

R2'S SHELL TUTORIALS

src: <http://www.injunea.demon.co.uk/pages/page201.htm>
copied oct/2009

Table of Contents

Preface.....	5
Introduction.....	6
Example code segment.....	6
Example screen output.....	6
Example Command Line.....	6
The Basic Shells.....	7
Why Use Shells?.....	7
The Shell History:.....	7
Bourne Shell:.....	7
C Shell:.....	7
Korne Shell:.....	7
Read That Manual!:.....	8
What is a Shell Script Anyway?:.....	8
Example basic shell script.....	8
In the Beginning:.....	8
The Borne Shell Syntax (sh, jsh, rsh):.....	10
Invocation Flags:.....	10
Definitions:.....	10
Blanks:.....	10
Names:.....	11
Parameters:.....	11
Script example_1.1 - The shell default parameters.....	11
Screen output for script example_1.1.....	11
Simple, Complex and Special Commands:.....	14
Simple Commands.....	14
Complex Commands.....	14
The for structure:.....	14
Example for syntax.....	14
The while structure:.....	15
Example while syntax.....	15
The if structure:.....	16
Example simple if syntax.....	16
Example complex if syntax.....	16
case structure:.....	16
Example case syntax.....	16
The parent and sub-shell structure:.....	17
The function structure:.....	17
Example function syntax.....	17
Special Commands:.....	19
Comment structure:.....	20
Command Substitution:.....	21
Example grave accent.....	21
Quoting and Escaping.....	21
Example quoting.....	22
Parameter Substitution:.....	22
Input and Output.....	23
Syntax Checking:.....	23
Example syntax checking.....	23
Protect Yourself:.....	23

Example file exists.....	23
Example verify.....	24
Reading Keyboard Input:.....	24
Example dot list.....	24
Example overwrite.....	25
Where's My Print Gone?:.....	25
Temporary File Generation:.....	26
Filenames.....	26
Using Files.....	28
Simple File Creation:.....	28
Example cat and variables.....	28
Example simple echo.....	29
Complex File Creation:.....	29
Example complex echo forms.....	29
Example list.....	29
Example counted list.....	30
Example sorted list.....	30
File Reading:.....	30
Example reverse list and count words.....	30
Example tail -f option.....	30
Pipes, Lists and Redirection.....	32
Pipe Dreams:.....	32
Example pipes.....	32
Lists:.....	32
Current Shell:.....	33
Sub-Shell:.....	33
Redirects:.....	33
Example redirected cat.....	33
Example indented cat.....	34
Example simple menu.....	34
Functions.....	36
Simple Date and Time functions:.....	36
Simple File Set-up Functions:.....	39
Simple String Functions:.....	42
Simple Menu Functions:.....	46
Simple Utility Functions:.....	47
SQL*Net 1.0 Functions.....	48
Start Up Files and Environment.....	50
What Are They?:.....	50
Why Use Them?:.....	50
Example useful profiles.....	50
Example cron test function.....	51
Example raw cron test.....	52
Example setting user environment.....	52
Environment:.....	52
Example environment variables.....	52
Debugging Scripts.....	54
Flags:	54
example flags.....	54
Echo:	55
Example echo.....	55
Null Parameter Trap:.....	56
Null Parameter Example.....	56
Error Messages:.....	56
Emergency Exit:.....	56

R2's Shell Tutorial

Design Considerations.....	58
Shell Script Style Guide.....	58
Banners:.....	58
Banner Example.....	58
Detailed Banner Notes:.....	59
Lesser Banners:.....	60
Example of Function or Procedure Banners:.....	60
Example of Logical_Block Banners:.....	60
Example of Function Definition with Logical Blocks:.....	60
Symbols:.....	61
Symbol Form:.....	61
Class Form:.....	61
Layout:.....	61
Layout Rules:.....	61
Layout Example:.....	62
Putting It All Together.....	63
Database Generator Script:.....	63
generate_db listing.....	63
Dedication.....	73
Appendix A - Start Up Files.....	74
Oracle's .cshrc file.....	74
Oracle's .profile.....	78
Oracle's .profile_references.....	79
Oracle's .profile_paths.....	81
Oracle's .profile_READ_ME.....	82

Preface

Shell programming was not something I wanted to do from the start, it kind of crept up on me over the years. You know how it is, you do something regular enough that

it gets boring and you try to find short cuts. Well after my first UNIX training course I went back to the office and started to look at the support requirements and it soon became obvious that to do a good job something had to be done regarding the overload of administration on a detailed level as the system grew. Initially my scripts were small and dedicated , and to do the same thing for several users I just copied the script into their home directory and edited a few lines and off it went. After a few weeks, I spotted the error of my ways and created a directory for scripts that could be shared. I put the directory into everyone's path and started moving scripts into this central place. By editing the scripts and putting in variables, instead of hard coded values, I found I could make them generic enough for anybody to use, as long as the user's environment contained enough background information for the script to pick up and use.

A few months down the line and I saw another opportunity as I was aware there was a lot of common code in most of These scripts. Why keep copying all this code when I could call it from another file? Two things sprang to mind initially, one was the .profile, which I was already using to store users environment variables, the other was functions, which I had seen one of my friends use in a long script suite. I dug out the man pages for the shells and spent the next few days looking at **sh**, **csh**, and **ksh** in detail, trying out short test scripts to make sure I understood what was written. I had a look in my local book store for books on shell programming generally. Alas the only books available were very basic and covered what I had already learned. What I did pick up at this time was that there was a general disliking for the C Shell in most books I found. It was pointed out on numerous occasions how many bugs there were. I must admit, even the man pages themselves highlight several pages of bugs at the back. I was also aware that to fully support a networked system there were several files you had to be able to edit and fix when things changes. These are the basic system set up files and spool daemons etc. All of these files are of course written using the Bourne Shell. You become quite good at the Bourne Shell when you set up printer drivers for a networked drawing archival system with distributed printers and plotters.

What I started to see here was a language that was very succinct - terse, but to the point. It had few bugs - certainly no one remarked on them. It was also very portable in that all UNIX systems support it and make use of it exclusively for low level set up scripts and installation suites. I found this a bit of a nuisance because I was a C Shell programmer and I liked my cosy environment! But when I saw these functions used, I started to see real benefits - C Shell does not have functions. I converted a few scripts, it's not too difficult - just a few key word changes, the if syntax, etc. Then I put in a few functions to replace some common code. Well, look at that! The scripts is now only half the size and still does the same thing! What's more, if I change this function, all the scripts that call it are immediately updated too! This is neat! The rest, as they say, is history.

Introduction

This book is aimed at those UNIX users who find themselves looking after more shell scripts than they can comfortably shake a stick at. These will be System Administrators or Specialists looking after a legacy system or even a brand new development team that is subject to constant changes. The scope is indeed broad, as is the application of what is contained within this book. Be on the lookout at all times for opportunities to simplify. Always look for a generic solution, which will always have wider application than the dedicated solution.

An understanding of UNIX will be helpful, but the reader need not have any shell programming experience. In fact to possess any existing bad shell scripting habits may hamper progress.

The conventions used in this book are very simple. The main text is in the same font and style as has been used here. Any examples of code will be presented as either one line or a numbered listing in Courier 8 point bold which looks like this:

Example code segment

```
# Example Code Example  
echo "Hello World"  
echo "The time is [ `date +%h%m%s%p` ] by my clock"
```

When required, screen output will use the same font as code but in a regular type instead of bold which looks like this:

Example screen output

```
user@system$ my_script argument  
Hello World  
The date is [12:37:42pm] by my clock
```

UNIX Command words, like **ls** or **cp** will be picked out in the text as bold, while variables used within the text body will be in *italic* text. Examples of UNIX command lines will be printed in Arial 10 point bold text which looks like this:

Example Command Line

```
cat *.log | sort -u > filename
```

And finally a Health Warning for Real Programmers: Real programmers may get upset by some of the constructs used here. I make no apologies to the incensed reader. Just remember, there are no correct ways in shell programming, just ways that work. I hope you have as much fun trying out the concepts used in this book as I had while writing it.

The Basic Shells

Why Use Shells?:

Well, most likely because they are a simple way to string together a bunch of UNIX commands for execution at any time without the need for prior compilation. Also because it's generally fast to get a script going. Not forgetting the ease with which other scripters can read the code and understand what is happening. Lastly, they are generally completely portable across the whole UNIX world, as long as they have been written to a common standard.

The Shell History:

The basic shells come in three main language forms. These are (in order of creation) **sh**, **csh** and **ksh**. Be aware that there are several dialects of these script languages which tend to make them all slightly platform specific. Where these differences are known to cause difficulties I have made special notes within the text to highlight this fact. The different dialects are due, in the main, to the different UNIX flavours in use on some platforms. All script languages though have at their heart a common core which if used correctly will guarantee portability.

Bourne Shell:

Historically the **sh** language was the first to be created and goes under the name of The Bourne Shell. It has a very compact syntax which makes it obtuse for novice users but very efficient when used by experts. It also contains some powerful constructs built in. On UNIX systems, most of the scripts used to start and configure the operating system are written in the Bourne shell. It has been around for so long that it is virtually bug free. I have adopted the Bourne shell syntax as the de facto standard within this book.

C Shell:

Next up was The C Shell (**csh**), so called because of the similar syntactical structures to the C language. The UNIX man pages contain almost twice as much information for the C Shell as the pages for the Bourne shell, leading most users to believe that it is twice as good. This is a shame because there are several compromises within the C Shell which makes using the language for *serious* work difficult (check the list of bugs at the end of the man pages!). True, there are so many functions available within the C Shell that if one should fail another could be found. The point is do you *really* want to spend your time finding all the alternative ways of doing the same thing just to keep yourself out of trouble. The real reason why the C Shell is so popular is that it is usually selected as the default login shell for most users. The features that guarantee its continued use in this arena are aliases, and history lists. There are rumours however, that C Shell is destined to be phased out, with future UNIX releases only supporting **sh** and **ksh**. Differences between **csh** and **sh** syntax will be highlighted where appropriate.

Korne Shell:

Lastly we come to The Korne Shell (**ksh**) made famous by IBM's AIX flavour of UNIX. The Korne shell can be thought of as a superset of the Bourne shell as it contains the whole of the Bourne shell world within its own syntax rules. The extensions over and above the Bourne shell exceed even the

level of functionality available within the C Shell (but without any of the compromises!), making it the obvious language of choice for real scripters. However, because not all platforms are yet supporting the Korn shell it is not *fully portable* as a scripting language at the time of writing. This may change however by the time this book is published. Korn Shell does contain aliases and history lists aplenty but C Shell users are often put off by its dissimilar syntax. Persevere, it will pay off eventually. Any **sh** syntax element will work in the **ksh** without change.

Read That Manual!:

On any UNIX system the on-line documentation available from the man pages covers all the commands available, including the script languages. For Bourne it's usually between 10 and 15 pages whereas for C Shell or Korn it's about 30 pages or more. It is a good idea to get this printed out and put into a loose leaf folder. The on-line documentation is always more up to date than that in the printed manuals that arrive with the system and if stored in normal A4 folders (I use the clear plastic pockets to store the loose pages) this can be used as your reference source. People usually gasp in horror when they hear this - "Why not just open a new window and use man!" - they cry. Well for one thing, if I know the information I want is towards the back, I can jump straight to the page rather than fight with the scrollbar. Secondly, if the information is wrong (Heaven Forbid!) or not too clear (Really?), I can add some of my own clarification in the margins! The all important *dialect* or platform specifics will also be listed within these man pages.

The other thing you must do is cultivate an ability to read *through* these man pages and understand the meaning *behind* the words. This is a trick that takes time, don't think for a moment that what is written in a man page is actually correct. It is simply someone else's idea of how to explain what is supposed to happen under known circumstances. Don't forget that it's rare for the code writer to be the documentation writer too. This ability will put you in good stead for negotiating the most troublesome passages. If it doesn't make sense the first time, change your view and look at it from another angle. Develop this skill well, it will pay dividends later.

I have included some syntactical information from my man pages within this document but I have kept it to the minimum required and always used examples to explain each concept, something sadly lacking in man pages generally. All examples will be repeated for each shell for comparison when appropriate, although I now only use Bourne shell full time because of its portability (See and).

What is a Shell Script Anyway?:

A shell script, in its most basic form, is simply a collection of operating system commands put into a text file in the order they are needed for execution. Using any of the shells mentioned so far, a text file containing the commands listed in would work every time the script was executed.

Example basic shell script

```
#!/bin/sh
rm -f /tmp/listing.tmp          > /dev/null 2>&1
touch /tmp/listing.tmp
ls -l [a-z]*.doc | sort        > /tmp/listing.tmp
lpr -Ppostscript_1 /tmp/listing.tmp
rm -f /tmp/listing.tmp
```

Of course not all scripts are this simple but it does show that ordinary UNIX commands can be used without any extra, fancy scripting constructs. If this script was executed any number of times the

result would be the same, a long listing of all the files starting with lower case letters and ending with a **doc** extension from the current directory printed out on your local PostScript printer. Not very exciting. Lets start to look at this in detail starting at the beginning.

In the Beginning:

The first line of any script should always look a bit like the top line in , the only difference would be the path leading to the shell executable file, in this case the **/bin/sh** part. By default the UNIX system will attempt to execute an ASCII text file if the files name is supplied as the first argument on the command line. UNIX will attempt to execute the file in the current shell, and try to process the included command strings within the file using the syntax rules of the current shell. So, if you are using the Bourne Shell as your default environment and the ASCII file contains a list of UNIX command structures formatted how the Bourne Shell likes them to be formatted, all will work fine. However, if you try and execute a C Shell file with a different syntax structure, the Operating System will complain about unrecognised commands and syntax errors. The reason is, no one told the Operating System that it was a C Shell file, so it processed it as a Bourne Shell. The correct way to indicate this to the Operating System is to pass the script name to the shell executable as an argument thus:

```
user@system$ /bin/csh my_script arg_1 arg_2
```

However, it didn't take long for someone to notice that this was extra typing¹ that could well be done away with and hence the short hand *first Line* comment was born (See -).

Basically, all comments in shell scripts start with a hash sign (#). However, for the first line only, UNIX reads past the hash to see what's next. If it finds an exclamation point (!), or *Bang!*, as it's known, then what follows is taken as the path to the shell executable binary program. Not only that, but all the command line arguments that the shell executable allows can also be stacked up on this line (See -). With this feature it doesn't matter what flavour of shell your environment is, you can execute a script in any other flavour of shell, as if it was a real UNIX command. Lets look at the Bourne Shell syntax rules more closely. It would be helpfull at this point to print out the **sh** man pages from your system so that you can compare them with what I have here.

¹ There is a general rule among UNIX systems programmers that states - "Any command which has more than two letters in it's name, is too difficult to spell!"

The Bourne Shell Syntax (sh, jsh, rsh):

The next few sections are all based on the syntax rules for the Bourne shell as listed in the man pages for sh from my system. Compare these rules with those of your system. There may be slight differences where the specification of the sh is loosely defined. Where there are these minor differences, experiment for yourself to validate that what is written agrees with what happens. Trust no-one and test everything until you are happy you fully understand each point. Only then can you begin to use the sh to your advantage. Make the syntax rules second nature. Lets read what it says.

The Bourne Shell is available in three forms on most systems. These are:

- **sh** The Standard Bourne Shell
- **jsh** The Job Control Bourne Shell
- **rsh** The Restricted Bourne Shell

The syntax and usage rules are common across all these types except where noted. In general, the **rsh** is more secure and forces users to comply with additional rules imposed by the system manager, while the **jsh** adds some features which aid the control of background processes (batch jobs) from the users interactive session.

Lets look at the standard *man page* information and interpret what this represents in some real world examples. First is the SYNOPSIS section which gives very brief syntax information regarding all versions of the shell. On my system, the list of three lines shows **sh** and **jsh** then **/usr/lib/rsh** indicating that the path for the restricted shell is not normally included in the users path. This is because of an unfortunate conflict between the spelling of **rsh** and **rsh** (!) One being the restricted shell, the other being the remote shell command which allows a shell process to be started on a remote system. For instance you might want to list your home directory on a remote machine but not want to login and do any work on the system. To do this the command **rsh remote_system ls -l** where *remote_system* is the alias of the remote machine, would be useful.

Invocation Flags:

In square brackets following the command name is a list of *flag* parameters which modify the way the command behaves. You do not need to use any of these, indeed the square brackets indicate that they are optional, but quite a few are useful on occasion. In common with most man pages my system lists the *flag characters* then forgets to say anything else about them until pages 13 (under SET) and 15 (under INVOCATION) by which time the reader has completely forgotten where they came from. It is never clarified on my system that the SET *flags* are the same as the ones listed under the SYNOPSIS section. However there is an inference at the end of SET which indicates “**\$1, \$2 etc.**, following the *flags*, will be treated as input parameters for the shell” - that's your only clue. The flags will be covered as appropriate within the text where relevant. I won't bother elucidating the DESCRIPTION section as this has been covered in some detail above.

Definitions:

The next bit on my systems *man pages* is DEFINITIONS where it tries to explain some very basic facts about key words used in the rest of the document. Some of these definitions are not always very clear and a misinterpretation here can lead to later confusion. Lets try and take these one step at a time.

Blanks:

A **blank** is a **tab or space**. What this actually means is - a **blank** is any chunk of white space between anything that is printable (a character or word). So **blank** can be several *spaces* or *tabs* or a combination of *multiples* of the two.

Names:

A **name** is a **sequence of ASCII letters, digits, or underscores, beginning with a letter or an underscore**. Well, almost. What they are really saying here is - these are the rules for a *variable name* or *function name* within a shell script program. What has been omitted here is that the *names* are case sensitive, you can mix case within a *name* (LikeThisOne), and they don't always have to start with a letter or an underscore (See -). It is never stated what the length limit is for a *name*. The limit on my system is 31 characters. Names longer than 31 characters do not give rise to any error messages, but if you have several names which only differ after character 32, then the shell will treat them all as the same variable. This can lead to unexpected results. You have been warned.

Parameters:

A **parameter** is a **name, a digit, or any of the characters *, @, #, ?, ~, \$, and !\^**. So, what's the difference between a *name* and a *parameter* exactly? Not much actually, it's all in the usage. If a *word* follows a command, as in: **ls -l word**, then *word* is one of the *parameters* (or *arguments*) passed to the **ls** command. But if the **ls** command was inside a **sh** script, then in all likelihood the *word* would also be a *variable name*. So a *parameter* can be a *name* when passing information into some other command or script. Viewed from inside a script however, the command line *arguments* appear as a line of *positional parameters* named by *digits* in the ordered sequence of arrival (See -). So a *parameter* can also be a *digit*. The other *characters* listed are *special characters* which are assigned values at script start up and may be used if required from within a script.

Well after reading through the above, I am still not sure if this is any clearer. Lets see if an example can help to clarify things a little.

Script example_1.1 - The shell default parameters

```
#!/bin/sh -vx
#####
# example_1.1 (c) R.H.Reepe 1996 March 28 Version 1.0 #
#####
echo "Script name is          [$0]"
echo "First Parameter is      [$1]"
echo "Second Parameter is     [$2]"
echo "This Process ID is      [$$]"
echo "This Parameter Count is  [#]"
echo "All Parameters          [$@]"
echo "The FLAGS are           [$_]"
```

If you execute the script listed in with some arguments as shown below, you will get the output on your screen that appears in .

```
user@system$ example_1.1 fred bill bert
```

Screen output for script *example_1.1*

```
+ echo "Script name is          [$0]"
Script name is          [example_1.1]
+ echo "First Parameter is     [$1]"
First Parameter is     [fred]
+ echo "Second Parameter is    [$2]"
Second Parameter is    [bill]
+ echo "This Process ID is     [$$]"
This Process ID is     [16219]
+ echo "This Parameter Count is [$#]"
This Parameter Count is [3]
+ echo "All Parameters         [$@]"
All Parameters         [fred bill bert]
+ echo "The FLAGS are          [$_]"
The Flags are          [vx]
```

Looking back at the example script, in the first line of the file there is a special sequence of characters (`#!`) which the shell will only interpret on the first line. Normally the hash character indicates to the shell that this is the start of a comment and the shell must ignore everything up to the next newline character. However, when on the first line, the shell will go on to read the path to the shell executable program and optionally some shell flags (See -). I have added the flags `-xv` here because they are very useful when debugging. The `-v` flag is the verbose setting (also available part way through a script by using `set -v` if required) which forces the shell to output or **echo** each command it finds in the script as it encounters it. This will allow you to find which particular line in your code has the syntax error, output will stop at this point and the script exits. The `-x` flag is similar except that it puts a plus sign (+) in front of any command that gets processed. This is not quite the same as `-v` which will show you the command whether it is processed or not (See - which shows output from both `-v` and `-x` together). If you process a loop structure for instance, the `-v` will output the whole construct once as it is seen, but the `-x` will show each pass through the loop too. The path shown on the first line is for the Bourne shell. For C shell use `/usr/bin/csh` and for Korn shell use `/usr/bin/ksh`.

The next three lines are my default header. See Section for information on script style, layout and symbol format.

Next is the body of the script which displays to the terminal or **echoes** some text strings and some values. You will note I have put each *variable/parameter/name(!)* inside some square brackets. This is a good way of checking for included blank space within a variable's value. I would not expect to see any blanks in any of these variables but when debugging, it's a good idea to check. The first three are *positional parameters* which will display the *parameters* following the command name (or script) when executing. The first of these is `$0` which is the command (or script) name itself. This is a useful thing to have as you can use this when outputting errors or building logfiles or audit trails. The *real* input parameters are available from `$1` to `$9` inclusive. What if you have more than 9 parameters? Well there is a *shift* feature, which we will cover later (See -), which gives access to parameters above 9. Incidentally, the *dollar symbol* (\$) at the front of all these variables is a request to the shell to substitute the *value* of the variable at that point. All variable names used in *all* the shell types need to be prefixed with the dollar if you want the value substituted (See -).

