```

### 2.3.2 Functions

Another common use of the `.ksrhc` file is to define functions - commonly used routines that you want to always be available. Now, it can be more efficient, useful and functional to code these as separate scripts, and it is up to your specific requirements to decide what you need to do.

For example, I have the following function defined in my `.kshrc`

```
function lnp  
{  
    ping 192.168.0.$1 -c 2  
}
```

This enables me to ping a host on my local network. I just issue a command such as `lnp 182`, at the prompt, and the host 192.168.0.182 is pinged twice. This could be implemented as a script, too, but I try to steer away from one-line scripts where possible.

Functions can be suitably complex too - consider the following example. This is one that probably should be coded as a separate script, but I use it here to serve as an example of the complexity that functions can employ

```
function up
{
  _UP=`uptime | awk '{print $3}' | tr ',' ' '`
  _MINS=`echo ${_UP} | awk -F : '{print $2}'`
  _HRS=`echo ${_UP} | awk -F : '{print $1}'`

  if [ "${_HRS}" -eq "1" ]; then
    echo "We've been up for ${_HRS} hour and ${_MINS} minutes"
  else
    echo "We've been up for ${_HRS} hours and ${_MINS} minutes"
  fi
}
```

Here's a fairly useful function, that displays all files in a specified directory that were modified today

```
function today
{
  #files modified today

  _TODAY=`date +%d-%m-%Y`

  ls -l "$@" | grep "${_TODAY}" | more
}
```

It goes without saying that the `date` format I use above is very sepecific to my needs - due to the fact that I use the GNU long style option `--time-style` to define my `ls -l` output date format. You should also notice that I use `"$@"` above, so that any parameters passed are expanded properly when the shell comes to interpret the command. This causes any positional parameters to be expanded as single words. If you want to learn more about `$@` consult your shells manpages.

On HP-UX, which uses the traditional `ls -l` date format, I have to do some jiggery-pokery with `sed` to account for leading blanks on dates less than 10. On this system, therefore, I implement `today` as a script which can be seen below

```
#!/bin/sh
# ls of all files modified today

_MONTH=`date +%b`
_DAY=`date +%d`
# sed to take care of leading zeros
_FIXED_DAY=$( echo $_DAY | sed 's/[0]\([0-9]\)/\1/' )

ls -l "$@" | grep "$_MONTH[ ]*$_FIXED_DAY " | more

exit 0
```

Because `date +%d` generates dates less than 10 with a leading zero, and the `ls -l` command prefixes them with a leading space instead, the `sed` command removes the leading zero, and the regular expression in the `grep` command allows for any number of spaces.

Another function I use allows me to see an `ls -l` listing of any given executable in my `PATH`.

```
function showme
{
  which $1 > /dev/null 2>&1
  if [ "$?" -eq "0" ]; then
    echo "$1 is $(which $1)"
    ls -l `which $1`
  else
    echo "$1 not found in PATH"
  fi
}
```

The next function just breaks down the output of `runlevel` and presents it to me in an easy to digest format

```
function rl
{
  #display runlevels
  _prev=$( runlevel | awk '{print $1}' )
  _curr=$( runlevel | awk '{print $2}' )
  echo "Previous Runlevel:\t${_prev}"
  echo "Current Runlevel:\t${_curr}"
}
```

You may need to use `echo -e` on your system for the tab to be evaluated.

I have only scraped the surface of what can be implemented using functions, but I think I have illustrated that they can be as complex or as simple as you wish.

This concludes my discussion of the `.kshrc` file, and of shell initialisation.

## 3 Home Directory Layout

Home Directory layout is very much a personal thing. Most programs will dump a ton of nonsense into it (especially with the advent of GUI's - every program you run will create a configuration file or a directory under `$HOME`!). Run `ls -la $HOME | more` to see what I mean!

This aside, we can organise the rest of our home directory in a logical and structured manner. I choose a directory layout which mimics the standard UNIX heirarchical filesystem. I do not use `*all*` of the UNIX root filesystem directories. I don't write shared libraries and thusly do not have a `~/lib` directory. A `~/dev` directory would make no sense at all.

You will see from the above that I am using the tilde (`~`) to represent `$HOME`. This is commonplace within modern Bourne Compatible shells (notably `ksh` and `bash`). You can also reference individual users home directories using the tilde - my home directory is `~`, or fully `~kevin`. Therefore, I can say that Jim's home directory is `~jim`. Again, the usual get-out here as we're deviating from the subject is read the man pages for your shell.

### 3.1 `$HOME/bin`

The first subdirectory in my mini-UNIX filesystem is `~/bin`. Predictably, I use this directory for holding all of my shell scripts, C programs, Ruby scripts, and other "executable" programs. Then, I can just add `$HOME/bin` to my `PATH` and have all of the programs become available to me.

As a matter of personal choice, I do not create any subdirectories off of this directory.

### 3.2 \$HOME/docs

I know, I know! Where on the UNIX root filesystem is the "docs" directory?! Well, I have amassed a ton of technical documentation and need somewhere to put it, so this seems like the best choice. On my main system, I actually use a `/files` filesystem hanging off of the root filesystem for holding this very sort of miscellany.

I subdivide my `~/docs` directory for various programming languages and other topics.

### 3.3 \$HOME/etc

I use the `~/etc` directory to hold configuration files used by my own programs. I also have a subdirectory, `~/etc/bak`, which I use for storing backup copies of important configuration files (of course these are then included in my routine system backups).

### 3.4 \$HOME/sbin

Within this directory, I store programs that I have written which must be executed as `root`, or have the potential for causing major changes to the system. As an example, I have scripts which add users to the system, monitor security, etc. I do not include this directory in my `PATH`, so I must reference things explicitly when I want to use them.

### 3.5 \$HOME/src

I use this directory to hold source code for programs I have written. This directory is then divided into subdirectories for each language that I work with.

### 3.6 \$HOME/tmp

Much the same as the traditional `/tmp` and `/var/tmp` directories, I use this for temporary files of any nature, and also for running test scripts and programs in a chrooted environment. I should empty this directory more than I do!

### 3.7 Final Notes

This is by no means an exhaustive list of the contents of my home directory, but it does serve to illustrate the general structure of it. As at follows the standard UNIX hierarchical filesystem layout, I know where to look for a file when I want it. Keeping your home directory organised like this is not only good practice, but also highly efficient.

## 4 Conclusion

Having the right home environment available to you can make the difference between an enjoyable experience and a mouse-bashing keyboard-mashing disaster. I hope that this document has given newbies a taster of what's available, and experienced geeks a few new ideas.

Visit my website at <http://www.zazzybob.com> where you can find a large and growing script repository (holding around 70 scripts as of 4th July 2004), as well as a large number of full programs and utilities, and other articles.

Cheers  
Kevin Waldron

[kevin@zazzybob.com](mailto:kevin@zazzybob.com)