Next is an odd looking one called `$$` which returns the *process id* of this script. When UNIX executes a script it will create a *process* to handle the work and this is its *number*. It is an integer between 1 (unlikely!) and 32767 on most systems. Every task that is run on a UNIX system has its own *process*

id which is why the number 1 is unlikely². There will be tens (maybe hundreds) of processes already running when you login and you just get the next available. When UNIX runs out of *process id* numbers, it wraps around and re-uses defunct *process id*'s by starting again at the lowest available number. The \$\$ parameter is not just a pointless random number generator. It is very useful when creating temporary files for instance, where each instance of the script³ can create a unique temporary filename based on the *process id* (or PID).

Then we have \$# or the *parameter count*. This returns an integer number representing the number of *positional parameters* (the *\$digits*) following the script name. In our example in that would be 3 but I have not found a real limit, except when exceeding the UNIX line length. When dealing with counts larger than 1 this is a useful loop control parameter for use with the *shift* feature (See -).

Next we have the @\$ parameter which lists out the complete set of *positional parameter* values found on the command line (excluding \$0), a handy way to pass them all on to a sub-script or function.

Lastly I have included the \$- parameter which will list out the current *flags* in use. This parameter is volatile and will be updated to reflect the status of any **set** commands processed during script execution (See - for a complete listing of invocation flags).

² it is usually used by one of the bootstrap programs during system start up.

³ several users may all be using it at the same time.

Simple, Complex and Special Commands:

Back in the man pages the next section is called USAGE and goes on to talk about *pipelines* and *lists*. Most of what it says here can be understood by any UNIX user so I will skip this for now but there will be some examples later showing various implementations of these definitions. The issue I want to deal with next is the simple, complex and special commands. This is nowhere near as bad as it sounds.

Simple Commands

Simple commands are just straight UNIX commands that exist regardless of the surrounding shell environment. Like our old favourites **ls -l** or **df -al** or **lpr -Pprinter filename**. There are large numbers of commands that fall into this category but the following list is a selection of the more useful when scripting.

- **sort** Sorts lines in ascending, descending and unique order
- **grep** Searches for regular expressions in strings or files
- **basename** Strips the path from a path string to leave just the filename
- **dirname** Removes the file from a path string to leave just the pathname
- **cut** Chops up a text string by characters or fields
- **wc** Count the characters, words, or lines
- **[(test)]** Predicate or conditional processor
- **tr 'a' 'b'** Transform characters
- **expr** Simple arithmetic processor
- **bc** Basic Calculator
- **eval** Evaluate variables
- **echo** Output strings
- **date** Create date strings
- **nawk** Manipulate text strings
- **head | tail** Access lines in files

Some of the above commands can be very complex indeed, especially when assembled into pipelines and lists. However, these are still referred to as simple commands - presumably because they stand alone. Take a close look at the man pages for all of the above commands, you will find them invaluable during your scripting sojourn.

Complex Commands

Complex commands are just the shells internal commands which are used to group simple commands into controlled sets based on your requirements. These include the loop constructs and conditional test structures. These cannot stand alone. An **if** requires a **then** and a **fi** at the very least. Lets take a look at the man pages again at this point.

The for structure:

It says on my systems man page **for name [in word ...] do list done** as a syntax description of the *for* command construct. Well, it is correct but does not really show the layout of the command at all. Look at Example and you can see straight away what is supposed to happen.

Example for syntax

```
alphabet="a b c d e"           # Initialise a string
count=0                       # Initialise a counter
for letter in $alphabet       # Set up a loop control
do                             # Begin the loop
    count=`expr $count + 1`    # Increment the counter
    echo "Letter $count is [$letter]" # Display the result
done                           # End of loop
```

So in plain English, **for** each *letter* found in *alphabet* loop between **do** and **done** and process the *list* of commands found. Lets take this one line at a time from the top. This is the way the **sh** likes to have its variables set. There is no *leading* word as in the **csh** (**set**) just start with the *variable name*. There are also no blanks either side of the equal sign. Indeed, if you put a blank in, the shell will give you an error message for your trouble. This also gives rise to the difference between the top two lines in this example. Because I want to include spaces in my string for *alphabet*, I must enclose the whole string in *double quotes*. On the next line this is not required as there are no embedded *blanks* in the value of *count*. When setting variables, no *blanks* are allowed. Everywhere else, **sh** loves *blanks*.

In line 3 the **for** statement creates a *loop construct* by selecting the next *letter* from *alphabet* each time through the loop and executing the *list* found between the **do** and the **done** for each *letter*. This process also strips away any *blanks* (before and after) each *letter* found in *alphabet*⁴. The **do** and **done** statements are not executed as such, they simply mark the beginning and end of the loop *list*. They are however a matched pair, leave one out and the shell will complain.

Inside the *loop* are two *simple commands* (apparently!). The first one just increments the *loop counter* by adding one to its current value. Note the use of the *back-quote* here to force the execution of the **expr** command before setting the new value of *count*. There will be more about this later.

The next line is something we have seen before, just a display command showing the values of the variables. Note the use of the **\$** symbol to request the *value* of the variables.

The while structure:

There is another similarly structured command in the **sh** called **while**. Its syntax structure is listed as **while list do list done** which you should now be able to translate yourself into something that looks like Example below.

Example while syntax

```
alphabet="a b c d e"           # Initialise a string
count=0                       # Initialise a counter
while [ $count -lt 5 ]        # Set up a loop control
do                             # Begin the loop
    count=`expr $count + 1`    # Increment the counter
    echo "Letter $count is [$letter]" # Display the result
done                           # End of loop
```

⁴ Including *newlines* if the list is a result of an **ls -1** command

Most of this is the same construct, I have just replaced the **for** loop set-up with its equivalent **while** syntax. Instead of stepping through the *letters* in *alphabet*, the *loop control* now monitors the size of the *count* with [*\$count -lt 5*]. The **-lt** flag here represents *less-than* and is part of the **test** UNIX command, which is implied by the *square brackets*. Any other command, list or variable could be put here as long as its substituted value equates to an integer. A zero value will exit the loop, anything else and the loop will continue to process. From the above you can work out that **test** returns 1 for *true* and 0 for *false*. Have a look at the man pages for **test** at this point, you will find it a very useful command with great flexibility.

The if structure:

Next in complexity is **if list then list [elif list then list] ... [else list] fi**, or the if construct. What does that lot mean? Well usually **if** statements in any language are associated with predication and so as you would expect there is some more implied use of the UNIX **test** command. Lets generate an example to see the structure in a more usual form. The square brackets in the echo statement have no relevance other than to clarify the output when executed (See - Section).

Example simple if syntax

```
if [ -f $dirname/$filename ]
then
    echo "This filename [$filename] exists"
elif [ -d $dirname ]
then
    echo "This dirname [$dirname] exists"
else
    echo "Neither [$dirname] or [$filename] exist"
fi
```

You can see here more examples of what **test** can do. The **-f** flag tests for existence of a *plain file*, while **-d** tests for existence of a *directory*. There is no limit (that I can discover) to the number of **elif**'s you can use in one **if** statement. You can also stack up the tests into a *list* using a *double pipe* or *double ampersand* as in below. Here the use of the *double pipe* (| |) is the syntax for a logical **or** whereas the *double ampersand* (&&) is the logical **and**.

Example complex if syntax

```
if [ -f $dir/$file ] || [ -f $dir/$newfile ]
then
    echo "Either this filename [$file] exists"
    echo "Or this filename [$newfile] exists"
elif [ -d $dir ]
then
    echo "This dirname [$dir] exists"
else
    echo "Neither [$dir] or [$file or $newfile] exist"
fi
```

In the **sh if** construct it is important to put the **then** word on its own line or **sh** will complain about an invalid **test**. Also important is the *blank* inside each end of the **test**. Without this the **test** will generate a syntax error - usually "test expected!" which is a bit meaningless.

case structure:

Next is the **case word in [pattern [pattern] ...) list ;;] esac** which is probably the most complicated construct to decode from the simple syntax listed above. It is a bit like a multi-line **if** statement linked with logical **or** symbols (|). It is commonly used to process a list of parameters passed into a script as arguments when the actual parameters could be in any order or of any value. The layout is shown in below, which is a section from a print script.

Example case syntax

```

size=0                                # Default Char Point Size (!)
page=660                               # Default Page Point Size
while [ "$1" != "" ]                  # When there are arguments...
do                                     # Process the next one
    case $1                             # Look at $1
    in
        -l)    lines=47;                # If it's a "-l", set lines
                page=470;                # Set the Landscape Page Point
                options="$options -L -l"; # Set the Landscape Options
                shift;                    # Shift one argument along
        -p)    lines=66;                 # If it's a "-p", set lines
                options="$options -l";    # Set the Portrait Options
                shift;                    # Shift one argument along
        -s)    size=$2;                  # If it's a "-s", set size
                shift 2;                  # Shift two arguments along
        *)    echo "Option [$1] not one of [p, l, s]"; # Error (!)
                exit;;                   # Abort Script Now
    esac
    if [ $size = 0 ]                    # If size still un-set...
    then
        size=`echo "$page / $lines" | bc` # Set from pages over lines
    else
        lines=`echo "$page / $size" | bc` # or
    fi
    # Set lines
done
options="$options$lines -s$size"        # Build complete option list
lp -P$PRINTER $options $filename        # Output print file to printer

```

Here we see a **while** loop, exiting when no more parameters are found on input line, enclosing a **case** statement. The **case** statement repeatedly tests **\$1** against a list of possible matches indicated by the right parentheses. The star (*) at the end is the default case and will match anything left over. When a match is found, the list of commands following the right parentheses are executed up to the double semi-colon. In each of these lists, there is a **shift** statement which shifts the input parameters one place left (so **\$2** becomes **\$1** etc.), allowing the next parameter to be tested on the next pass through the loop. In the case of the "-s" parameter, an extra following argument is expected, the *size* value, which is why the **shift** instruction contains the additional argument 2 (shifting the parameters 2 spaces left). This effectively allows the processing of all the passed arguments in any order and includes an exit for an invalid parameter condition via the star match. The **if** statement at the end checks if the size parameter has been set then uses the **bc** command to set either *size* or *lines* accordingly. When complete, the final options are created and passed to the **lp** command to print the file.

The parent and sub-shell structure:

Then there are two easy ones the `(list)` and `{ list; }` constructs which simply execute the whole *list* of commands in a separate sub-shell `()` or in the parent shell `{ }` with a note that the *blanks* between the `{ }` are mandatory.

The function structure:

Lastly in the complex command section we come to what is probably the most underused but most useful construct for serious scripters. The function definition. The syntax is deceptively simple which I guess is what leads most users to assume it's not worth learning about. How wrong they are. Just take a look at this Example to see what I mean.

Example function syntax

```
i_upper_case()
{
    echo $1 | tr 'abcdefghijklmnopqrstuvwxy' 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
}
```

This is a very simple function called `i_upper_case`, you can probably guess what it does. Note that it gets its input argument from a passed parameter (`$1`). So to make use of this function within a script you simply need to call it with an argument as follows:

```
i_upper_case "fred"
```

or

```
name="fred"
i_upper_case $name
```

And you will get back `FRED` in either case. A more appropriate usage would be something like:

```
small_name="$input_argument"
large_name=`i_upper_case "$small_name"`
echo "Large Name = [$large_name]"
```

Which allows the case to be changed and put into a new variable. The advantage of doing this at all is that you don't have to re-code the same thing over again when you want to use the feature several times within the script. Note the use here of the double quotes around the variables to the right of the equal signs - this is to preserve any blanks within the strings which would otherwise be treated as argument separators and hence the function would only process the first argument in the list. What this means is:

```
small_name="fred smith"
large_name=`i_upper_case "$small_name"`      # Quoted parameter
echo "Large Name = [$large_name]"
```

Will display `FRED SMITH`, whereas:

```
small_name="fred smith"
large_name=`i_upper_case $small_name`      # Unquoted parameter
echo "Large Name = [$large_name]"
```

Will display **FRED** only. This *bug* can be traced back to the *function definition* which only reads in the **\$1** *parameter*. Changing this to read the **\$@** *parameter* would correct the *bug* for this function. But beware, this type of fix would not be appropriate in all situations. Try and think generically when creating functions and make them as useful as possible in all situations.

There are two very basic rules to remember when dealing with functions:

1. You cannot use a function until it is defined. Thus all function definitions should appear either at the top of the script or in a start-up file such as `~/profile`.
2. Functions can be nested to any depth, as long as the first rule is not violated.

At the end of the complex command section there is a reminder message that all of the keywords used in these complex commands are reserved and not available as variable names. This means that you can screw up any UNIX command by using it as a variable but you cannot screw up a complex shell reserved word.

```
echo()
{
    /usr/bin/user/my_echo "$@"
}
```

Is perfectly okay as a function definition and the **sh** will happily use your **echo** function whenever an **echo** command is required within the script body.

```
while()
{
    /usr/bin/user/my_while "$@"
}
```

Is not okay and the function definition will fail at runtime.

Special Commands:

The following are a set of special commands which the shell provides as stand alone statements. Input and output redirection is permitted for all these commands unlike the complex commands. You cannot redirect the output from a **while** loop construct, only the simple or special commands used within the loop list.

- The colon (**:**) does nothing! A zero exit code is returned. Can be used to stand in for a command but I must admit not to finding a real use for this command.
- The dot (**.** *filename*) we have already met. It allows a file to be read into the current environment. If the *filename* following the dot is not in the current working directory, then the shell searches along the *PATH* variable looking for a match. The first match that is found is the file that is used. The file is read into the shell and the commands found are executed within the current environment.
- The break (**break** [*n*]) command causes an exit from inside a **for** or **while** loop. The optional *n* indicates the number of levels to break out from - the default is one level. Although not stated in the syntax rules, I have used this statement in an **if then else fi** construct to good effect in where it causes an exit from the function but does not cause an exit from the calling script.
- The continue (**continue** [*n*]) command resumes the next iteration of the enclosing **for** or **while** loop at the [optional *nth*] enclosing loop. Can't say I've used this one either.

- The `cd` (**cd** [*argument* ¼]) command is the *change directory* command for the shell. The directory is specified with *argument* which defaults to *HOME*. The environment variable *CDPATH* is used as a search path for directories specified by *argument*.
- The `echo` (**echo** [*argument* ¼]) command is the shell output statement. See the man pages for `echo(1)` for full details.
- The `eval` (**eval** [*argument* ¼]) command reads the *arguments* into the shell and then attempts to execute the resulting command. This allows pre-emptive parameter substitution of hidden parameters or commands.
- The `exec` (**exec** [*argument* ¼]) command reads in the command specified by arguments and executes them in place of this shell without creating a new process. Input an output arguments may appear and, if no others are given, will cause the shell input and or output to be modified.
- The `exit` (**exit** [*n*]) command causes a shell to exit with the exit status specified by the *n* parameter. If the *n* parameter is omitted, the exit status is that of the last executed command within the shell.
- The `export` (**export** [*variable* ¼]) command we have already met and is the command which makes shell *variables* global in scope. Without a *variable*, `export` will list currently exported variables.
- The `getopts` command is provided to support command syntax standards - see `getopts(1)` and `intro(1)` man pages for details.
- The `hash` (**hash** [**-r**] [*name* ¼]) command remembers the location in the search path (*PATH* variable) of the command *name*. The option **-r** causes the shell to forget the location of *name*. With no options the command will list out details about current remembered commands. This has the effect of speeding up access to some commands.
- The `newgrp` (**newgrp** [*argument* ¼]) command is equivalent to `exec newgrp argument`. See `newgrp(1M)` for usage and description. Haven't got a clue!
- The `pwd` (**pwd**) command literally prints the current working directory. Usually used to set the *CWD* variable internally.
- The `read` (**read** *name* ¼) command will be seen in several examples. It allows the shell to pause and request user input for the variable *name*, which is then accepted as the variables value.
- The `readonly` (**readonly** [*name* ¼]) command sets a variable as immutable. Once named in this command they cannot be reassigned new values.
- The `return` (**return** [*n*]) command causes a function to exit with the return value *n*. If the *n* is omitted, the return value is the exit status of the last command executed within the function. Unlike `exit` this does not result in termination of the calling script.
- The `shift` (**shift** [*n*]) command causes the positional parameters to be moved to the left (*\$2* becomes *\$1*, etc.) by the value of *n*, which defaults to one.
- The `test` command is used to evaluate conditional expressions. See the man pages for `test(1)` for full details and usages.
- The `times` command prints the accumulated user and system times for processes run from the shell.
- The `trap` (**trap** [*argument*] [*n*] ¼) command allows conditional execution of the commands contained within *argument* dependant on the shell receiving numeric or symbolic signal(s) *n*.
- The `type` (**type** [*name*] ¼) command indicates how *name* would be interpreted if used as a command name.
- The `ulimit` and `umask` commands exist in their own right as UNIX commands. See man pages.
- The `unset` (**unset** [*name*] ¼) command allows *names* to be unset. This removes the values from the variable or function. The *names* *PATH*, *PS1*, *PS2*, *MAILCHECK*, and *IFS* cannot be unset.

- The wait (**wait** [*n*]) command waits for the background process *n* to terminate and report its termination status; where *n* is the *process id*. With no arguments, all current background processes are waited for.

Most of these special commands get used somewhere in this book and more detailed explanations will follow at that time.

Comment structure:

The next thing on my systems man page is a reference to the hash (#) comment character. It states that any word beginning with # causes that word and all the following characters up to a newline to be ignored. There are no notes about the first line exceptions that I gave in when we were dealing with shell indicators (The #! sequence)

Command Substitution:

Next up is COMMAND SUBSTITUTION although there is a lot about quoting in this section too. It is far too difficult to split quoting away altogether, so instead I have elected to combine both sections and include all the QUOTING information here too where appropriate.

We have already seen the set of *back-quotes* (or grave accents) in use in Examples and where the *count* variable was incremented. The shell reads the command, or list of commands, or pipeline from between the grave accents and executes them. The output from this operation is then available to the shell for use in setting a variable or echoing to the terminal. When the command string is executed in this way, the output is stripped of newlines and blanks are truncated to one space between words (unless preserved in double-quotes). The Example shows several different uses of the grave accent of varying complexity.

Example grave accent

```
current_dir=`pwd`  
line_count=`wc -l $file_name | cut -c1-8`  
third_path=`echo $PATH | cut -f3-3 -d:`  
last_word=`head -$line_count $file_name | tail -1 | tr "/" | basename`
```

In order, these commands and pipelines are all used to set some variable values.

- First is getting the current directory using the **pwd** command, the simplest usage. Just execute the command and use the output to set the variable.
- Next is getting the count of lines in a file. Use the **wc** command (word count) with the **-l** flag (lines) then *pipe* this to the **cut** command and use the **-c** flag (characters) to cut out the 1st to 8th character (the *number*) from the output. Use this to set the variable. The **cut** is required because **wc** insists on including the *filename* in the output string. Also beware of the *number* part, as it is not an *integer* but a *string of numbers*. If you want a number, then use the **expr** command to add a zero to the result which has the effect of converting the type. Although I said earlier that shell variables are un-typed, the **test** command gets upset when presented with a string and is then told to evaluate it as greater or less-than another number. The variables maybe typeless, however their contents are typed unless converted.
- Next is getting the third string from a complex string list. The *PATH environment variable* (See - Section) is a list of search paths for the UNIX system to use when searching for command locations within the directory structure. Each path is separated from its neighbour by a colon thus: [/bin:/usr/bin:/usr/lib:/usr/etc]. This command *pipeline* echoes the path string into **cut** using the **-f** flag (field) to select the third field. The **-d** flag specifies the delimiter character for the fields as a *colon* in this case. Once again the result of the executed pipeline is used to set a variable.
- Lastly finding the last word on a particular line. Use the **head** command with the *line_count* variable to echo out the top *line_count* lines of a file. Pipe this through the **tail** command and only pass the last one line. Having got the line you want, pipe this through **tr** (transform) and change each *blank* into a *forward-slash* (a UNIX directory separator). Then pipe this through **basename** which always strips away the full path leaving just the filename. This leaves you with just the last word, however long the line was, and however many words there were on the line, which can then be used to set our variable.

Quoting and Escaping

The difference between the single and double quote are also useful to know. The single quote is what you use to enclose a literal string. Whatever is between single quotes remains unchanged by the shell. If you enclose something in double quotes then concatenated blanks are preserved (as in the literal case with the single quotes) but variables are substituted by their values and filenames are expanded from their wild-cards to full filenames and or paths. The best way to see this is in below.

Example quoting

```
my_name="Fred Smith"      # Set a variable
echo "$my_name"          # Will output - Fred Smith
echo '$my_name'          # Will output - $my_name
```

This may give the impression that the single quote is not much use. However, if you want to echo a dollar sign out to another script file for later execution, it can be very handy. See below.

```
echo "echo \"\$my_name\" > $file_name"
```

Here the **echo** statement is creating another **echo** statement in an output file for later use. The *\$my_name* variable is hidden inside the *single-quotes* and so will be passed as a literal string to the output file. Another feature shown here is the *back-slash* which is used as an *escape signal* for the shell. It tells the shell not to read the next character, but just pass it on. In this way the *double-quote* also makes it to the output file unscathed. The backslash can be used to escape any character that is special to the shell - even a newline. This allows you to create very long lines (See - Section). To pass a *back-slash* use a *double back-slash*.

Parameter Substitution:

As noted earlier, the key character here is the dollar symbol (\$) which instructs the shell to substitute the value where the parameter name (or variable) is located. There is a touch more flexibility here than at first appears however. There are in fact 5 different ways that the shell can substitute the value depending on the syntax used.

1. `${name}` The value, if any, is substituted. The braces are only required if *name* is followed by characters which are not to be interpreted as part of the *name* string.
2. `${name:-word}` The value, if any, is substituted. If the value is NULL, substitute *word* instead. *Word* can be a literal string, another parameter or a substituted command string output. Can be used to substitute a positional parameter (*\$digit*).
3. `${name:=word}` As above but cannot be used to substitute a positional parameter.
4. `${name:?word}` If *name* is set and is not NULL, substitute its value. Else output *word* and exit from shell. If *word* is missing, the message 'parameter null or not set' is output. This can be a useful debugging aid.

5. `${name+word}` If *name* is set and is not NULL, substitute *word*. Else substitute nothing. This means the parameter is always substituted by *word* unless parameter is null, in which case it remains null.

Input and Output

As we saw in Section , UNIX is flexible about where it takes its input and where it puts its output. This should also be true of any scripts you create. Have a care for the poor user who is trying to make use of your script. Give him a few clues if he goes wrong, tell him what's happening if there's likely to be a delay, etc. One of the worst things you can do in a script is take someone's input then display a frozen screen for 10 seconds. That is the most stressful 10 seconds in that user's life! Has it hung? Did I do it right? Is it still waiting for something from me? The best way to handle this is with some simple checks at the start of the script to test for syntax and immediate feedback when the user has entered something. If at all possible, make the script flexible enough to be able to handle alternate usages.

Syntax Checking:

When you write your script, you know its internal structure and specification. You know how it works and what it expects as input. Don't expect the users to be that aware, they usually assume you have done everything for them anyway, you might just as well - it'll save time later on!

If a script has 2 mandatory arguments (say input and output) then it's just plain silly to try and run it with only one. It's also not too bright to run it when there are three (er... like which two should it choose?). Thankfully there is a very simple check you can do to see if a script has been started with the correct number of arguments, and it relates to `$#` - a shell variable which is set when a script is invoked. It carries the number of arguments or passed parameters that were on the command line when it was executed. The following listing is something I used to use on most of my early scripts. A better solution can be found in the section on functions (See -).

Example syntax checking

```
if [ $# -ne 2 ]
then
    echo "Error in $0 - Invalid Argument Count"
    echo "Syntax:  $0 input_file output_file"
    exit
fi
```

This segment of code can be placed at the start of any script to check the user's usage of the script. The `$#` parameter is compared to the constant 2 (in this case 2 arguments was the correct number of parameters for the script). If there is a match, nothing happens and the script proceeds. If there is a mismatch, then the two lines are echoed out to the screen and the script exits promptly. Notice the messages that I have output. The first one says there is an error in this script (you might call a sub-script, so it's as well to put the `$0` name in here - the currently executing one) and you have supplied the wrong number of arguments. The next line tells the user what to do about it - here is the correct syntax. Most users can cope with this and it is only 6 lines of code. Don't forget to revise the argument names and the count constant when you use this code in one of your own scripts.

Protect Yourself:

You think that's all there is to it? Come on, users are much more clever! How about one of these tests next? Could save you time later explaining why the user's latest work has gone missing.

Example file exists

```
if [ ! -f $infile ]
then
    echo "Input file [$infile] not found - Aborting"
    exit
fi
```

Note the use of the **test not** flag (!) in front of the **-f** (file exists) flag. Or how about:

Example verify

```
if [ -f $outfile ]
then
    echo "Output file [$outfile] already exists"
    /usr/5bin/echo "Okay to overwrite? ( y/n ) : \n"
    read answer
    if [ "$answer" = "n" ] || [ "$answer" = "N" ]
    then
        echo "Aborting"
        exit
    fi
fi
```

Well, it might work. But there is a failing with this second example. The test of the user answer tests for a "NO!" response. If the user misses the key, it will carry on. Far better to test for a "YES!" response which if missed just fails safe. Revise the test line to:

```
if [ "$answer" != "y" ] && [ "$answer" != "Y" ]
```

Note the use of the logical *and* (&&) instead of the logical *or* (|) here.

Reading Keyboard Input:

There is also a second fault in Example but in this case it is less relevant as the script is about to exit anyway. Whenever you use the **read** statement to get some user interaction into the script, remember to tell the user that the input has been accepted. This is most commonly done by simply putting a blank echo on the next line thus:

```
read $answer
echo ""
```

This guarantees the user sees the cursor move after (s)he hits the return key. Try not to overuse the **clear** command, as users tend to get upset when the text they were just about to pick up with the mouse gently scrolls off the top of the screen. I also try to indicate something is happening when the processing time is going to be a bit lengthy. A simple "please wait" can be useful, or if you are running a loop which takes a few seconds to complete each cycle, try **echoing** out a dot list using something like:

a zero. Well the loop counter *\$more* can do that when it has been decremented enough, so we can do away with the test - another saving. Inside the loop, the counter is decremented and the next **echo** statement nimbly backspaces over the last 12 characters of the original message line, then **echos** out the new 12 characters over the top. On leaving the loop, the **echo** is terminated with a *blank* string, which provides the final *newline*.

Where's My Print Gone?:

Is a common cry from users when they haven't got a clue where their default printer is located. You can be a real help here by setting each users set-up files to include a default printer environment variable and make sure all your printing scripts know how to use it. In the users `.cshrc` or `.profile` (whichever they use), put a line like:

```
setenv PRINTER hplj2          (.cshrc)
or...
```

```
export PRINTER=hplj2         (.profile)
```

Then in any script you can use this variable as the destination printer for the users output. Just use something like:

```
lpr -P$PRINTER $filename
```

Temporary File Generation:

Of course, the same thing applies to temporary files too. Sometimes you can get away with storing a list in a variable, sometimes you can't. When that happens you probably create a temporary file. And where do you create this file exactly? Well hopefully, you will have the good sence to put it in the `/tmp` directory where it will not clog up the users home directory. Do you remember to delete them after use too? I do hope your not one of those who rely on a system re-boot to empty out the `/tmp` directory. There is a simple solution to these problems and as always, it requires a bit of thought to work out a consistant but flexible scheme. This is what I worked out.

Filenames

1. Always locate the temporary files in the `/tmp` directory
2. Always start the name with something unique for the scripts I create
3. Always include the calling script name in the file
4. Always use an extension that indicates contents
5. Allow several users to call the same script simultaneously
6. Allow a script to create several such files without conflict

Number 1 is easy, I do this all the time. For number 2, I chose the characters **db_** as most of my scripts are related to database work. Number 3 is easy, the `$0` parameter is always available to lend a hand here. For number 4, I have settled on a three character extension (most users have seen DOS) and I try to stick to a firm scheme of mnemonics to indicate contents as follows:

tmp	=	temporary file - content could be anything
lst	=	file contains a list of something
dat	=	file contains data
mnu	=	file contains a menu
sql	=	file contains SQL_Plus* statements
txt	=	file contains text

R2's Shell Tutorial

log	=	file is a log of process or user activity
ftp	=	file contains ftp commands

For number 5, there is another useful script variable called **\$\$** which is the *process id*. No two users or shells or tasks can have the same *process id* concurrently. Lastly, you need to be able to cope with an additional character or two in the filename passed as an argument. So put the lot together and what have we got? We have this:

```
a_temp_file="/tmp/db_${0}$_$$_$arg.tmp"
rm -f $a_temp_file
touch $a_temp_file
```

The **rm** and **touch** commands make sure we always start in a known condition, the file is there but empty. See the section on functions () to see how this is packaged up into a single call such as:

```
tmp0=`s_tmp 0`
```

Notice with this scheme, how easy it is to clean up at the end of a script. You just need to remove the files which contain the script name plus the **\$\$**. And if two people are using the script concurrently, the **\$\$** will guarantee that they both have unique filenames.

Using Files

When creating scripts, it is often a requirement to generate some form of file for storing some temporary information or even something more permanent, like a log file. Sometimes you may even want to create another script for later execution. There are a number of ways to do this, some of which we have already met.

Simple File Creation:

There are two simple ways to create another file, one uses the **cat** command in conjunction with the *redirect* symbol, the other way is to use the **echo** command in conjunction with the *redirect* symbol. Example is a good example of the **cat** method in the *Pipes and Redirects* section. This example only contains literal text however. It is more appropriate to see something like Example below, which shows a variable being used in the source data block.

Example cat and variables

```
cat >> $sql0 <<-EOA
    SET ECHO OFF
    SET FEEDBACK OFF
    SET HEADING OFF
    SELECT my_package.my_function($column)
           FROM v\ $database
           WHERE name LIKE '%&1%';
EXIT
EOA
sqlplus -s $uid/$password@database @$sql0 $sql_arg_1 > $log0
```

The file created has its name stored in the variable *\$sql0* and as we can see the block between the **EOA** flags is the data that goes into the file. The data block is actually a segment of SQL*Plus statements, as indicated by the filename variable. As is common with SQL*Plus code, the key words are picked out in ALL CAPS, with objects (tables, procedures, columns, etc.) all in lower case. The **SELECT** line contains a reference to a called, packaged, PL/SQL function which has a column name as an argument. Here the column name is held in a variable called *\$column* and this will be substituted at script *run-time* by the real value.

There are some unfortunate consequences of generating SQL*Plus statements from within a shell script which you have to be aware of. Firstly, don't forget to put the **EXIT** statement at the end of the block or you will end up with a script that stays in SQL*Plus forever⁵. Secondly, don't forget to put the *semi-colons* (;) at the end of every SQL statement⁶, or each statement will overwrite the previous one or just create one long unprocessable mess. Thirdly, some internal database tables may contain the dollar symbol, which is *special* to the shell, so escape them with the back-slash (\) as shown on the **FROM** line.

On the **WHERE** line there is a reference to a SQL*Plus positional parameter '&1' which will pick up its value from the variable *\$sql_arg_1* at *run-time* as shown in the last line, just after the end of block flag. Did I say this was a simple example? Well, at least you don't have to worry about quoting when

⁵ type EXIT at the keyboard if this happens

⁶ not required for SQL*Plus statements like SET, EXIT, etc

using this method. All quotes find their way to the destination file unscathed. Now to do the same thing using **echo** instead of **cat**, see Example below.

Example simple echo

```
echo "SET ECHO OFF" >> $sql0
echo "SET FEEDBACK OFF" >> $sql0
echo "SET HEADING OFF" >> $sql0
echo "SELECT my_package.my_function($column)" >> $sql0
echo " FROM v\$database" >> $sql0
echo " WHERE name LIKE '%&1%';" >> $sql0
echo "EXIT" >> $sql0
sqlplus -s $uid/$password@database @$sql0 $sql_arg_1 > $log0
```

Complex File Creation:

So what's the point of all this extra typing? Well for one thing it allows you to put special bits of code into the block which will only be used at certain times, by hiding them in complex command groups. The Example shows how this is done below.

Example complex echo forms

```
echo "SET ECHO OFF" >> $sql0
echo "SET FEEDBACK OFF" >> $sql0
echo "SET HEADING OFF" >> $sql0
echo "SELECT my_package.my_function($column)" >> $sql0
echo " FROM v\$database" >> $sql0
if [ "$db_type" = "m" ]
then
    echo " WHERE name = '$db_name';" >> $sql0
else
    echo " WHERE name LIKE '%&1%';" >> $sql0
fi
echo "EXIT" >> $sql0
sqlplus -s $uid/$password@database @$sql0 $sql_arg_1 > $log0
```

This is basically the same block except the **WHERE** clause has been hidden inside an **if** statement. Now, depending on the Database Type in the *\$db_type* variable, the **WHERE** clause can take one of two forms. Conveniently, the additional argument which is not required by SQL*Plus in the first form, is ignored at execution time, even though it is still available on the last line. This is common with all scripts, arguments are only *used* if they are *referenced* from within the script.

So there you have the first two ways of creating another file from a script. The version using **cat** can only cope with a single output form, the version using **echo** can output a multitude of forms depending on the complex command forms you use. The choice is yours. There are, however, other ways to create output files. You can use direct generation as in Example to create a list of files. Or the indirect method shown in Example where lines are built inside a loop construct and then appended to the file to create a menu file. Or in Example where a list of words is sorted into alphabetic order, duplicates are removed, then the rest stored in a file.

Example list

```
ls -l *.log > $lst0
```

Example counted list

```
count=1
for file in `ls -l *.log`
do
    echo "$count: $file" >> $mnu0
    count=`expr $count + 1`
done
```

Example sorted list

```
echo $@ | tr ' '\n' | sort -u > $lst0
```

File Reading:

There are also the **head** and **tail** commands which are useful when reading files as opposed to writing. We have already seen one example of their use in a previous section. Here are a few more. Example **reverse** reverses the order of lines in a file and adds a word count to the end of each line. Note the use of **expr** to change the type of the counter variable *\$file_length* before the **while** command sees it.

Example reverse list and count words

```
file_length=`wc -l $input_file | cut -c1-8`           # Count Lines
file_length=`expr $file_length + 0`                 # Change Type
while $file_length                                  # Start Loop
do
    line=`head -$file_length $input_file | tail -1` # Get Line
    words=`s_count_args $line`                       # Count Words
    echo "$line = $words words" >> $output_file      # Write Line
    file_length=`expr $file_length - 1`             # Decrement
done
```

Don't forget that the **tail** command has a few more options up its sleeve than most users remember. As well as the *minus*, there is the *plus*. This acts a bit like an inverse **head** command in that the command **tail +10 -10** will give you lines 10 through to 10 from the end - in a 33 line file, that is from 10 through 23. There are also the **l**, **b**, and **c** options (lines, blocks, characters) which affect the *element* being counted by the *plus* and *minus* numbers. Then there is **r** which reverses the order of the lines returned. Lastly there is **f** which is often used as part of a monitor script. This option does not terminate, but just keeps looking at the end of the named file waiting for something else to be written. Here is an example of the **tail -f** option in use in Example below, which is a small section of a much longer script

Example tail -f option

```
#-----
# Launch FTP in background and collect return code on completion
#-----
( ftp -nv < $ftp_commands > $ftp_log 2>&1 ; echo $? > $ftp_return_code ) &
job_1_number=$!
tail -f $ftp_log | grep $search > $done_log &
job_2_number=$!

#-----
# Check done_log until complete
#-----
while [ `grep -c $search $done_log` = 0 ]
do
    sleep 1
done

#-----
# Tidy Up the jobs if hung and log result
#-----
if [ "`cat $ftp_return_code`" = "" ]
then
    kill -9 $job_1_number $job_2_number
    echo "FTP Complete at `s_date`" >> $process_log
else
    echo "FTP returned a [ `cat $ftp_return_code` ]" >> $process_log
    echo "Aborting $0 at `s_timestamp`"
    echo "=====“
    cat $process_log
    echo "=====“
    exit
fi
```

Note the use of **#!** to return the background job numbers for later termination. Also see how the two commands on the first line have been submitted to a *subshell* in the *background* and yet the return code from inside the subshell, being redirected to a file, is available for testing in the parent shell. The **tail -f** command will stay looking at the end of the log file for ever if left, hence the need for the tidy up later, but the **grep \$search** only passes a known *string* on to the completion file *\$done_log*. This can be checked in a loop with **sleeps** until something appears. Notice how flexible the **test** command is by allowing the output of a **cat** command to be used as part of a string comparison. Of course in a real script you might want to code in some loop counters to stop a hung FTP process from hanging the script. Say, count to 100 loops max or something. Its your choice how long to wait.

Pipes, Lists and Redirection

Now, as promised, a closer look at the pipe, list, and redirection characters and their functionality.

Pipe Dreams:

Pipes are a UNIX feature which allows you to connect several commands together in one line and pass data from one to the next much like a chain of firemen sending buckets of water down a line. The data in the bucket is processed by each command and then passed on to the next command without ever coming up for air. This happens because of two things:

1. Most UNIX commands get input from **stdin** and pass output to **stdout**
2. The pipe symbol (`|`) directs UNIX to connect **stdout** from the first command to the **stdin** of the second command.

So that sounds simple. How does it work in practice and what does a command pipe look like. The Example shows several pipes made up of groups of commonly piped commands. You will see examples of these syntax structures in most scripts somewhere in the code.

Example pipes

```
line_count=`wc -l $filename | cut -c1-8`  
process_id=`ps -ef | grep $process | grep -v grep | cut -f1 -d\ `br/>upper_case=`echo $lower_case | tr '[a-z]' '[A-Z]'`
```

In all cases the pipeline has been used to set a variable to the value returned by the last command in the pipe. In the first example, the `wc -l` command counts the number of lines in the filename contained in the variable `$filename`. This text string is then piped to the `cut` command which snips off the first 8 characters and passes them on to stdout, hence setting the variable `line_count`.

In the second example, we are searching for the `process_id` or PID of an existing command running somewhere on the system. The `ps -ef` command lists the whole process table from the machine. Piping this through to the `grep` command will filter out everything except any line containing our wanted `process` string. This will not return one line however, as the `grep` command itself also has the `process` string on its command line. So by passing the data through a second `grep -v grep` command, any lines with the word `grep` on are also filtered out. We now have just the one line we need and the last thing is to get the PID from the line. As luck would have it, the PID is the first thing on the line, so piping through a version of `cut` using the field option, we finally get the PID we are looking for. Note the field option delimiter character is an escaped tab character here. Always test the blank characters that UNIX commands return, they are not always what you would think they are.

You should be able to work out the last example yourself based on just the variable names alone. Note the shorthand version of the complete alphabet we used for the `tr` in Example .

Lists:

Lists look similar to pipes except the pipe symbol `|` is replaced by one of the following *list* symbols between each command in the list: `;`, `&`, `&&`, or `| |`, and optionally terminated by `;` or `&`. The semi-colon character is interpreted by UNIX to be a Carriage Return, so a list of commands separated by semi-colons behaves in much the same way as a list of commands on separate lines would behave (hence the name). The difference is that all the list types can be executed in the current shell or in a

sub-shell by utilising a slightly different syntax and the output from the completed list can be redirected (see below) or piped (see above). The two syntax forms for shell locations of lists are shown in and below.

Current Shell:

```
{ command; command; command; }
```

Sub-Shell:

```
( command; command; command; )
```

The other list symbols change the way the list is processed and they have the following meanings:

1. **&** Asynchronously executes the preceding pipeline (as a background task)
2. **&&** Execute only if preceding command or pipe terminated with zero exit status
3. **||** Execute only if preceding command or pipe terminated with non-zero exit status

Redirects:

The matter of redirecting input and output follows a similar principle to that of piping. The significant differences are that redirects work with files, not commands, and there is a limit to how many you can put on one line - depending on the open file descriptors. Whereas the pipe connects one commands output to the next commands input, a redirect tells a command to put its output into a file or collect its input from a file. There is also a difference in the syntax due to the way UNIX executes its commands.

Normally UNIX will try and find an executable file somewhere on the command path (`$PATH` variable) which matches the first word on the current command line. So the first command in a pipe gets found, then executed, and data is piped on to the next command. Because redirecting is for files and not commands, a redirect file cannot be placed ahead of the command on the line. Take another look at the last pipe in the above example. Rewriting this command as a redirect would give the following:

```
tr '[a-z]' '[A-Z]' < $in_file > $out_file
```

Now you can see the difference. The command must come first, the *in_file* is directed in by the **less_than** sign (`<`) and the *out_file* is pointed at by the **greater_than** sign (`>`). The file descriptor in the *in_file* can include a *wild card* to select a number of files. However, the *out_file* must be unique. Just remember the *in_file* points its arrow at the command, while the *out_file* gets pointed at.

The redirect arrows can also be doubled up as in the next example. Here the output from the `cat` command is a file as before. The **double greater_than** (`>>`) directs the output to be appended to the file, if it already exists. If the file does not exist, it is created. The single arrow form, as used above, would always create a new file if there was none there, or overwrite an existing file.

On the input side the double arrow has a slightly different meaning. Here, where the single arrow gets the input from a file, the double arrow gets its input from the shell file that is currently executing. You may be wondering how the shell can tell where the end of this input is and where the continuing shell script restarts. Well, that is the reason for the word following the **double less_than** sign. The word is a marker that the shell will look out for when reading the input stream. When the word shows up, input will stop and the script will continue.

Example redirected cat

```
cat >> $out_file << EOF
first line of data
second line of data
more data
the end of the data
EOF
```

In this case I have used the flag word EOF to indicate the End Of File. However, any word will be acceptable as long as it is unique in the script file. What I generally do is use EOA, EOB, EOC, etc., if I create several files within a script. The capitalisation is not important, but it does make the flags easy to match up as a pair when reading the script.

There is one more thing you can do with this redirected input from a script file. Look at this next example and you will see a minus sign between the double arrows and the flag name:

```
cat >> $out_file <<-EOF
```

This instructs the shell to remove leading tab characters from all lines in the input stream including the matched flag word. This handy trick allows you to use a code indent which makes reading much easier as in the following example which is a copy of the previous code segment, but in this new easier to read format.

Example indented cat

```
cat >> $out_file <<-EOF
    first line of data
    second line of data
    more data
    the end of the data
EOF
```

Now it looks more like a code block and the flag word stands out too. The section which is indented is easily understood to be the contents of the created file. This feature is not available in the C Shell.

In addition, the redirect arrows can actually redirect input and output, to and from the **stdio** files, known by their descriptor names (0 and 1). These are the default input and output files for UNIX, usually connected to keyboard (0), display (1) and errors (2). Thus the syntax:

<&digit

uses the file associated with file descriptor *digit* as the standard input. The same goes for standard output if you reverse the arrow. You can also associate one file with another as in this example:

```
ls -l $directory/*.log > $out_file 2>&1
```

Here we see an **ls** command outputting to *out_file*. At the end however, is another redirect which is indicating that **stderr** (file descriptor 2) should also be sent into the **stdout** (file descriptor 1), which in this case is our *out_file*. To say the same thing in C Shell the syntax looks simpler, but is harder to read because the descriptor numbers are missing:

```
ls -l $directory/*.log >& $out_file
```

Don't forget, you can use this mechanism to create files with any content you like generated from any other combination of commands. Here is an example of a menu file listing the files in a directory. This can then be displayed to the screen and the users choice selected quite simply.

Example simple menu

```
count=0
for file in `ls -l $source_directory`
do
    count=`expr $count + 1`
    echo "$count:      $file" >> $menu_file
done
echo "Please select a number from this menu"
cat $menu_file
read $choice
echo "Thanks"
filename=`grep $choice $menu_file | cut -f2 -d:`
echo "You chose [$filename]"
```

This example is very simplistic however and will not cope with filenames that contain digits or filename lists longer than 9 lines. Both of these conditions could lead to the **grep** returning more than one line which is an error condition (See - for a better solution to these problems).

Functions

The Function or Subroutine is one of the most useful features of the shell programming languages. The man pages generally call them Functions which is what I have done here but this can lead to misunderstandings. By definition a function must return one thing but sometimes these shell functions return more than one by using **stdout** as well as the *return value*. The things you need to remember are best listed out as short reminders which can then be used as a quick reference guide later. This then, is my list of useful things to know about shell subroutines or functions.

1. A function encapsulates a small piece of code that will be used often.
2. You only need to define it once.
3. A function will execute faster than the equivalent code in the script.
4. Parameters are passed into the function via numbered arguments just like scripts.
5. The *special* shell variables are also set up when a function is called (*\$#*, *\$@*, etc.) but in this case they relate to the called function not the calling script. Their scope is local to the function.
6. Output can be in the form of *stdout* or a *return code* value or *both*. You cannot stop the function returning a value via its *return code*, but you can exercise control over the *return code*. If you choose not to, then the function will return the value of the last executed sub command from within the function.
7. Variables used and set inside a function *can* be visible from the calling script and
8. Variables set within the environment or calling script *can* be visible in the function. This relies on the use of the **export** command in both directions.
9. Functions can be nested to any depth - well I haven't found a limit.
10. No amount of nesting will ever disguise the *\$0* parameter. The *\$0* will never be set to the function name, it is only ever set to the name of the calling script.
11. With care and **shift**, any number of parameters can be handled.

If you think of any more, please add them to this list yourself. The rest of this section contains example listings of some of my own functions, which I have created over the years to support my normal script programming activities. They are grouped into areas (strings, files, menus, etc.) and stored in separate files. A master profile exists for the accounts which execute the scripts and this profile adds in all the function files at run time. If this sounds slow and complicated, it isn't. In practice, the separate files are easier to handle, edit and distribute around a networked system. The additional start-up time for scripts which have to read 200 functions is less than 0.5 seconds. This is more than compensated for by the shorter scripts which result, the easier maintenance load, the faster script creation time, etc. Well, that's enough of that, lets get on with some examples. Note the use of the function name itself as part of the banner line for each function. The (c) flag is there for another script called **fun** which searches out lines containing the flag in all the function definition files. In this way it is easy to get an up-to-date list of all the functions currently available.

Simple Date and Time functions:

```
#=====
===
s_timestamp()      # (c) RHReepe. Returns string (YYYY-Mon-DD@HH:MM:SS)
#=====
===
{
```

R2's Shell Tutorial

```
        date "+%Y-%h-%d@%H:%M:%S"
}

#####
===
s_now()      # (c) RHReepe. Returns string (YYYYMMDDHHMMSS)
#####
===
{
        date +%Y%m%d%H%M%S
}

#####
===
s_time()     # (c) RHReepe. Returns string (HHMMSS)
#####
===
{
        date +%H%M%S
}

#####
===
s_date()     # (c) RHReepe. Returns string (YYYYMMDD)
#####
===
{
        date +%Y%m%d
}
}
```

They may look simple, but in each case the character count of the function name is about half that of the function content. That in itself is enough justification. It also means you never need to remember the combination of letters (and cases) required to generate your date or time string ever again. You might also notice the use of capital “Y” in all the date oriented functions. If you check out the man pages for the **date** command, you probably won’t find any references to “Y” at all. That’s because the plus sign (+) actually tells **date** to call another function called **strftime** which understands a lot more formatting arguments than **date** does. Check out the man pages for **strftime** for yourself.

```
#####
===
s_month_length() # (c) RHReepe. Returns number of days in MONTH (INT)
#####
===
# Arg_1 = MONTH_NUMBER
{
        if [ $1 -lt 1 ] || [ $1 -gt 12 ]
        then
                echo "$0:s_month_length(): [$1] is not between 1 and 12"
                exit
        fi
        lengths="312831303130313130313031"
        cut2=`expr $1 + $1`
        cut1=`expr $cut2 - 1`
        echo $length | cut -c$cut1-$cut2
}
}
```

The **s_month_length()** function is quite simple but watch out for leap years as there is no account taken of Feb 29th. The fix is actually quite easy too - just divide the year by 4, then multiply by 4, and see if you get the same number you started with. You will need to use the **bc** command for

division and multiplication. If it is the same number then Feb may have 29 days (unless its a century, or maybe it still does, if it's a millenium!).

Now here are two slightly more complex examples, `s_interval()` and `s_back_date()`, which get used for checking out superceded archives and timing test functions or scripts. The function `s_interval()` is fully in-stream documented. This is a bit over the top in this example, but you should be able to see how such treatment enhances understanding when reading the functions. In fact the only additional thing I'll say about this function, is that I use it to time new scripts and functions to help me tune out any slow operations. It has two mandatory arguments which can be provided by calling the `s_time()` function at each end of the timed process. The output is in the same format (HHMMSS), don't forget and think that 000120 = 2 minutes, it is 1 minute 20 seconds.

```

=====
s_interval()      # (c) RHReepe. Returns a time difference (HH:MM:SS)
=====
# Arg_1 = start_time (Format - See s_time)
# Arg_2 = stop_time (Format - See s_time)
{
    h1=`echo $1 | cut -c1-2`      # Get Start Hour
    m1=`echo $1 | cut -c3-4`      # Get Start Minute
    s1=`echo $1 | cut -c5-6`      # Get Start Second
    h2=`echo $2 | cut -c1-2`      # Get Stop Hour
    m2=`echo $2 | cut -c3-4`      # Get Stop Minute
    s2=`echo $2 | cut -c5-6`      # Get Stop Second
    s3=`expr $s2 - $s1`          # Calculate Second Difference
    if [ $s3 -lt 0 ]             # Test for Negative Seconds
    then
        s3=`expr $s3 + 60`        # If yes - add one minute...
        m1=`expr $m1 + 1`         # ... and to subtractor
    fi
    m3=`expr $m2 - $m1`          # Calculate Minute Difference
    if [ $m3 -lt 0 ]             # Test for Negative Minutes
    then
        m3=`expr $m3 + 60`        # If yes - add one hour...
        h1=`expr $h1 + 1`         # ... and to subtractor
    fi
    h3=`expr $h2 - $h1`          # Calculate Hour Difference
    if [ $h3 -lt 0 ]             # Test for Negative Hours
    then
        h3=`expr $h3 + 24`        # If yes - add one day
    fi
    for number in $h3 $m3 $s3    # Loop through numbers...
    do
        if [ $number -lt 10 ]    # If number is single digit...
        then
            /usr/5bin/echo "0$number\c" # ... add leading zero
        else
            /usr/5bin/echo "$number\c" # ... else - don't
        fi
    done
    echo ""                       # Terminate the string
}

```

Lastly the function `s_back_date()` which returns a date, an integer number of days before today. This is used to check when to retire an old archive for instance. Internal documentation has been limited to block headers this time to highlight the differences in presentation. Note the use of the optional parameter [DAYS:-1] with a default of one-day, so calling the function with no arguments will return you yesterday's date. The date string padding in the final block shows an example of un-

typed data concatenation, as the predicate tests the numeric value of the strings, then prepends a string zero if it is too short. There is no attempt to cope with back dates that flow over a year boundary. This could be added for completeness if required by replicating the outer **if-else-fi** structure under the **date_m** decrement statement.

```
#####  
====  
s_back_date      # (c) RHReepe. Returns a date string DAYS back  
#####  
====  
# Arg_1 = [DAYS:-1]  
{  
    days=${1:-1}  
    date_d=`date +%d`  
    date_m=`date +%m`  
    date_y=`date +%Y`  
    #-----  
    # Days Back Size Test  
    #-----  
    if [ $days -lt $date_d ]  
    then  
        date_d=`expr $date_d - $days`  
    else  
        days=`expr $days - $date_d`  
        date_m=`expr $date_m - 1`  
        month_length=`s_month_length $date_m`  
        while [ $days -gt $month_length ]  
        do  
            days=`expr $days - month_length`  
            date_m=`expr $date_m - 1`  
            month_length=`s_month_length $date_m`  
        done  
        date_d=`expr $month_length - $days`  
    fi  
  
    #-----  
    # Date String Padding  
    #-----  
    if [ $date_d -lt 10 ]  
    then  
        date_d="0"$date_d  
    fi  
    if [ $date_m -lt 10 ]  
    then  
        date_m="0"$date_m  
    fi  
    echo $date_y $date_m $date_d  
}
```

Simple File Set-up Functions:

This next section deals with functions related to file set-up. This includes temporary files, status flags, log-files and many other related functions. Some of these may seem simple and rudimentary, but that is less important than easing the readability of the calling scripts and reducing the need to re-code the same thing several times.

```
#####  
====
```

R2's Shell Tutorial

```
s_prog()          # (c) RHReepe. Returns the calling script name
#=====
===
{
    basename $0
}

#=====
===
s_file()          # (c) RHReepe. Creates a file name
#=====
===
# Arg_1 = file extension
# Arg_2 = unique identifier
{
    temp_file="/tmp/db_`s_prog`_$$_$2.$1"
    rm -f $temp_file
    touch $temp_file
    echo $temp_file
}

#=====
===
s_rmfile() # (c) RHReepe. Removes file names
#=====
===
# Arg_1 = [file extension]
# Arg_2 = [unique identifier]
{
    rm -f /tmp/db_`s_prog`_$$_${2:-""}*.${1:-""}*
}

#=====
===
s_tmp()          # (c) RHReepe. Creates a TMP filename
#=====
===
# Arg_1 [unique identifier]
{
    s_file tmp ${1:-$$}
}

#=====
===
s_rmtmp()        # (c) RHReepe. Removes a TMP filename
#=====
===
# Arg_1 = [unique identifier]
{
    s_rmfile tmp ${1:-$$}
}

#=====
===
s_sql()          # (c) RHReepe. Creates a SQL filename
#=====
===
# Arg_1 [unique identifier]
{
    s_file sql ${1:-$$}
}

#=====
===
```

R2's Shell Tutorial

```
s_rmssql()          # (c) RHReepe. Removes a SQL filename
#=====
===
# Arg_1 = [unique identifier]
{
    s_rmfile sql ${1:-$$}
}

#=====
===
s_log()            # (c) RHReepe. Creates a LOG filename
#=====
===
# Arg_1 [unique identifier]
{
    s_file log ${1:-$$}
}

#=====
===
s_rmlog()          # (c) RHReepe. Removes a LOG filename
#=====
===
# Arg_1 = [unique identifier]
{
    s_rmfile log ${1:-$$}
}

#=====
===
s_cleanup()        # (c) RHReepe. Cleans up ALL files for this script
#=====
===
# Arg_1 = [unique identifier]
{
    s_rmfile ??? ${1:-""}
}

#=====
===
s_truncate()       # (c) RHReepe. Truncates extension from a filename
#=====
===
# Arg_1 - FILENAME
{
    dot=`s_in_string $1 "."`
    dot=`expr $dot - 1`
    echo $1 | cut -c1-$dot
}

#=====
===
s_running()        # (c) RHReepe. Checks if script is running
#=====
===
{
    if [ -f /tmp/db_`s_prog`_running.lock ]
    then
        echo "Script [$0] is already running!"
        exit -1
    else
        touch /tmp/db_`s_prog`_running.lock
    fi
}
```

```
#####  
====  
s_stopping()      # (c) RHReepe. Removes running lock file  
#####  
====  
{  
    rm -f /tmp/db_`s_prog`_running.lock  
}
```

There are probably some surprises here, so I'll try and explain them starting with **s_prog()**. You may wonder why I have used **basename** in this function when the name of the calling script is blatantly obvious. Well just try and run a script from the **crontab** and display the script name with **echo \$0** in the script and see what happens. If you can't wait that long, I'll tell you. What you get is the scripts full pathname (`/co/host/u/user_accounts/me/scripts/better_scripts/really_brill_stuff/this_one`) which is really going to look professional when it finds its way into a logfile! Note also that each of the subsequent functions makes a call to **s_prog()** internally, another bunch of characters you won't have to type in.

The next two functions I class as paired (**s_file()**, **s_rmfile()**) as one creates something and the other removes it. In this case we are talking about general files, the function **s_file()** pre-deletes the file in case one already exists, then creates an empty file using **touch**, then last of all the filename is returned via **stdout** which can be used to set a variable. The function has two arguments which are both mandatory. The first is the extension to use at the end of the filename, and second is a unique identifier, required if you want to create more than one file of this type from within the same script. The next function **s_rmfile()** removes the files created by the first function, as long as you provide the optional arguments - extension and unique identifier. If you forget to provide the arguments, it will remove all the files created during the running of the calling script. This may sound a bit dangerous, but this is where the nesting of functions starts to show real benefits.

Take a look at the next six functions **s_tmp()**, **s_rmtmp()**, **s_sql()**, **s_rmsql()**, **s_log()** and **s_rmlog()** all of which make calls on **s_file()** and **s_rmfile()**. The three create functions all have an optional single argument - unique identifier - which allows you to have some control over the actual filename. If you choose not to exercise this right, then the function will select the PID of the parent script and use that instead. Note that all of these create functions call the **s_file()** function with a fixed extension based on their own names. The same is true of the three remove functions also, which makes it particularly easy to use these higher level functions within a script. Notice also that the function **s_cleanup()** is nothing more than a call to **s_rmfile()** with one argument. Inspecting the function **s_rmfile()** will reveal what is happening. Its optional argument is set to NULL or the empty string ("") when no argument is supplied. Hence the expanded filename includes all members of the group `/tmp/db_[calling_script_name]_[PID]_*.???` and so cleans up all files created by this run of the script. By copying the two tmp functions several times and revising the three character extensions for the filenames, you can create functions to manage `.log`, `.sql`, `.tmp`, `.dat`, `.bin`, `.lst`, etc.

Following on is the function **s_truncate()** which is used to remove the extension from some filenames. The function makes no changes to the file itself, you will still have to move or copy the file yourself if you want it done. The function just returns the name minus its extension. It works by searching for the dot in the name and returning what is in front of the dot. A word of warning, beware filenames with more than one dot, this function will cut at the first dot found. If you really wanted to check for *three* character extensions only and allow multiple dots in the filenames, get the length of the name and set up the cut to always remove 4 characters from the end, but this will lock you in to three character extensions only. You may spot a more elegant way to achieve this after studying the next group of string functions.

The next two functions are again paired but these two relate to script instance control. When running some scripts, it is important to know that there is only one instance (it is only running once on the CPU). The `s_running()` function checks for a running flag file and if not found creates one, thus setting the flag. If there is an existing flag file, then the function exits. Usefully, the `exit` statement also terminates its calling script as well as the processing function, aborting the script if an instance is already running. The `s_stopping()` function is provided to clean up the flag file just as the script terminates. These two functions are placed at the head and tail of a script to protect against multiple instances.

Simple String Functions:

This next batch all relate to string functions and text handling. Starting simply with join strings and case changers, we move on through padding into character and string searches.

```
#####
===
s_join()          # (c)RHReepe. Creates single string from several
#####
===
# Arg_n = Input Strings
{
    echo "$@" | tr '""_'
}

#####
===
s_upper()        # (c) RHReepe. Returns the input string in upper case
#####
===
{
    echo $@ | tr '[a-z]' '[A-Z]'
}

#####
===
s_lower()        # (c) RHReepe. Returns the input string in lower case
#####
===
{
    echo $@ | tr '[A-Z]' '[a-z]'
}

#####
===
s_count_args()   # (c)RHReepe. Returns count of input ARG_LIST
#####
===
# Arg_n = ARG_LIST
{
    echo $#
}

#####
===
s_count_lines()  # (c)RHReepe. Returns count of lines in FILENAME
#####
===
# Arg_1 = FILENAME
{
    line_count=`wc -l $1 | cut -c1-8`
}
```

R2's Shell Tutorial

```

        expr $line_count + 0
    }

#####
===
s_length() # (c)RHReepe. Returns length of string
#####
===
# Arg_1 = string
{
    length=`echo "$@" | wc -c | cut -c1-8`
    length=`expr $length -1`
}

#####
===
s_right_pad() # (c)RHReepe. Right pads a string to width
#####
===
# Arg_1 = string
# Arg_2 = width
{
    string="$1"
    echo "$string" | cut -c1-$2
}

#####
===
s_left_pad() # (c) RHReepe. Left pads a string to width
#####
===
# Arg_1 = string
# Arg_2 = width
{
    string="          $1"
    length=`s_length "$string"`
    echo "$string" | cut -c`expr $length - $2`-$length
}

#####
===
s_find_char_all() # (c) RHReepe. Finds all positions of CHAR in LIST
#####
===
# Arg_1 = CHAR
# Arg_n = LIST
{
    echo "$@" | cut -f2- -d\ | sed -e 's/./&\
/g' | grep -n $1 | cut -f1 -d:
}

#####
===
s_find_char() # (c) RHReepe. Finds first position of CHAR in LIST
#####
===
# Arg_1 = CHAR
# Arg_n = LIST
{
    s_find_word_all "$@" | head -1
}

```

```
#####  
====  
s_get_char()      # (c) RHReepe. Returns the CHAR at POSITION in LIST  
#####  
====  
# Arg_1 = POSITION  
# Arg_n = LIST  
{  
    echo "$@" | cut -f2- -d\ | cut -c$1 -d\  
}  
  
#####  
====  
s_find_word_all() # (c) RHReepe. Finds all positions of word in list  
#####  
====  
# Arg_1 = word  
# Arg_n = list  
{  
    echo $@ | cut -f2- -d\ | tr ' '\n' | grep -n $1 | cut -f1 -d:  
}  
  
#####  
====  
s_find_word() # (c) RHReepe. Finds first position of word in list  
#####  
====  
# Arg_1 = word  
# Arg_n = list  
{  
    s_find_word_all $@ | head -1  
}  
  
#####  
====  
s_get_word()      # (c) RHReepe. Returns the word at position in list  
#####  
====  
# Arg_1 = position  
# Arg_n = list  
{  
    echo $@ | cut -f2- -d\ | cut -f$1 -d\  
}
```

This is an interesting little collection. Starting from the top, the **s_join()** I created as a specific bug fix for a piece of mail software. I needed a way of picking up a list of words (used as the mail title) and making one long string. I tried various forms of quoting to encapsulate the string, but the mail software always took the first word as the title and tried to mail to all the other words as addressees. The function has since found other uses and is a good example of the **tr** command which is used to translate characters piped through it from the first quoted string to the second.

Next come two related functions which should be very obvious from their names alone. They simply change the case of the passed string. Note the use of **\$@** which passes all the arguments as one list.

Next is another very simple function **s_count_args()** which finds use in lots of places. Again this makes use of a very simple feature of the shell, the **\$#** parameter, which returns the number of passed arguments. So passing a list of words to the function causes it to return the integer number of words.

The `s_length()` function returns the character length of a string. Again `$@` is used to get all the parameters in a single pass. Here the character count is reduced by one as the `wc -c` command will include the newline (`\n`) at the end of the `echo` by default.

The next two functions are interesting for their unusual approach to a problem. Give any normal programmer a specification for a left pad or right pad function and they will start creating loops that add a character at each pass. In the shell, we can make use of the `cut` command and simple un-typed concatenation of dissimilar variables (the input string could be a number) to create first a long string, and then chop off the wanted length to give a quick result. There is one extra command in the left pad as we need to find where the end is before we can make the right hand cut.

The last six functions in this section are split into two groups. The first deals with searching for characters in strings. The second deals with searching for words in strings. The function `s_find_char_all()` is the base function in the first group. It makes use of the `sed` command in a novel way. The `sed` command is a **Stream EDitor** which has been passed a quoted command string. The command substitutes every character in the input stream to the same character plus a newline. This turns our horizontal input string into a vertical list of characters placed one per line. Now it's easy, the next command is a `grep` to find the character in `$1`, the `-n` option requests `grep` to give the line number as well as the found string. Then along comes `cut` and snips out the line number to be returned.

Of course this will return a list of line numbers if there is more than one match. If you only want to find the first match, then use the `s_find_char()` function which includes the `head -1` statement. In the previous group I said a more elegant solution may present itself, and here it is. Just replace the `head -1` with `tail -1` and you have the location in the string of the last dot instead of the first. This can then be used to trim the filename extension correctly, however long it is.

Lastly, the `s_get_char()` function returns the character found at *position* in the string. Nothing very difficult here, but just remember to put the double quotes around the `$@` symbol or the character counts will be out if some characters are separated by multiple blanks.

The second group of functions consists of `s_find_word_all()`, `s_find_word()` and `s_get-word()`. The first function `s_find_word_all()` actually contains 5 piped commands. Lets lay them out individually:

<code>echo \$@</code>	Echo ALL Arguments
<code>cut -f2-</code>	Forget the first Argument
<code>tr ' ' '\n'</code>	Translate the SPACES to NEWLINES
<code>grep -n \$1</code>	Get the LINE NUMBER where WORD appears
<code>cut -f1 -d:</code>	Cut off the LINE NUMBER and return it

You will notice the similarity to the previous group of functions, except `sed` is not required here. This function also delivers a list of matches if there is more than one. The next function just calls the first one and adds the `head -1` statement as before, thus returning only the first match. The last `s_get_word()` function is just as simple. Notice that the final `cut` statement only has one address for the cut (`cut -f$1`). As both addresses are the same when one item (`f = field`, `c = char`) is cut, the second address is optional. Look for these short cuts and optional syntax forms, the reduction in code size can stack up nicely in larger modules.

Simple Menu Functions:

This next section contains functions related to Menu operations. This was one of the first groups that I created when one of my colleagues requested an easier interface than remembering the command line arguments. Certainly for the rarely used scripts and even for the regular scripts which require the user to select from a list, these are worthwhile functions.

```

=====
===
s_banner()      # (c) RHReepe. Creates a banner for User Menus
=====
===
# Arg_n = BANNER_TEXT
{
    clear
    echo ""
    echo " _____"
    echo ""
    echo "$@"
    echo " _____"
    echo ""
}

=====
===
s_menu_list()   # (c) RHReepe. Creates a Menu List from a Directory
=====
===
# Arg_1 = DIRECTORY
{
    count=0
    for file in $1/*
    do
        count=`expr $count + 1`
        echo "      $count  `basename $file`"
    done
}

=====
===
s_answer()      # (c) RHReepe. Returns a Menu Pick Pointer Number
=====
===
# Arg_1 = PROMPT_STRING
# ARG_2 = [MAX_MENU_PICKS:-NULL]
{
    answer=""
    max=${2:-""}
    while [ "$answer" -lt "1" ] || [ "$answer" -gt "$max" ]
    do
        echo ""
        /usr/5bin/echo "      $1 ( Between 1 and $max ) : \c"
        read answer
        if [ "$answer" = "" ]
        then
            exit
        fi
    done
    return $answer
}

```

```
#####  
====  
s_menu_pick()      # (c) RHReepe. Returns the Filename from the Menu  
#####  
====  
# Arg_1 = DIRECTORY  
# Arg_2 = MENU_POINTER  
{  
    count=0  
    for file in $1/*  
    do  
        count=`expr $count + 1`  
        if [ $count = $2 ]  
        then  
            echo "$file"  
        fi  
    done  
}
```

The first function **s_banner()** is very easy, it just takes a list of words as input and places a nice line above and below all one tab space in. The clear makes sure you have a nice empty screen for your menu.

Next is the function **s_menu_list()** which generates a nice numbered list under the heading. Be careful if the list extends off the screen as it will just scroll away. It also takes longer to generate the list if it gets too long. Use more subdirectories and fewer files per directory to speed up usage.

Next up is the function **s_answer()** which prompts the user to select from the list just generated. The users answer must be between 1 and *max* where *max* is the number of the last entry in the list. A special case is made for a null string (carriage return on a blank line) which is used as the exit from the menu and also terminates the script, in case of user error.

Lastly, we have the function **s_menu_pick()** which returns the filename the user selected from the menu as a text string. The filename includes the complete path as well, so don't add that back on.

By using these four functions together, a very succinct menu system can be generated out of the file system. With careful planning this can be a real boon when organising files or selecting backups and archives to restore etc. I have not yet created a function to generate a menu from a list in a file, but it can only be a matter of time!

Simple Utility Functions:

This next group do not fit in any other category, so I have lumped them all together for now. You have met two already from this group, but these are the final versions.

```
#####  
====  
s_version()        # (c) RHReepe. Returns 1 (cron) or 0 (not cron)  
#####  
====  
# Arg_1 = [SCRIPT_NAME]  
{  
    head -3 ${1:-$0} | grep "(c)" | grep Version | cut -f9 -d\  
}  
  
#####  
====  
s_is_cron()        # (c) RHReepe. Returns 1 (cron) or 0 (not cron)
```

R2's Shell Tutorial

```
#####  
====  
{  
    tty | grep -ci not  
}  
  
#####  
====  
s_syntax()# (c) RHReepe. Checks Script Syntax Usage  
#####  
====  
# Arg_1 = $# (Literally $# in the calling script)  
# Arg_n = REQUIRED_ARGS  
{  
    arg_list=`echo @$@ | cut -f2- -d\  
    if [ "$arg_list" = "" ]  
    then  
        required_args=0  
    else  
        required_args=`s_count_args $arg_list`  
    fi  
    if [ $1 != $required_args ]  
    then  
        echo ""  
        echo "Syntax Error in $0"  
        echo "Usage: $0 $arg_list"  
        echo ""  
        exit  
    else  
        break  
    fi  
}
```

SQL*Net 1.0 Functions

This next group of functions were created to help with distributed database connections. They are all based on the Oracle SQL*Net 1.0 protocol. There is a matching set for SQL*Net 2.1 later in this section. To understand these functions fully, you may need to make reference to some other files listed in the last section - Putting It All Together. I will highlight this again nearer the time.

```
#####  
====  
s_sid_list()      # (c) RHReepe. Returns list of DB SID's on HOST  
#####  
====  
# Arg_1 = [HOSTNAME:-THISHOST]  
{  
    tcpctl stat @${1:-$THISHOST} 2> /dev/null      |\  
    grep "ORACLE SID"                             |\  
    cut -f2 -d:                                    |\  
    tr ',' ''                                       |\  
}  
  
#####  
====  
s_connect_list() # (c) RHReepe. Returns list of Global Connect Strings  
#####  
====  
{  
    connect_strings=""  
    for next_host in $REFERENCE_DBSERVER  
    do
```

R2's Shell Tutorial

```
list_of_database_sids=`s_sid_list $next_host`
for next_sid in $list_of_database_sids
do
    connect_strings="$connect_strings @t:$next_host:$next_sid"
done
done
echo $connect_strings
}

#=====
s_select_db()      # (c) RHReepe. Returns list DB's of TYPE
#=====
# Arg_1 = DATABASE_TYPE
# Arg_2 = [DATABASE_SERVER:-THISHOST]
{
    type_list=""
    wanted_type=`s_lower $1 | cut -c1`
    database_list=`s_sid_list ${2:-$THISHOST}`
    for database in $database_list
    do
        db_type=`echo $database | grep -c "$wanted_type"`
        if [ $db_type -gt 0 ]
        then
            type_list="$type_list $database"
        fi
    done
    echo $type_list
}
```

In the function `s_sid_list()` the stderr file (file descriptor 2) is directed to `/dev/null` to stop complaints of a non-running oracle server inflicting damage on crontab scripts. If a server is down it just returns a null string instead of an error message which cron would forward to your mailbox. As almost every script of crontab job I create has a call to `s_sid_list` in it somewhere, this would be very tiresome. Note also that this is just a single piped command list. The back-slashes hide the newline from the shell and make printout of this function easier

The `s_connect_list()` function returns a complete list of global connect strings which is used for running reports on all databases from a crontab job.

The `s_select_db()` function allows selection of a database by type on a server. I have a few utility scripts which address particular database types only; such as test, development, master, evaluation, production, etc.

Start Up Files and Environment

Now lets just take a timeout from the syntax rules and have a quick look at startup files.

What Are They?:

Start-up Files are files that are read by the script at the start of execution. There is a set of editable default files supplied with UNIX, or you can create you own and force the shell to read these instead, or indeed, as well as. Again there is a difference between C Shell and the rest. The C Shell will read two files by default at login, then one each time the C Shell is invoked. The files are **.login** and **.cshrc** (note the leading *dot*) and both files do not require their execution bit to be set. They are only *read* by the shell and the commands contained within the files are executed within the current shells' environment. It is in these files that environment variables usually get set. If you want to force another file to be read at shell start up, you can use the source command within a script thus:

```
source .my_environment
```

Which will read the file **.my_environment** into the current shell and then execute the commands found just as if they had existed in the executing script.

For the **sh** & **ksh** there is only one default file called **.profile** and it is only run at login time. If you want a start-up file to be run at every execution of these shells, or at the start of any shell script, you must force it with the dot (.) command as shown below:

```
. .my_profile
```

Now this is what I call an abbreviation! It can be very easy to overlook such an innocent looking command, which is one reason I always include it as the first thing after the banner and give the read file such an obvious name. As my default environment is the C Shell, I just use **.profile** for the **sh** & **ksh** start-up files (yes they can and do share a common file - just put the fancy **ksh** specific stuff in a file that **sh** will not see). In all cases, nesting the start-up files is allowed and I am not aware of any nesting limits within reason.

Why Use Them?:

Well, it's nice to know that when you call on previously written utilities from within a script, that they will be found. Paths therefor are a common feature of the start-up files. There are other bits of information that you may initially think are not so useful but are indeed called upon regularly from within scripts. The Example shown below is a chunk from my own **.profile** to show some useful features which most of my other scripts rely on.

Example useful profiles

```
export COMPANY='ACME'
export THISHOST=`uname -n`
export MASTER_DB='mdb_001'
export SCRIPTS=/${COMPANY}/${THISHOST}/unix/cen/scripts
export SQL=${SCRIPTS}/sql
export PRINTER='hplj5`
if [ `tty | grep -ci not` -eq 0 ]
then
```

```
    stty ERASE ^H
    TERM=SUN
fi
```

Here you see the export statement used before the setting of the variable, which has the effect of exporting the variable into the parent environment. On the second line there is a command called **uname** which is executed (by enclosing in *back-quotes*) with an option **-n** to give the name of the executing host machine. The output from this command is passed to the environment variable THISHOST. Then on line 4 you can see this same variable supplying its value in the setting of the SCRIPTS variable. This is then used to help set the SQL variable on the next line. There is a literal string 'ACME' on line 1, and another on lines 3 and 6 - indicated by the single forward facing quotes.

The actual .profile that I use is now in several parts for ease of editing. The parts are:

- .profile Always called
- .profile_functions Always called (now in 6 sub-files)
- .profile_oracle Always called
- .profile_paths Always called
- .profile_cron Called from cron jobs only

The first file is called from within the script using the obvious call:

```
. $HOME/.profile
```

The .profile itself ends with three calls to **.profile_functions**, **.profile_oracle**, and **.profile_paths**. All cron job scripts contain a further call to **.profile_cron** which contains paths specific for the cron jobs and some additional security features that stop the cron scripts from being executed by a user.

A word of warning about start-up files. Don't put commands in here which will output things to the screen (**echo**, **cat**, etc.) unless it is for debugging purposes only. Output from the start-up files will interfere with **rsh**⁷, **ftp** and most other remote access procedures. Output will also get sent to USER mail from a script run from the **crontab** (the UNIX batch queue). Now this is a useful feature because it can be used as an alert mechanism. But overloading the facility with all kinds of garbage messages from start-up files soon gets very annoying and eventually you will start deleting the mail before actually reading it, which is no good at all.

Also, if your one of those for whom the backspace key should deliver a <CTRL-H> character (as I am), then you no doubt have an **stty erase ^H** statement hanging around in your start-up files somewhere. Beware of **tty** and **stty** statements if there's a chance that a cron job will pick them up. The cron processor has no terminal, so references to terminal set-up statements can create disruptive messages in the USER mailbox from cron. The way I cope with this is to have a user defined function in my **.profile_functions** called **s_is_cron()** which contains the line:

```
tty | grep -ci not
```

This function will return the value 0 for a script running with a terminal, and 1 for a cron script running in background. This can then be used in an **if** statement to protect several terminal commands as follows:

Example cron test function

```
if [ `s_is_cron` -eq 0 ]
then
```

⁷ rsh = remote shell, not restricted Bourne shell.

R2's Shell Tutorial

```
    stty erase ^H
    set TERM=SUN
fi
```

This same example is tagged onto the end of Example above but without relying on a function call. For the C Shell (where there is no function definition statement - another good reason not to use it) the equivalent syntax would be as shown in Example below.

Example raw cron test

```
if ( `tty | grep -ci not` != 0 ) then
    stty erase ^H
    setenv TERM SUN
endif
```

Of course, if you are going to use functions, then you have to make sure that you load the definition of the function before you call it for use. Usually these personalised sections are placed last, and a good way to do this is to put it all in a separate file and call it last in the start-up sequence. I have my own personal file which contains my own set of aliases and other short-cuts. They are all in the file **.my_defaults** in my home directory. The first line of this file says source **\$PROJECT/.cshrc** which picks up the departments defaults for this project, the rest of the file adds my own C Shell tweeks. Remembering of course that all the C Shell stuff will be forgotten as soon as a **sh** or **ksh** script starts in its own sub-shell.

Another trick when connecting to a shared account (Several DBA's may have shared access to the dba account for creating databases etc.) is to set your own personalised environment from the following test (shown in C Shell as it would appear in a shared **.cshrc** file):

Example setting user environment

```
#=====
# Set Up Dave's Environment
#=====
set dave_1 = `who am i | grep -c 'dwsun3'`
set dave_2 = `who am i | grep -c '19.123.123.123'`
set dave = `expr $dave_1 + $dave_2`
if ( $dave == 1 ) then
    stty erase ^H
    setenv EDITOR emacs
    setenv DISPLAY 19.123.123.123:0.0
    setenv PRINTER hplj2
endif
```

Depending on how Dave logged in, the **who am i** command will return either a machine name where Dave connected from, or the IP address if the local **/etc/hosts** file has no record of Dave's terminal to look up the name mapping. So one of the two variables will be set to one. Add them together and the next variable will always be set to one whenever Dave logs in. From here the test is simple and all Dave's personal settings can be set-up. For some real world examples of **.profile** and **.cshrc** files, see Appendix A which contains listings from my own system.

Environment:

As well as the variables you set yourself or export into the environment using the **export** command, there are some *environment variables* which get set when you launch or execute a **sh** script. We covered the special variables like `$$` and the *positional parameters* (`$digit`) in , but there are some more. There is a set that you collect at login time from the system defaults and another set supplied by the shell, as it is invoked. Some of these can be updated by the shell, some cannot. The following list is the default set showing where they are set and their default values.

Example environment variables

- HOME Is set by LOGIN to your *home directory* and is used as the default argument for the **cd** command. This can be re-set.
- PATH The *command search path* is set at login and updated when a shell is invoked to reflect the path for that shell. This is volatile and can be updated unless the shell is restricted (`usr/bin/rsh`), in which case it is fixed.
- CDPATH The *path* of the **cd** command. Needed in case you screw up the PATH.
- PS1, PS2 The primary and secondary *prompt* strings. Initialised by the shell. Can be revised if required (usually: root = #, user = \$).
- IFS Internal Field Separators (normally *space, tab and newline*). The **cut -d** option temporarily changes the IFS for the duration of the **cut** command.
- SHELL If set and if the contents include the string **rsh**, shell will become the restricted shell (`usr/bin/rsh`).
- SHACCT If set to a filename writable by the user, the shell will write an account record in the file for each shell procedure executed.
- MAIL +
- MAILPATH +
- MAILCHECK Are all related to the mail system you are using. See the mail command for details.
- LC_TYPE +
- LC_MESSAGE Are related to the local language and character sets in use. They allow messages and codes to be displayed in the local language and character set.

The shell provides defaults for PATH, PS1, PS2, IFS, and MAILCHECK. While the user login service provides defaults for HOME and MAIL. The SHELL variable is not changeable at any time and is set at shell invocation time. The PATH variable becomes immutable if the SHELL contains a reference to the restricted shell. In the restricted shell certain other commands will also become unavailable, but which ones rather depends on your system administrator.

Debugging Scripts

There are actually very few helpful things you can do to debug scripts. This however makes the task easier because there are less things to remember and because you soon learn to be very systematic in your approach. Firstly there are two **flags** you can set, either at the start of the script or part way through, which can give you information about what the shell thinks it is seeing. Then you can use the **echo** command to see what particular variables contain. Finally you can rely on the error messages the shell presents when it encounters a problem.

Flags:

The two flags are the **-v** and **-x** options for the shell. We have met these before in section , in passing, now lets see what they do and how to use them.

They can both be set on the command line, if required, by calling the script using a command line that looks something like:

```
/bin/sh -xv my_script arg_1 arg_2
```

A more usual approach is to put the options on the first line of the script file as was shown in Example in Basic Shells. This will switch both options on for the whole script in either case. If this generates too much output, which is possible for long scripts, then you can be more selective by using the **set** command part way down the script. Example shows both options around a problem area:

example flags

```
echo "Start of Problem Area"
set -v on
set -x on
if [ -f $input_file ] && [ ! -f $output_file ]
then
    more=`wc -l $input_file | cut -c1-8`
    while $more
    do
        line=`head -$more $input_file | tail -1`
        echo "$more: $line" >> $output_file
        more=`expr $more - 1`
    done
else
    echo "Error with files [$input_file] [ $output_file]"
    exit
fi
set -v off
set -x off
echo "End of Problem Area"
```

What this example will show is two things. Firstly, the **-v** will show every statement that the shell sees. It will echo the statements to the screen as the script runs. The **-v** option will also substitute the values of any variables for you so you can see the values being generated inside any **if** statements or other complex command constructs. Secondly, the **-x** option will indicate whether the script has executed the statement by putting a *plus sign* (+) in front of each statement processed as they are **echoed** to the screen. In the example, a plus sign would be inserted before the **if** statement, if it were true. If it were untrue, you would get a plus sign in front of the **else** statement. In this way you can monitor what values are being set and what statements are being processed during execution.

Another useful flag for debugging is the **-n** flag which causes the shell to *parse* the syntax structures without execution. It cannot do any damage to input or output files because nothing is being executed, but it will complain about syntax errors in your code and point out quote mismatches in the usual way, with cryptic messages about unexpected end of file. The lesser used flags are as follows:

- The **-u** flag causes unset variables to be treated as an error condition. This is useful in detecting which variables managed to escape substitution and could point to the requirement of a default.
- The **-t** flag exits after reading and executing one command.
- The **-e** flag immediately exits if a command terminates with a non-zero exit status.
- The **--** flag instructs the shell not to change any flags.

There are one or two more which I have yet to find a real use for. Suggest you check your man pages and see if you can throw some light on them. You can also use the + instead of the - in front of any flag which has the effect of turning the flag off instead of on. Don't forget, the current flags can be found in the \$- variable.

Echo:

Next in the debugging tool-set is the **echo** statement. The easiest you would think, but it is amazing how many times this one catches you out. Usually something is wrong with the script, you know about where the problem is and so you put in a few **echoes** to narrow down the area. Then when you re-run the script, either nothing is printed at all or the message "unexpected end of file" is printed. Very frustrating! What you have to watch here is the use of *quotes* and making sure that they *balance*. The key here is the message "unexpected end of file". It is usually accompanied by a reference to a line number which is one more than there are lines in the script. This is your clue. The message is telling you that you have opened a quote and the script interpreter has scanned the rest of the file and cannot find a matching closing quote. It could even be some way *above* where you think the problem is, where the opening quote you have highlighted may be acting as the closing quote for the real problem. The second clue for this is the fact that your **echo** has not printed out anything at all - the **echo** statement, itself enclosed by the bad quoting, has been hidden from the shell.

Once you have managed to get your **echoes** working, use them to check expected values before and after **tests** and in loops. Use the square bracket [] to enclose the variables, as shown in the numerous examples in this book. That way leading and trailing *blanks* will show up making it obvious why a **test** fails. Make use of the **/usr/5bin/echo** command, with its extra controls, to indicate the status of loops as shown in Example below. Here the **echo** is outputting a line of dots - one dot per loop cycle.

Example echo

```
while $more
do
    /usr/5bin/echo “.\c”                # Debug Dot
    line=`head -$more $input_file | tail -1`
    echo “$more: $line” >> $output_file
    more=`expr $more - 1`
done
```

Null Parameter Trap:

Another useful feature is one we met earlier under Parameter Substitution in . If your not sure if a parameter has been set and it is difficult to see - it might be set to a non-printing character for instance - then you can detect this by using one of the parameter substitution forms shown below:

Null Parameter Example

```
echo “Difficult Parameter = [${arg:?}]”
```

What this does is check whether the parameter has been set and returns either its value or the message “parameter null or not set”. Try setting the parameter to <CTRL-G> and see what happens. The window will flash as the echo tries to display the character, but the square brackets show no character between them at all. Now set the parameter to the empty string and try again. This time the message is returned and the script exits, proving it has really been set to NULL.

Error Messages:

Get to know the error messages for your implementation. There is nothing wrong with generating error messages, just get to know what they mean. In this way, when they show up unexpectedly, you have a few clues about what is going wrong and how to fix it. There are a few key ones on my system which are good pointers. The messages themselves are usually very misleading, because the same messages are generally used for a whole range of errors relating to a syntactical element. But if you mess up enough you will eventually see which errors are responsible for particular messages. Here are some rules of thumb:

<u>Message</u>	<u>Meaning</u>
1. unexpected end of file at <i>nnn</i>	- Bad <i>quoting</i> or complex command <i>group</i>
2. test expected	- No space around [or] or missing then
3. badly formed if	- Missing then or fi statements
4. missing test	- No do under while or for statements
5. syntax error	- Tried to compare <i>alpha</i> and <i>number</i> in test
6. invalid number	- A <i>string</i> in expr or bc contains <i>alpha</i> characters
7. bad number	- A <i>real number</i> used in expr (must be integer)

Start a file of all the messages you get from your shell. Against each message, put a list of all the errors that have generated that message and how you fixed the problem. You will find this most valuable and a great time saver.

Emergency Exit:

If you need to stop a script while it is running, use the <CTRL-C> character. If you have launched a script into background by appending the & symbol, then you will need to return it to foreground

processing first, using the **fg** command, then use the <CTRL-C> character. If you have launched a script from the **crontab** and it is running away or stuck in an infinite loop, then you must locate its PID by using:

```
ps -ef | grep script_name
```

...then use...

```
kill -9 PID
```

...to kill the process. Also be on the look out for sub-processes that may have started in background (intentionally or otherwise!) from lists or sub-shells terminated by an ampersand (&). You may have to trace these back using the parent PID of the calling script, so remember to print it out or log it. The **ps -ef** command shows the current PID of all executing commands as well as the parent PID of all spawned processes.

Design Considerations

One of the key design considerations for any piece of code is the house style to be used within your company or department. It will only take a short while to list out the rules you want others to follow but this will bring real benefits when someone new has to edit one of your files. There is nothing worse than changing someone else's code when you are not sure where objects are derived from or what variable names have been used already, or what they mean. If you use meaningful variable names, then the chances are, they'll be easier to spot in the code. Structure your code so that each section follows in a logical order, making the code easier to read. Sprinkle the code with comments, especially in the difficult to follow areas.

This next section is actually a copy of a Style Guide I created for my development team to follow. It follows all the rules you have been unwittingly exposed to whilst reading this book and a few more besides. You might want to use this as a basis for your own style guide, then again you may not. The choice is yours. But do give it a read first - Please!

Shell Script Style Guide:

This document is a Style Guide for writing and revising UNIX Shell Scripts. Following this style guide will ease the burden of future maintenance and greatly improve the readability of any conforming code segments. Users new to Shell Scripting should read this guide first and observe the layout rules when generating new code to preserve the house style and guarantee ease of future maintenance. The guide is broken down into three main areas:

1. Banners
2. Symbols
3. Layout

Banners:

Script Banners are used to carry information about the whole script in a readable format. The rules used here create banners that are readable by script users and embedded code functions. Following this style is therefore essential for continued operation of some areas of code. The style and format of the UNIX Script Banners used here are composed of 4 main parts:

1. Shell Language Indicator
2. Title Line
3. Description and Support
4. History

Shown below is an example banner from a current UNIX Bourne shell script followed by a detailed breakdown of its component parts.

Banner Example

```
#!/bin/sh
#####
```

```
# resource (c) RHReepe 1996 February 26 Version 1.0 #
#####
# Description: Allows resource maintenance for databases
# Syntax: resource(menu) (Alias resource)
# Extern: s_banner s_answer s_menu_list s_menu_pick
#####
# Notes: The startup menu requests the user to choose
# between Resource Usage Report or Resource Delete options.
# A second menu displays a selection list of Resource Types
# to choose from. A third menu then displays a dynamic
# list of user entered files from the LIST subdirectory.
# The files contain lists of resource id's to process
# which are appropriate for the type selected in menu 2.
# Usage reports are generated for each production database
# in turn. Delete requests are processed against master
# databases at each site only, with an update to all
# production databases COST_MODEL_USAGE flag. Deletes
# will be propagated to production overnight by the
# normal download mechanism from the masters.
#####
# 19960226 RHR Genesis
```

Detailed Banner Notes:

1. All the lines in this banner are comments, except:
2. Line 1 is not a comment, it is a shell indicator (See -).
3. Line 2 is the banner start marker.
4. Line 3 is the header and contains 10 fields as follows:

```
#          Comment Indicator
word          Name of Script [must not include spaces - use the
underscore]
Copyright    Flag [used by s_version() function]
Authors name  Must use one field
1996         Year of creation
February     Month of creation
26           Day of creation
Version      Key word [used by s_version() function]
Version Number Important to be field 9 - [Function s_version() expects it]
#          End of Comment marker
```
5. Line 4 is a banner divider line.
6. Line 5 is a short description. Note the use of an Alias at the end of the line, which indicates that the script can be executed from any directory as the Alias will always point to the correct script at execution time. Also, the script name is followed by the word menu in parentheses which indicates that a menu will start up when the script runs [the script **help** expects it].
7. Line 6 is a syntax description. Alternatives should be shown in brackets. Any optional arguments should be enclosed within square brackets [the script **how** expects it].
8. Line 7 is a list of the external shell functions called by this script [useful when updating].
9. Line 8 is a banner divider line.

- 10.Line 9 is the start of an [Optional] Notes section which details operational processes for complex scripts. This can be as long or as short as required but please do not dump pages of detail here. If it is a complex procedure, break it down into smaller sections and put a note in each section.
- 11.Line 22 is the banner end marker.
- 12.Line 23 is the History start line. Comprised of Date in JIS format, Authors Initials, and Comment for each change of the script (Genesis indicates The Beginning of history for the script). Any comments should indicate what was there before the current change, what is there now is obvious for all to see.

Lesser Banners:

For function, procedure, or logical unit banners, much less detail is required. It is obvious from the main banner that the same overall condition apply for the whole script. So in these local banners, just the name and a short one-line description are required in the banner header. Notice how the actual function definition name is used as part of the title in the banner. Also the heavy handed hashes (####) are replaced by the much lighter equals (====) for the banner markers, above and below the title line. There is an optional set of lines that *can* follow this local banner to describe the passed arguments if any. See below for details.

Example of Function or Procedure Banners:

```
#=====
Banner Marker
function_name()      # (c) Author: Does this...      # Name,Auth & Description
#=====
Banner Marker
# Arg_1 = filename      # Argument 1 (Mandatory)
# Arg_2 = [number]      # Argument 2 (Optional)
```

Example of Logical_Block Banners:

```
#=====
# Doing this to that...
#=====
```

As an option, for logical block banners, the equals (====) can be substituted by minus (----), to lighten the appearance even more. As shown in the following function definition:

Example of Function Definition with Logical Blocks:

```
#=====
====
i_master_db_list() # (c) Author: Return List of Databases
#=====
====
# Arg_1 = [host_name:THISHOST]
{
```

```

#-----
# Get list of databases on host
#-----
for sid in s_sid_list ${1:-$THISHOST}
do
    db_type=`echo $sid | cut -c6`
    #-----
    # If correct type - add to list
    #-----
    if [ "$db_type" = "m" ]
    then
        master_db_list="$master_db_list $db_type"
    fi
done
echo "$master_db_list"
}

```

Symbols:

The format for code symbols is also important as an aid to readability and comprehension when modifying or creating scripts. Here are some general but useful guidelines laid down to help both layout and understanding of the code.

Symbol Form:

Symbol Description	Illustration	Comment	Detail
In Stream Comments	# Code Fragment	# A comment	(InitCaps - starts with #)
UNIX Keywords	grep -i fred wc -l	# UNIX Commands	(lowercase)
Writing Shell Variables	variable="hello world"	# Setting a Variable	(lowercase)
Reading Shell Variables	echo "\$variable"	# Retrieving Variable	(lowercase)
Environment Variable	echo "\$THISHOST"	# Retrieving Variable	(UPPERCASE)
Internal Function	i_sub_doit	# Function Name	(lowercase starts with i_)
External Function	s_sub_doit	# Function Name	(lowercase starts with s_)

The use of symbol prefixes to identify class or some other connection between symbols is allowed as long as it is made plain by corresponding comments. So several variables starting with "r_" could be associated with some "remote" connection as shown at the start of the following code segment:

Class Form:

```

#=====
# Get Remote Database List from Remote Server
#=====
r_database_list=`s_sid_list $r_host`

```

Layout:

The layout of code within a script has a great impact on readability. In general the flow of the program should be immediately obvious from the general layout. To this end, indenting of code

within predicates and loop constructs shall be observed. Generally, indenting should be carried out by tab-stops, but for heavily nested code this will be relaxed in favour of 3 spaces per nesting level to stop the code leaving the page. The following rules show the most common layout placements.

Layout Rules:

1. Use Blank Lines between logical programming blocks, function definitions, and any other programming construct which defines a related object, or complex command group.
2. Use Tabs to indent code when inside loops, functions, or predicates.
3. Use 3 Spaces to indent code if nesting level becomes too deep.
4. Use in-stream comments to explain complex structures.
5. Keep in-stream comments justified to a column - at least on a per block basis.
6. Don't do it twice, use an internal function.
7. If you use it again in another script, create an external function in one of the files called from \$ORACLE_BASE/.profile_functions.
8. Use variable names that mean something (up to 31 characters recognized).
9. Echo all valid results to a logfile for cron scripts.
10. Echo invalid results to screen (Cron will mail them back during a failure).
11. Use -xv on line 1 (shell indicator) for debugging.
12. Create all temporary files in /tmp directory.
13. Use filename suffixes to indicate probable content (.log, .lst, .dat, .mnu, sql, etc.).
14. Pre-delete temp files, but don't delete after use when debugging - they are handy for inspecting the outcome.
15. Look through the list of available functions and subroutines, it may already be there.
16. Use the .profile files to get a good selection of existing predefined variables. They may save you some time.

The following example shows most of the rules in action within a dummy script layout. For a much broader example of the house style, users are requested to inspect the current UNIX shell scripts in the \$ORACLE_BASE/scripts directory or the \$ORACLE_BASE/cron directory.

Layout Example:

```
. $HOME/.profile                                # Read in the default environment

tmp0=`s_tmp 0`                                  # Create a temporary file name
tmp1=`s_tmp 1`                                  # Create a temporary file name

#####
# Create SQL statement in Temp file
#####
cat >> $tmp0 <<-EOF                             # Fill the tempfile with data up to the EOF flag
    SELECT name FROM v\${database};             # First line of data
    EXIT                                         # SQL EXIT statement
EOF                                              # EOF flag marker

#####
# Set up Account Strings
#####
s_set_account_name "system_coded" $0 $$        # Pass Coded UserID to Security Audit Function8
.s_account_string$$                             # Retrieve Login String
```

⁸ It is a requirement for our company not to have User ID's and Passwords in a readable format within any code segments. This two line extract complies with this requirement and also records an audit trail of database accesses. The s_set_account_name function also calls C++ functions to complete its tasks which improves security effectiveness.

R2's Shell Tutorial

```
#=====
# Loop through Databases and Process SQL
#=====
db_sid_list=`s_sid_list $THISHOST`          # Create a list of DB SID's for this host name
for sid in $db_sid_list                    # Create a loop to process each SID in the list
do                                          # Start of DO LOOP
    sql_command="sqlplus -s $account@t:$THOSHOST:$sid" # Create an Oracle Login string9
    $sql_command @tmp0 >> tmp1           # Run the SQL and log the output
done                                      # End of DO LOOP
```

⁹ The SQL*Plus connection string is using SQL*Net 1.0 syntax. For SQL*2.0 syntax, the t:\$THISHOST: part is not required.

Putting It All Together

This whole section is just a listing of a current development script. I make no claims to its accuracy or functionality. Being a development site most of my scripts are in constant change anyway, which is why I embarked on a method of simplification and supportability. In these pages you should find more than enough examples of ways to do things. Some good, some not so good. The thing is, you should now be able to understand what this listing means and how it works.

Database Generator Script:

This listing is a Bourne Shell Script which uses many of the features already described in this document. There are still some areas where another function would be helpful but the script is still in flux and will not yet submit to functions in all areas. The script itself is used to create an empty Oracle database on a Solaris platform. The script expects a particular disk structure based on the Oracle Flexible Architecture scheme and so there is some hard coding in places. The tablespace sizes and extent and segment defaults are all selected and calculated from some dependant environment variables which are set up on a per server basis, with numbers based on expected user community size. There is a special profile called `.profile_references` which contains the complete networked site parameters for all the servers in the network that this script runs on. This profile is also duplicated in appendix A for your information.

generate_db listing

```
#!/bin/sh
#####
# generate_db (c) R. H. Reepe 9th February 1995 Version 3.1 #
#####
# Description: Generates a Database by asking questions and picking up various system defaults
# Syntax: generate_db
# Extern: s_running, s_stopping, s_find_string, s_get_string
#####
# Process generates a create.sql script and then uses sqldb
# to run the script. Automatic disk stripe sequencing is
# carried out as is multiple files per tablespace. Database
# size is gauged from .profile_references DB_SIZES parameter
# for the host THISHOST. Globally Unique Database Names are
# guaranteed as long as DATABASE_REFERENCES is maintained
# with this in mind. Tablespace Sizes are relational and
# keep in step as database size is increased. Some tablespaces
# are further modified by database type constraints.
# =====
# 960515 RHR Genesis
# 960803 RHR Revised Generated Create Script into two parts and logged
# 960912 RHR Added GDU controled SGA & Buffer Cache Sizes
# 961016 RHR Updated TEMP Tablespace size from gdu1
# 961025 RHR Updated ROLBACK Segment size from 200 k to gdu6 k
# 961025 RHR Updated All Segment Sizes to be 1% of Tablespace Size
# 961114 RHR Updated ROLBACK Tablespace Size from gdu5 to gdu6 (100%)
# 961118 RHR Updated Definitions of Segment Sizes to source from tablespace sizes

debug="off" # Debugging Option - set to ( on | off )

. $HOME/.profile # Setup Environment

s_header # Generate a Program Header

s_running $0 # Detect a second run

tmp0=`s_tmp 0` # Setup Temporary Filenames
tmp1=`s_tmp 1`
tmp2=`s_tmp 2`
tmp3=`s_tmp 3`
log0=`s_plog` # Logfile for this process

disk_full=77 # Limit Space on Disk in Percent

#####
```

R2's Shell Tutorial

```

if [ "$debug" = "on" ]
then
    echo "          Setting Up Variables"
    s_stopping $0
fi
#####

#=====
# Get Generic Database Unit Size (GDU) for this Host
#=====
position=`s_find_string $THISHOST $REFERENCE_DBSERVER`      # Find position of host in list
gdbu_size=`s_get_string $position $REFERENCE_DB_SIZES`      # Find Database Size from host position
echo "Generic Database Unit Size for server $THISHOST = $gdbu_size Mb"      # Display Size

#=====
# Set up some useful GDU multipliers for later use
#===== # gduN = (Fraction of GDU)
gdu0=`echo "$gdbu_size / 3" | bc`      # 0 = (1/3 gds)
gdu1=`expr $gdu0 + $gdu0`      # 1 = (2/3 gds)
gdu2=`expr $gdu0 + $gdu1`      # 2 = ( 1 gds)
gdu3=`expr $gdu2 + $gdu2`      # 3 = ( 2 gds)
gdu4=`expr $gdu2 + $gdu3`      # 4 = ( 3 gds)
gdu5=`expr $gdu2 + $gdu4`      # 5 = ( 4 gds)
gdu6=`expr $gdu5 + $gdu5`      # 6 = ( 8 gds)
gdu7=`expr $gdu5 + $gdu6`      # 7 = (12 gds)
gdu8=`expr $gdu5 + $gdu7`      # 8 = (16 gds)
gdu9=`expr $gdu5 + $gdu8`      # 9 = (20 gds)

#=====
# Assign Sizes to Database Tablespace Objects      Size of Tablespace Files in Mbytes      =====
#=====
export SYST_F_SIZE=$gdu2      # Size of SYSTEM tablespace
export TOOL_F_SIZE=$gdu2      # Size of SYSTEM_TOOLS Tablespace
export TEMP_T_SIZE=$gdu2      # Size of TEMP Tablespace
export ADMN_F_SIZE=$gdu1      # Size of ADMIN Tablespace
export INDX_F_SIZE=$gdu2      # Size of INDEXES Tablespace
export CAPE_F_SIZE=$gdu2      # Size of CAPE_OBJECTS Tablespace
export ROLB_F_SIZE=$gdu4      # Size of ROLBACK Tablespace

export REDO_F_SIZE=128      # Size of REDO_LOG Files (In K)

export REDO_MAX_HIST=`echo "$gdu2 * 2" | bc`      # Max Number of REDO logs to Archive

#=====
# Assign Sizes to Database Segment Objects      Size of Segments in KBytes      =====
#=====
export TOOL_SEG_INIT=`echo "$gdu0 * 3" | bc`      # Size of TOOLS Segment INIT Parameter
export ADMN_SEG_INIT=`echo "$gdu0 * 3" | bc`      # Size of ADMIN Segment INIT Parameter
export SYST_SEG_INIT=`echo "$gdu0 * 3" | bc`      # Size of SYSTEM Segment INIT Parameter
export TEMP_SEG_INIT=`echo "$gdu0 * 3" | bc`      # Size of TEMP Segment INIT Parameter
export CAPE_SEG_INIT=`echo "$gdu0 * 2" | bc`      # Size of CAPE_OBJECTS Segment INIT Parameter
export INDX_SEG_INIT=`echo "$gdu0 * 2" | bc`      # Size of INDEXES Segment INIT Parameter

export TOOL_SEG_NEXT=`echo "$gdu2 * 8" | bc`      # Size of TOOLS Segment NEXT Parameter
export ADMN_SEG_NEXT=`echo "$gdu1 * 8" | bc`      # Size of ADMIN Segment NEXT Parameter
export SYST_SEG_NEXT=`echo "$gdu1 * 8" | bc`      # Size of SYSTEM Segment NEXT Parameter
export TEMP_SEG_NEXT=`echo "$gdu2 * 8" | bc`      # Size of TEMP Segment NEXT Parameter
export CAPE_SEG_NEXT=`echo "$gdu2 * 8" | bc`      # Size of CAPE_OBJECTS Segment NEXT Parameter
export INDX_SEG_NEXT=`echo "$gdu1 * 8" | bc`      # Size of INDEXES Segment NEXT Parameter

export ROLB_SEG_SIZE=`echo "$gdu8 * 2" | bc`      # Size of ROLBACK Segment INIT and NEXT Parameters
export ROLB_SEG_OPTM=`echo "$gdu8 * 4" | bc`      # Size of ROLBACK Segment OPTIMAL Parameter

#####
if [ "$debug" = "on" ]
then
    echo "          Generic Database Unit Profile and SEGMENT Size Allocation"
    echo ""
    echo ""
    =====
    echo "GDU Numbers:      0      1      2      3      4\
      5      6      7      8      9"
    echo "GDU Values:      $gdu0      $gdu1      $gdu2      $gdu3      $gdu4\
      $gdu5      $gdu6      $gdu7      $gdu8      $gdu9"
    echo ""
    =====
    echo ""
    echo "Database File Sizes for Tablespaces"
    echo ""
    echo "SYST_F_SIZE      = $SYST_F_SIZE"
    echo "TOOL_F_SIZE      = $TOOL_F_SIZE"
    echo "TEMP_T_SIZE      = $TEMP_T_SIZE"
    echo "ADMN_F_SIZE      = $ADMN_F_SIZE"
    echo "INDX_F_SIZE      = $INDX_F_SIZE"
    echo "CAPE_F_SIZE      = $CAPE_F_SIZE"
    echo "ROLB_F_SIZE      = $ROLB_F_SIZE"
    echo ""
    echo "Database Default Segment Size Paramaters for Tablespaces"
    echo ""
    echo "ADMN_SEG_INIT      = $ADMN_SEG_INIT K"
    echo "CAPE_SEG_INIT      = $CAPE_SEG_INIT K"
    echo "INDX_SEG_INIT      = $INDX_SEG_INIT K"
    echo "SYST_SEG_INIT      = $SYST_SEG_INIT K"
    echo "TEMP_SEG_INIT      = $TEMP_SEG_INIT K"
    echo "TOOL_SEG_INIT      = $TOOL_SEG_INIT K"
    echo ""
    echo "ADMN_SEG_NEXT      = $ADMN_SEG_NEXT K"
    echo "CAPE_SEG_NEXT      = $CAPE_SEG_NEXT K"
    echo "INDX_SEG_NEXT      = $INDX_SEG_NEXT K"
    echo "SYST_SEG_NEXT      = $SYST_SEG_NEXT K"

```

R2's Shell Tutorial

```
echo "TEMP_SEG_NEXT      = $TEMP_SEG_NEXT K"
echo "TOOL_SEG_NEXT      = $TOOL_SEG_NEXT K"
echo ""
echo "ROLB_SEG_SIZE      = $ROLB_SEG_SIZE K"
echo "ROLB_SEG_OPTM      = $ROLB_SEG_OPTM K"
echo ""
echo "REDO_F_SIZE         = $REDO_F_SIZE K"
echo "REDO_MAX_HIST       = $REDO_MAX_HIST"
echo ""
s_stopping $0
exit
fi
#####

#=====#
# Assign Sizes to Database Parameters
#=====#
export SGA_SIZE=`echo "600000 * $gdbu_size" | bc`      # Size of General SGA Memory
export DB_BUFFER_SIZE=`echo "100 * $gdbu_size" | bc`   # Size of DB_BLOCK_BUFFER Cache (2K Blocks)

#=====#
# Assign Remaining Variables
#=====#
export OS=`/usr/bin/uname -r | cut -c1-1`              # Unix Version (4=sunos,5=solaris)
export ORACLE_BASE=/ford/$THISHOST/unix/cen/oracle_base # Oracle Base Install Directory
export ORACLE_GEN=/ford/$THISHOST/unix/cen/oracle_base/GENERIC # Oracle GENERIC Files Directory

#####
if [ "$debug" = "on" ]
then
echo "      Selecting Database Software Product"
fi
#####

. $ORACLE_BASE/scripts/product                        # Run product selector script

#####
if [ "$debug" = "on" ]
then
echo " THISHOST =      [ $THISHOST   ]"
echo "   OS      =      [ $OS         ]"
echo "  UNIX      =      [ $UNIX       ]"
echo "ORACLE_BASE =      [ $ORACLE_BASE ]"
echo "ORACLE_HOME =      [ $ORACLE_HOME ]"

echo "      Selecting Database Type"
fi
#####

#=====#
# Find the database host and loc string
#=====#
list_position=`s_find_string $THISHOST $REFERENCE_DBSERVER`      # Find position of host in list
db_loc=`s_get_string $list_position $REFERENCE_LOCATION`         # Get db_location string

#=====#
# Ask the user for database type
#=====#
correct=0                                                         # Set correct = wrong
pick_list=""                                                     # Initialise a pick list
for next in $REFERENCE_DATABASE                                  # Get all valid types
do
pick=`s_lower_case $next | cut -c1-1`                             # Get the option letter
pick_list=$pick_list$pick                                       # Generate the valid pick list
done
#-----#
# Generate User Menu
#-----#
while [ $correct = 0 ]
do
echo ""
echo "Which type of Database do you wish to create ?"          # ... Instructions
echo ""
echo "Choices are:"
echo ""
for next in $REFERENCE_DATABASE
do
pick=`s_lower_case $next | cut -c1-1`
echo "      $pick      $next Database"                          # ... Display List
done
echo ""
/usr/5bin/echo "Your choice (p,m,s,etc) = \c"                    # ... Request user input
read answer
echo ""
correct=`echo "$pick_list" | grep -c $answer`                    # Validate user answer against list
if [ "$answer" = "" ]
then
echo "Invalid Answer - Aborting"
exit
fi
done

database_type=`s_lower_case $answer`                              # This is the Valid Answer

export GENERIC_PFILE=$ORACLE_GEN/init_generic.ora.$database_type # Get Generic init.ora filename

#####
if [ "$debug" = "on" ]
then
echo "      Adjust the Tablespace Sizes according to Type"
fi
```

R2's Shell Tutorial

```
#####

#=====
# Detect a Resource Database (Small)
#=====
if [ "$database_type" = "m" ]
then
    export RESOURCE=1
    CAPE_F_SIZE=`echo $CAPE_F_SIZE / 2 | bc`
    INDX_F_SIZE=`echo $INDX_F_SIZE / 2 | bc`
else
    export RESOURCE=0
fi

#=====
# Detect a Production Database (Large)
#=====
if [ "$database_type" = "p" ] || [ "$database_type" = "e" ]
then
    export TEMP_T_SIZE=`echo "$TEMP_T_SIZE * 3" | bc`
    export CAPE_F_SIZE=`echo "$CAPE_F_SIZE * 3" | bc`
    export INDX_F_SIZE=`echo "$INDX_F_SIZE * 3" | bc`
    export CAPE_SEG_NEXT=`echo "$CAPE_SEG_NEXT * 2" | bc`
    export TEMP_SEG_NEXT=`echo "$TEMP_SEG_NEXT * 2" | bc`
    export INDX_SEG_NEXT=`echo "$INDX_SEG_NEXT * 2" | bc`
fi

#####
if [ "$debug" = "on" ]
then
    echo "          Create the Empty Directories"
fi
#####

/usr/5bin/echo "Checking Disks and Directories...\c"
oracle_data_dirs="/ford/STHISHOST/u/oracle_data*"
oracle_sub_dirs="ADMIN ARCH TOOLS TEMP CONTROL_FILES REDO_LOGS SYSTEM DUMP ROLBACK USER RESOURCE INDEXES BACKUPS"
/usr/5bin/echo "\c"
for directory in $oracle_data_dirs
do
    /usr/5bin/echo "\c"
    cd $directory
    /usr/5bin/echo "\c"
    mkdir $oracle_sub_dirs > /dev/null 2>&1
    /usr/5bin/echo "\c"
    chgrp dba $oracle_sub_dirs > /dev/null 2>&1
    /usr/5bin/echo "\c"
    chmod -R 775 $oracle_data_dirs > /dev/null 2>&1
done
cd
/usr/5bin/echo "\c"
if [ $OS = 4 ]
then
    export UNIX="sunos"
    export ORATAB=/etc/oratab
else
    export UNIX="solaris"
    export ORATAB=/var/opt/oracle/oratab
fi

echo "."
initialise_db=$ORACLE_BASE/scripts/initialise_db

#=====
# Find all the globally similar sids
#=====
for host in $REFERENCE_DBSERVER
do
    sid_list=`s_sid_list $host`
    for sid in $sid_list
    do
        echo "$sid" >> $tmp1
    done
done
database_type="$database_type"
grep $db_loc$db$database_type $tmp1 | sort > $tmp2
current_db_seq=`tail -1 $tmp2 | cut -c8-8`

#=====
# Test sequence number and increment for new DB
#=====
if [ "$current_db_seq" = "" ]
then
    db_seq=1
else
    db_seq=`expr $current_db_seq + 1`
fi
seq_lng=`/usr/5bin/echo "$db_seq\c" | wc -c | cut -c1-8`
if [ $seq_lng = 1 ]
then
    db_seq="0$db_seq"
fi

export ORACLE_SID=$db_loc$db$database_type$db_seq
echo "This Database Name = $ORACLE_SID"

#####
if [ "$debug" = "on" ]
then
    echo "Database Location String is [ $db_loc ]"
    echo "Answer is [ $database_type ]"
    echo "Length is [ $seq_lng ]"
fi
```

R2's Shell Tutorial

```
echo "Database Sequence Number is [ $db_seq ]"
echo "Database Instance is [ $ORACLE_SID ]"
echo ""
echo "      Selecting Database Slot (A,B,C, or D)"
fi
#####

echo "Checking Database Slots In Use"

#####
# NOTE: When SQL-Net 2.1 conversion ready - this reference #
# to ORATAB must be replaced with a dual call to ORATAB and #
# TNSNAMES to check for version 1 & 2 SQL-Net setups      #
#####
for instance in `grep -v \# $ORATAB | cut -f1-1 -d:`          # List of DB_Names
do
  slot=`du -a /ford/$THISHOST/u/oracle_data*/SYSTEM | /bin/nawk '{print($2)}' | grep $instance*_system_01.dbf`
  if [ ! "$slot" = "" ]                                     # If there are database files for this instance
  then
    slot=`dirname $slot`                                    # Strip off a dir_level
    slot=`dirname $slot`                                    # Strip off another dir_level
    slot=`basename $slot`                                   # Take the last directory name
    echo "using slot [ $slot ]"                             # Find the database slot
    echo "slot = [ $slot ]"                                 # And add it to the list file
  else
    echo "Not on oracle_data disks"                         # This Database is probably a DBA database
  fi
done

slot1=`grep 1 $tmp3 | wc -l | cut -c8-8`                    # Check each slot for use
slot2=`grep 2 $tmp3 | wc -l | cut -c8-8`                    # Check each slot for use
slot3=`grep 3 $tmp3 | wc -l | cut -c8-8`                    # Check each slot for use
slot4=`grep 4 $tmp3 | wc -l | cut -c8-8`                    # Check each slot for use

#####
# Choose a slot                                           # Find the first empty slot
#####
slot=1
if [ "$slot2" -lt "$slot1" ]
then
  slot=2
elif [ "$slot3" -lt "$slot2" ]
then
  slot=3
elif [ "$slot4" -lt "$slot3" ]
then
  slot=4
fi

#####
if [ "$sdebug" = "on" ]
then
  echo "      Set up the Dynamic Resources for Creation"
fi
#####

if [ "$slot" = "1" ]                                       # Name the Slot Directories
then
  export O_D_2 ; O_D_2=/ford/$THISHOST/u/oracle_data_1    # Oracle Data 1 root path
  export O_D_3 ; O_D_3=/ford/$THISHOST/u/oracle_data_2    # Oracle Data 2 root path
  export O_D_4 ; O_D_4=/ford/$THISHOST/u/oracle_data_3    # Oracle Data 3 root path
  export O_D_1 ; O_D_1=/ford/$THISHOST/u/oracle_data_4    # Oracle Data 4 root path
elif [ "$slot" = "2" ]
then
  export O_D_3 ; O_D_3=/ford/$THISHOST/u/oracle_data_1    # Oracle Data 1 root path
  export O_D_4 ; O_D_4=/ford/$THISHOST/u/oracle_data_2    # Oracle Data 2 root path
  export O_D_1 ; O_D_1=/ford/$THISHOST/u/oracle_data_3    # Oracle Data 3 root path
  export O_D_2 ; O_D_2=/ford/$THISHOST/u/oracle_data_4    # Oracle Data 4 root path
elif [ "$slot" = "3" ]
then
  export O_D_4 ; O_D_4=/ford/$THISHOST/u/oracle_data_1    # Oracle Data 1 root path
  export O_D_1 ; O_D_1=/ford/$THISHOST/u/oracle_data_2    # Oracle Data 2 root path
  export O_D_2 ; O_D_2=/ford/$THISHOST/u/oracle_data_3    # Oracle Data 3 root path
  export O_D_3 ; O_D_3=/ford/$THISHOST/u/oracle_data_4    # Oracle Data 4 root path
elif [ "$slot" = "4" ]
then
  export O_D_1 ; O_D_1=/ford/$THISHOST/u/oracle_data_1    # Oracle Data 1 root path
  export O_D_2 ; O_D_2=/ford/$THISHOST/u/oracle_data_2    # Oracle Data 2 root path
  export O_D_3 ; O_D_3=/ford/$THISHOST/u/oracle_data_3    # Oracle Data 3 root path
  export O_D_4 ; O_D_4=/ford/$THISHOST/u/oracle_data_4    # Oracle Data 4 root path
else
  echo "No Slots Available - Program Screw-up"            # This should never happen (!)
  exit
fi

echo "Naming Database Files"

export DB_SYST_1=$O_D_1/$SYSTEM/$ORACLE_SID*_system_01.dbf # Set the DB File full paths
export DB_SYST_2=$O_D_1/$SYSTEM/$ORACLE_SID*_system_02.dbf
export DB_ROLE_1=$O_D_1/ROLBACK/$ORACLE_SID*_rollback_1.dbf
export DB_ROLE_2=$O_D_3/ROLBACK/$ORACLE_SID*_rollback_2.dbf
export DB_ROLE_3=$O_D_4/ROLBACK/$ORACLE_SID*_rollback_3.dbf
export DB_CONT_1=$O_D_2/CONTROL_FILES/$ORACLE_SID*_cnt11
export DB_CONT_2=$O_D_4/CONTROL_FILES/$ORACLE_SID*_cnt12
export DB_TOOLS=$O_D_3/TOOLS/$ORACLE_SID*_tools_01.dbf
export DB_TEMP=$O_D_1/TEMP/$ORACLE_SID*_temp_01.dbf
export DB_ADMIN=$O_D_2/ADMIN/$ORACLE_SID*_admin_01.dbf
export DB_REDO_1A=$O_D_2/REDO_LOGS/$ORACLE_SID*_redo1a.log
export DB_REDO_2A=$O_D_2/REDO_LOGS/$ORACLE_SID*_redo2a.log
export DB_REDO_3A=$O_D_2/REDO_LOGS/$ORACLE_SID*_redo3a.log
export DB_REDO_4A=$O_D_2/REDO_LOGS/$ORACLE_SID*_redo4a.log
```

R2's Shell Tutorial

```
export DB_REDO_5A=${O_D_2}/REDO_LOGS/$ORACLE_SID redo5a.log"
export DB_REDO_1B=${O_D_4}/REDO_LOGS/$ORACLE_SID redo1b.log"
export DB_REDO_2B=${O_D_4}/REDO_LOGS/$ORACLE_SID redo2b.log"
export DB_REDO_3B=${O_D_4}/REDO_LOGS/$ORACLE_SID redo3b.log"
export DB_REDO_4B=${O_D_4}/REDO_LOGS/$ORACLE_SID redo4b.log"
export DB_REDO_5B=${O_D_4}/REDO_LOGS/$ORACLE_SID redo5b.log"
export DB_USER_1=${O_D_3}/USER/$ORACLE_SID cape_objects_01.dbf"
export DB_USER_2=${O_D_3}/USER/$ORACLE_SID cape_objects_02.dbf"
export DB_USER_3=${O_D_3}/USER/$ORACLE_SID cape_objects_03.dbf"
export DB_USER_4=${O_D_3}/USER/$ORACLE_SID cape_objects_04.dbf"
export DB_INDX_1=${O_D_1}/INDEXES/$ORACLE_SID indexes_01.dbf"
export DB_INDX_2=${O_D_1}/INDEXES/$ORACLE_SID indexes_02.dbf"
export DB_INDX_3=${O_D_1}/INDEXES/$ORACLE_SID indexes_03.dbf"
export DB_INDX_4=${O_D_1}/INDEXES/$ORACLE_SID indexes_04.dbf"

#####
if [ "$debug" = "on" ]
then
    echo "          Set up the Static Resources for Creation"
fi
#####

export DB_ARCH=${O_D_4}/ARCH
export DB_DUMP=${O_D_4}/DUMP

#####
if [ "$debug" = "on" ]
then
    echo "          Database Constructed Parameter File"
fi
#####

echo "Creating Parameter File"

#=====#
# Parameter File Names
#=====#
export DB_PFILE=${ORACLE_DBS}/init${ORACLE_SID}.ora"
export DB_PFI_BU=${ORACLE_BASE}/SQL/DB_CREATE/init${ORACLE_SID}.ora"
export DB_CREATE_1=${ORACLE_DBS}/create_1-${ORACLE_SID}.sql"
export DB_CREATE_2=${ORACLE_DBS}/create_2-${ORACLE_SID}.sql"
export DB_CRE_BU_1=${ORACLE_BASE}/SQL/DB_CREATE/create_1-${ORACLE_SID}.sql"
export DB_CRE_BU_2=${ORACLE_BASE}/SQL/DB_CREATE/create_2-${ORACLE_SID}.sql"

#=====#
# Parameter File and Create File Clean-up and Prepare
#=====#
rm -f $DB_PFILE DB_PFI_BU
touch $DB_PFILE
rm -f $DB_CREATE_1 $DB_CRE_BU_1
touch $DB_CREATE_1
rm -f $DB_CREATE_2 $DB_CRE_BU_2
touch $DB_CREATE_2

#=====#
# Start Building Parameter File
#=====#
cat $ORACLE_GEN/init_header.ora
echo "# Specific init.ora file for database $ORACLE_SID"
echo "#####"
echo ""
echo "# Read Generic Parameter File"
echo "# -----"
echo ""
echo "          ifile                                = $GENERIC_PFILE"
echo ""
echo "# Instance Name"
echo "# -----"
echo ""
echo "          db_name                                = $ORACLE_SID"
echo ""
echo "# Memory Limits"
echo "# -----"
echo ""
echo "          db_block_buffers                        = $DB_BUFFER_SIZE"
echo "          open_cursors                          = $DB_BUFFER_SIZE"
echo "          shared_pool_size                      = $SGA_SIZE"
echo ""
echo "# Dumpfiles"
echo "# -----"
echo ""
echo "          background_dump_dest                  = $DB_DUMP"
echo "          user_dump_dest                        = $DB_DUMP"
echo ""
echo "# Control Files"
echo "# -----"
echo ""
echo "          control_files                        = $DB_CONT_1"
echo "          control_files                        = $DB_CONT_2"
echo ""
echo "# Archiver"
echo "# -----"
echo ""
echo "          log_archive_start                    = true"
echo "          log_archive_dest                    = $DB_ARCH/$ORACLE_SID"
echo "          log_archive_format                  = LOG%$S_%T.ARC"
echo ""

#####
if [ "$debug" = "on" ]
then
    echo "          Database Constructed Create File"
fi
```

R2's Shell Tutorial

```
fi
#####

echo "Creating Database Create Script"

#=====
# Create Script Number 1
#=====
echo "REM # Create Database (1) $ORACLE_SID on $UNIX system $THISHOST" >> $DB_CREATE_1
echo "REM # -----" >> $DB_CREATE_1
echo "REM # Script created by generate_db (c) R. H. Reepe" >> $DB_CREATE_1
echo "REM # Version 2.0 date `date`" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "connect internal" >> $DB_CREATE_1
echo "startup nomount" >> $DB_CREATE_1
echo "create database $ORACLE_SID" >> $DB_CREATE_1
echo " maxinstances 1" >> $DB_CREATE_1
echo " maxlogfiles 16" >> $DB_CREATE_1
echo " maxloghistory $REDO_MAX_HIST" >> $DB_CREATE_1
echo " character set '\E8ISO8859P2'" >> $DB_CREATE_1
echo " datafile" >> $DB_CREATE_1
echo "      '$DB_SYST_1' size $SYST_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_SYST_2' size $SYST_F_SIZE M" >> $DB_CREATE_1
echo "      logfile" >> $DB_CREATE_1
echo "          group 1 ('$DB_REDO_1A'," >> $DB_CREATE_1
echo "                '$DB_REDO_1B'" >> $DB_CREATE_1
echo "                size $REDO_F_SIZE K," >> $DB_CREATE_1
echo "          group 2 ('$DB_REDO_2A'," >> $DB_CREATE_1
echo "                '$DB_REDO_2B'" >> $DB_CREATE_1
echo "                size $REDO_F_SIZE K," >> $DB_CREATE_1
echo "          group 3 ('$DB_REDO_3A'," >> $DB_CREATE_1
echo "                '$DB_REDO_3B'" >> $DB_CREATE_1
echo "                size $REDO_F_SIZE K," >> $DB_CREATE_1
echo "          group 4 ('$DB_REDO_4A'," >> $DB_CREATE_1
echo "                '$DB_REDO_4B'" >> $DB_CREATE_1
echo "                size $REDO_F_SIZE K," >> $DB_CREATE_1
echo "          group 5 ('$DB_REDO_5A'," >> $DB_CREATE_1
echo "                '$DB_REDO_5B'" >> $DB_CREATE_1
echo "                size $REDO_F_SIZE K;" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "create rollback segment temp1 storage (initial 10K next 10K);" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "alter rollback segment temp1 online;" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "create tablespace rollback_segs datafile" >> $DB_CREATE_1
echo "      '$DB_ROLB_1'" >> $DB_CREATE_1
echo "      size $ROLB_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_ROLB_2'" >> $DB_CREATE_1
echo "      size $ROLB_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_ROLB_3'" >> $DB_CREATE_1
echo "      size $ROLB_F_SIZE M" >> $DB_CREATE_1
echo "      default storage ( initial $ROLB_SEG_SIZE K" >> $DB_CREATE_1
echo "                        next $ROLB_SEG_SIZE K" >> $DB_CREATE_1
echo "                        minextents 2" >> $DB_CREATE_1
echo "                        pctincrease 0" >> $DB_CREATE_1
echo "                        optimal $ROLB_SEG_OPTM K );" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "create rollback segment rs1 tablespace rollback_segs;" >> $DB_CREATE_1
echo "create rollback segment rs2 tablespace rollback_segs;" >> $DB_CREATE_1
echo "create rollback segment rs3 tablespace rollback_segs;" >> $DB_CREATE_1
echo "create rollback segment rs4 tablespace rollback_segs;" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "alter rollback segment rs1 online;" >> $DB_CREATE_1
echo "alter rollback segment rs2 online;" >> $DB_CREATE_1
echo "alter rollback segment rs3 online;" >> $DB_CREATE_1
echo "alter rollback segment rs4 online;" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "create tablespace tools datafile" >> $DB_CREATE_1
echo "      '$DB_TOOLS' size $TOOL_F_SIZE M" >> $DB_CREATE_1
echo "      default storage (initial $TOOL_SEG_INIT K" >> $DB_CREATE_1
echo "                        next $TOOL_SEG_NEXT K" >> $DB_CREATE_1
echo "                        pctincrease 0 );" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "create tablespace temp datafile" >> $DB_CREATE_1
echo "      '$DB_TEMP' size $TEMP_T_SIZE M" >> $DB_CREATE_1
echo "      default storage (initial $TEMP_SEG_INIT K" >> $DB_CREATE_1
echo "                        next $TEMP_SEG_NEXT K" >> $DB_CREATE_1
echo "                        pctincrease 0 );" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "create tablespace admin datafile" >> $DB_CREATE_1
echo "      '$DB_ADMIN' size $ADMIN_F_SIZE M" >> $DB_CREATE_1
echo "      default storage (initial $ADMIN_SEG_INIT K" >> $DB_CREATE_1
echo "                        next $ADMIN_SEG_NEXT K" >> $DB_CREATE_1
echo "                        pctincrease 0 );" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "create tablespace cape_objects datafile" >> $DB_CREATE_1
echo "      '$DB_USER_1' size $CAPE_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_USER_2' size $CAPE_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_USER_3' size $CAPE_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_USER_4' size $CAPE_F_SIZE M" >> $DB_CREATE_1
echo "      default storage (initial $CAPE_SEG_INIT K" >> $DB_CREATE_1
echo "                        next $CAPE_SEG_NEXT K" >> $DB_CREATE_1
echo "                        pctincrease 20 );" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
echo "create tablespace indexes datafile" >> $DB_CREATE_1
echo "      '$DB_INDX_1' size $INDX_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_INDX_2' size $INDX_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_INDX_3' size $INDX_F_SIZE M," >> $DB_CREATE_1
echo "      '$DB_INDX_4' size $INDX_F_SIZE M" >> $DB_CREATE_1
echo "      default storage (initial $INDX_SEG_INIT K" >> $DB_CREATE_1
echo "                        next $INDX_SEG_NEXT K" >> $DB_CREATE_1
echo "                        pctincrease 20 );" >> $DB_CREATE_1
echo "" >> $DB_CREATE_1
```

R2's Shell Tutorial

```

echo "" >> SDB_CREATE_1
echo "shutdown immediate" >> SDB_CREATE_1
echo "disconnect" >> SDB_CREATE_1
echo "exit" >> SDB_CREATE_1

#####
# Create Script Number 2
#####
echo "REM # Create Database (2) $ORACLE_SID on $UNIX system $THISHOST" >> SDB_CREATE_2
echo "REM # ....." >> SDB_CREATE_2
echo "REM # Script created by generate_db (c) R. H. Reepe" >> SDB_CREATE_2
echo "REM # Version 2.0 date `date`" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "connect internal" >> SDB_CREATE_2
echo "startup open" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "alter user system default tablespace tools temporary tablespace temp;" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
if [ $RESOURCE = 0 ] >> SDB_CREATE_2
then
    echo "create user capeprod identified by capeprod" >> SDB_CREATE_2
    echo "    default tablespace cape_objects" >> SDB_CREATE_2
    echo "    temporary tablespace temp" >> SDB_CREATE_2
    echo "    quota unlimited on cape_objects" >> SDB_CREATE_2
    echo "    quota unlimited on indexes;" >> SDB_CREATE_2
    echo "" >> SDB_CREATE_2
    echo "grant connect to capeprod;" >> SDB_CREATE_2
elif [ $RESOURCE = 1 ] >> SDB_CREATE_2
then
    echo "create user capeprod identified by capeprod" >> SDB_CREATE_2
    echo "    default tablespace cape_objects" >> SDB_CREATE_2
    echo "    temporary tablespace temp" >> SDB_CREATE_2
    echo "    quota unlimited on cape_objects" >> SDB_CREATE_2
    echo "    quota unlimited on indexes;" >> SDB_CREATE_2
    echo "" >> SDB_CREATE_2
    echo "grant connect to capeprod;" >> SDB_CREATE_2
fi
echo "" >> SDB_CREATE_2
echo "alter tablespace system" >> SDB_CREATE_2
echo "    default storage (initial    $$SYST_SEG_INIT K" >> SDB_CREATE_2
echo "    next          $$SYST_SEG_NEXT K" >> SDB_CREATE_2
echo "    pctincrease   5          );" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "alter rollback segment temp1 offline;" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
if [ $RESOURCE = 0 ] >> SDB_CREATE_2
then
    echo "create role estimator;" >> SDB_CREATE_2
    echo "grant select any table," >> SDB_CREATE_2
    echo "    delete any table," >> SDB_CREATE_2
    echo "    update any table," >> SDB_CREATE_2
    echo "    create session," >> SDB_CREATE_2
    echo "    select any sequence," >> SDB_CREATE_2
    echo "    insert any table to estimator;" >> SDB_CREATE_2
    echo "grant estimator to public;" >> SDB_CREATE_2
else
    echo "create role loader;" >> SDB_CREATE_2
    echo "grant select any table," >> SDB_CREATE_2
    echo "    delete any table," >> SDB_CREATE_2
    echo "    update any table," >> SDB_CREATE_2
    echo "    create session," >> SDB_CREATE_2
    echo "    select any sequence," >> SDB_CREATE_2
    echo "    insert any table to loader;" >> SDB_CREATE_2
    echo "grant loader to capeprod;" >> SDB_CREATE_2
    echo "grant dba to capeprod;" >> SDB_CREATE_2
fi
echo "@$ORACLE_HOME/rdbms/admin/catalog" >> SDB_CREATE_2
echo "@$ORACLE_HOME/rdbms/admin/catproc" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "create role dd_access;" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "@$ORACLE_BASE/SQL/DBA/data_dictionary_access" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "grant dd_access to system;" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
$account_name $system_coded $0 $$
. $account_string$$
acc_uid=`/usr/5bin/echo "$account" | cut -c1-6`
acc_pwd=`/usr/5bin/echo "$account" | cut -c8-`
echo "alter user $acc_uid identified by $acc_pwd;" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "connect $account" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "@$ORACLE_HOME/sqlplus/admin/pupbld" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
if [ $RESOURCE = 1 ] >> SDB_CREATE_2
then
    echo "connect capeprod/capeprod" >> SDB_CREATE_2
fi
echo "" >> SDB_CREATE_2
echo "@$ORACLE_HOME/rdbms/admin/catdsyn" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "connect internal" >> SDB_CREATE_2
echo "shutdown normal" >> SDB_CREATE_2
echo "startup mount" >> SDB_CREATE_2
echo "" >> SDB_CREATE_2
echo "alter database archivelog;" >> SDB_CREATE_2
echo "archive log start;" >> SDB_CREATE_2
echo "alter database open;" >> SDB_CREATE_2
echo "exit" >> SDB_CREATE_2

#####

```

R2's Shell Tutorial

```
if [ "$debug" = "on" ]
then
    echo "          Database Files"
fi
#####

echo ""
echo "Creating a slot $slot database using the following files"
echo ""
echo "$DB_SYST_1          $$SYST_F_SIZE Mbytes"
echo "$DB_SYST_2          $$SYST_F_SIZE Mbytes"
echo "$DB_TOOLS          $$TOOL_F_SIZE Mbytes"
echo "$DB_TEMP          $TEMP_T_SIZE Mbytes"
echo "$DB_ADMIN          $ADMIN_F_SIZE          Mbytes"
echo "$DB_CONT_1          1          Kbytes"
echo "$DB_CONT_2          1          Kbytes"
echo "$DB_REDO_1A          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_2A          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_3A          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_4A          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_5A          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_1B          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_2B          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_3B          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_4B          $REDO_F_SIZE Kbytes"
echo "$DB_REDO_5B          $REDO_F_SIZE Kbytes"
echo "$DB_ROLB_1          $ROLB_F_SIZE Mbytes"
echo "$DB_ROLB_2          $ROLB_F_SIZE Mbytes"
echo "$DB_ROLB_3          $ROLB_F_SIZE Mbytes"
echo "$DB_USER_1          $CAPE_F_SIZE Mbytes"
echo "$DB_USER_2          $CAPE_F_SIZE Mbytes"
echo "$DB_USER_3          $CAPE_F_SIZE Mbytes"
echo "$DB_USER_4          $CAPE_F_SIZE Mbytes"
echo "$DB_INDX_1          $INDX_F_SIZE Mbytes"
echo "$DB_INDX_2          $INDX_F_SIZE Mbytes"
echo "$DB_INDX_3          $INDX_F_SIZE Mbytes"
echo "$DB_INDX_4          $INDX_F_SIZE Mbytes"
echo ""
echo "$DB_ARCH [directory]"
echo "$DB_DUMP [directory]"
echo ""
echo "$DB_PFILE"
echo "$DB_CREATE_1"
echo "$DB_CREATE_2"
echo ""
echo "Database SID          $ORACLE_SID"
echo "SGA Size          $SGA_SIZE M"
echo "DB Block Buffers          $DB_BUFFER_SIZE (2K Blocks)"
echo "Open Cursors          $DB_BUFFER_SIZE"
echo ""

#####
if [ "$debug" = "on" ]
then
    echo "          Lets Just Doit"
    lpr $DB_PFILE
    lpr $DB_CREATE_1
    lpr $DB_CREATE_2
    exit
fi
#####

echo "Checking For Available Disk Space"

od1_size=`df -k $O_D_1 | tail -1 | /bin/nawk '{print($5)}' | cut -f1-1 -d%` # Used Space on Oracle Disk 1
od2_size=`df -k $O_D_2 | tail -1 | /bin/nawk '{print($5)}' | cut -f1-1 -d%` # Used Space on Oracle Disk 2
od3_size=`df -k $O_D_3 | tail -1 | /bin/nawk '{print($5)}' | cut -f1-1 -d%` # Used Space on Oracle Disk 3
od4_size=`df -k $O_D_4 | tail -1 | /bin/nawk '{print($5)}' | cut -f1-1 -d%` # Used Space on Oracle Disk 4

if [ $od1_size -gt $disk_full ] # If any disk is at limit size...
then
    echo "WARNING - Disk [ $O_D_1 ] is $od1_size % full" # Then I cannot fit a database in.
    db_create=no
elif [ $od2_size -gt $disk_full ]
then
    echo "WARNING - Disk [ $O_D_2 ] is $od2_size % full"
    db_create=no
elif [ $od3_size -gt $disk_full ]
then
    echo "WARNING - Disk [ $O_D_3 ] is $od3_size % full"
    db_create=no
elif [ $od4_size -gt $disk_full ]
then
    echo "WARNING - Disk [ $O_D_4 ] is $od4_size % full"
    db_create=no
else
    echo "There is space to create this database"
    db_create=yes
fi

if [ "$db_create" = "no" ]
then
    echo ""
    echo "          #####"
    echo "          ## Database will not be created ##"
    echo "          #####"
    echo ""
    echo "But init$ORACLE_SID.ora and create-$ORACLE_SID.sql scripts have"
    echo "been made available for your inspection in the dbs directory"
    echo ""
    echo "Database instance name is $ORACLE_SID"
    echo ""

```

R2's Shell Tutorial

```
s_stopping $0
exit
fi

db_exists=`grep $ORACLE_SID $ORATAB | wc -l | cut -c7-8`          # Get Database Existence
if [ $db_exists = 1 ]                                           # Does it already exist ?
then
  echo ""
  echo "Database Instance [ $ORACLE_SID ] found in [ $ORATAB ]"
  /usr/sbin/echo "Is this okay ? (y/n) \c"
  read answer
  echo ""
  if [ "$answer" = "n" ] || [ "$answer" = "N" ]
  then
    echo "Aborting"
    exit
  fi
else
  echo "Creating entry for [ $ORACLE_SID ] in [ $ORATAB ]"
  echo "$ORACLE_SID:$ORACLE_HOME:Y >> $ORATAB"                  # Register Database in ORATAB file
fi

cp $DB_PFILE $DB_PFI_BU                                         # Save backup of DB Parameter File
cp $DB_CREATE_1 $DB_CRE_BU_1                                     # Save backup of DB Create File
cp $DB_CREATE_2 $DB_CRE_BU_2                                     # Save backup of DB Create File
echo ""
echo "Saved Oracle Create File and Parameter File in:"
echo "    ->          $ORACLE_BASE/SQL/DB_CREATE"
echo ""
cd $ORACLE_HOME/dbs                                             # Go to Startup Directory

sqldba command="\@$DB_CREATE_1\"                                 # Run create script 1 in sqldba

echo ""
echo "Generate_DB Script 1 complete"
echo "Generate_DB Script 2 will now start in Silent Mode"
echo "Output will be spooled to the [generate] logfile"
echo "Use [logs generate] to monitor results"
echo "This terminal window will remain locked until Script 2 completes"
echo ""

sqldba command="\@$DB_CREATE_2\"      > $log0 2>&1             # Run create script 2 in sqldba & log

echo ""
echo "Updating SQL-Net 2.1 Files"
echo ""
#
#####
# Sql Net 2 Listener Stuff
#####
#
$ORACLE_BASE/scripts/NET2FILES/scan_tns_listener
$ORACLE_BASE/scripts/NET2FILES/sync_listener
$ORACLE_BASE/scripts/NET2FILES/sync_tnsname
#
#####

echo ""
echo "Generate_DB Script 2 complete"
echo "Generate_DB Terminated"
echo ""

s_stopping $0
```

Dedication

To the following people, I owe a great debt of gratitude. For their forbearance and patience whilst creating this tomb, a heartfelt thanks must go to Guy Footring, Rob Stubbings, and Andy Kucewicz. For testing my understanding and stretching my knowledge, I would like to thank both Peter Prenzel and Jürgen Schlösser. I would also like to say a very special thank you to Linda Low, without whose constant encouragement this landmark project would never have been undertaken or completed.

R2's Shell Tutorial

```

/etc
/usr/etc
/inference/art/emacs
/$PROJECT/NeWSprint/bin
/usr/openwin/bin
/usr/ccs/bin
/$PROJECT/DFrag/bin
$ORACLE_HOME/bin)

#####
# Check for Terminal Access - if yes, setup aliases #
#####
set answer = `tty` >& /dev/null
set answer = $status
if ($answer == 0) then
    # The rest of this file is inside the IF #

#####
# If its Rob Stubbings from ukf151
#
# and set up environment to remote display oracles mail messages at home #
#####
set rob1 = `finger | grep "$ME" | grep ukf151 | wc -l | cut -c1-8`
set rob2 = `finger | grep "$ME" | grep "254.254.2.3" | wc -l | cut -c1-8`
set rob = `expr $rob1 + $rob2`
if ( $rob >= 1 ) then
    setenv DISPLAY 254.254.254.254:0.0
    setenv PRINTER $REAL_PRINTER
endif

#####
# If its Richard Reepe from ukf068 - then set my erase char to CONTROL-H #
# and set up environment to remote display oracles mail messages at home #
#####
set richard1 = `finger | grep "$ME" | grep ukf068 | wc -l | cut -c1-8`
set richard2 = `finger | grep "$ME" | grep "254.254.3.2" | wc -l | cut -c1-8`
set richard = `expr $richard1 + $richard2`
if ( $richard >= 1 ) then
    setenv DISPLAY 254.254.254.254:0.0
    setenv PRINTER $REAL_PRINTER
    stty erase ^H
    set ws = "-Ws 585 281"
    # Window Size Params for Oracle Mailtools
    if ( "$THISHOST" == "thcsv01" ) set opt = "-Wp 0 4 \
        $ws -WP 5 320 -Wl thcsv01 -WL thcsv01 \
        -Wf 000 000 000 -Wb 255 255 000"
    if ( "$THISHOST" == "tc0295" ) set opt = "-Wp 10 24 \
        $ws -WP 5 390 -Wl tc0295 -WL tc0295 \
        -Wf 000 000 000 -Wb 100 255 255"
    if ( "$THISHOST" == "vacsv01" ) set opt = "-Wp 20 44 \
        $ws -WP 5 460 -Wl vacsv01 -WL vacsv01 \
        -Wf 000 000 000 -Wb 000 230 255"
    if ( "$THISHOST" == "dtsws070" ) set opt = "-Wp 30 64 \
        $ws -WP 5 530 -Wl dtsws070 -WL dtsws070 \
        -Wf 000 000 000 -Wb 000 180 255"
    if ( "$THISHOST" == "thcsv03" ) set opt = "-Wp 40 84 \
        $ws -WP 5 600 -Wl thcsv03 -WL thcsv03 \
        -Wf 255 255 255 -Wb 000 120 255"
    if ( "$THISHOST" == "ngcsv01" ) set opt = "-Wp 50 104 \
        $ws -WP 5 670 -Wl ngcsv01 -WL ngcsv01 \
        -Wf 255 255 255 -Wb 000 000 210"
    if ( "$THISHOST" == "cpd700" ) set opt = "-Wp 60 124 \
        $ws -WP 5 740 -Wl cpd700 -WL cpd700 \
        -Wf 255 255 255 -Wb 000 000 130"
    alias omail "toolwait mailtool -Wi $opt"
endif
```

R2's Shell Tutorial

```
#####
# If its Claire Harris from ukf153 - then set my erase char to CONTROL-H #
# and set my printer and display environment variables #
#####
set claire1 = `finger | grep " $ME " | grep ukf153 | wc -l | cut -c1-8`
set claire2 = `finger | grep " $ME " | grep "254.254.2.3" | wc -l | cut -c1-8`
set claire = `expr $claire1 + $claire2`
if ( $claire >= 1 ) then
    setenv DISPLAY 254.254.254.254:0.0
    setenv PRINTER $REAL_PRINTER
    stty erase
endif

#####
# If its Andrew Clark from fus122 or fus123 #
# and set up environment to remote display #
#####
set drew1 = `finger | grep " $ME " | grep fus122 | wc -l | cut -c1-8`
set drew2 = `finger | grep " $ME " | grep "254.254.254.254" | wc -l | cut -c1-8`
set drew = `expr $drew1 + $drew2`
if ( $drew >= 1 ) then
    setenv DISPLAY fus122.pd9.$COMPANY.com:0.0
    setenv PRINTER pr1953
    stty erase
endif

#####
# now some alias definitions #
#####
alias a alias
alias b ls -l /$COMPANY/thishost/u/oracle_data_1/BACKUPS
alias d 'df -k | grep ora'
alias h history
alias l ls -lisa
alias dba ~/scripts/sql_dba
alias s cd $ORACLE_BASE/scripts
alias S cd $ORACLE_BASE/SQL
alias doc cd $ORACLE_BASE/documentation
alias dbs cd $ORACLE_HOME/dbs
alias cron cd $ORACLE_BASE/cron
alias logs ~/scripts/read_logs
alias olds ~/scripts/read_old_logs
alias del ~/scripts/delete_script
alias distribute ~/scripts/distribute_file
alias disdir ~/scripts/distribute_directory
alias sql ~/scripts/sql_plus
alias sql2 ~/scripts/sql2_plus
alias access ~/scripts/create_database_locations
alias apply ~/scripts/data_ccar
alias report ~/scripts/report
alias unlock ~/scripts/unlock_process
alias process ~/scripts/completed_process
alias space ~/scripts/db_space
alias whos_using ~/scripts/whos_using
alias wots_up ~/scripts/wots_up
alias help ~/scripts/help
alias how ~/scripts/how
alias fun ~/scripts/functions
alias C '~/scripts/sccs_command C'
alias D '~/scripts/sccs_command D'
alias E '~/scripts/sccs_command E'
alias G '~/scripts/sccs_command G'
alias P '~/scripts/sccs_command P'
```

R2's Shell Tutorial

```
alias R                '~/scripts/sccs_command R'
alias menu             ~/scripts/script_menu
alias resource        ~/scripts/resource_maintenance
alias pdcrc           ~/scripts/create_pdcrc_file
alias product         '~/scripts/product;source /tmp/p_tmp'
alias list            'nl -ba -s}'
alias print           '~/scripts/print'
alias xprint          '~/scripts/xprint'
alias land            '~/scripts/land'
alias install         'shelltool -B Offset_x -Wl "Ora Inst" -WL "Ora Inst" &'
alias em              'emacs -bg snow1 &'
alias oracle-help    'cd $ORACLE_HOME; orainst/oradocm &'

endif                # Closing line from terminal test #
#####
# End of Terminal Access Setup #
#####
```

Next is the main **.profile** which is much shorter because it is now in several pieces for ease of management. This is followed by the other related **.profiles**. The functions are all grouped together in the section dedicated to functions.

Oracle's .profile

```
#####
# ORACLE's .profile (c) R. H. Reepe 8th March 1995 Version 3.0 #
#####
# 960419 RHR Restructured into separate .profiles

#####
# Add Database Name Reference Strings          #
#####
. $HOME/.profile_references

#####
# General Setup Information                    #
#####
export DEVELOPMENT_SERVER="ukf123"
export RECON_MASTER="fob_m01"
export RECON_SERVER="ukf124"
export EDITOR=vi
export UNIX=`/usr/bin/uname -r | cut -c1-1`
export THISHOST=`/usr/bin/uname -n`
export ORACLE_VERSION=7.1.6
export COMPANY=acme

#####
# Remaining General Profiles                  #
#####
. $HOME/.profile_paths
. $HOME/.profile_oracle
. $HOME/.profile_copyres
. $HOME/.profile_function

#####
# Count of Available Oracle Products (Versions of Oracle)  #
#####
export PRODUCT_COUNT=`ls -l $ORACLE_BASE/product | grep -c "`
```

Oracle's .profile_references

```
#####
# .profile_references (c) R. H. Reepe 17th May 1995 Version 1.0 #
#####
# This PROFILE contains mapping lists for servers and databases and is #
# used by several scripts to create and contact remote databases. The #
# format of this file is vital to Database Administration Integrity and #
# should be strictly adhered to for guaranteed continued operation. If #
# not set up correctly, Global Estimate Access will cease to work. #
# The data from this file is used primarily to populate the Master table #
# DATABASE_REFERENCES in all Master Databases. This is then copied to all #
# Production Databases as part of the copy_resources_db process for all #
# STATIC Data. Entries in DATABASE_LOCATIONS are based on the data stored #
# in DATABASE_REFERENCES as transformed and assembled by CAPE_ADMIN #
# Procedures and Functions, and CAPE_UTILITIES Procedures and Functions. #
# #
# REFERENCE_DATABASE = a list of single Words describing the DATABASE Type #
# used to populate the DATABASE_LOCATIONS table for #
# Global Estimate Access (GEA). #
# #
# REFERENCE_DBSERVER = a list of Database Host SERVERS which support a #
# CAPE Production Database. Used to populate the #
# DATABASE_LOCATIONS table. One of the KEY fields in #
# the DATABASE_REFERENCES table. #
# #
# REFERENCE_IPDOMAIN = a list of TCP/IP DOMAIN Names where Servers are #
# located on the Intranet. Used for file transfers. #
# #
# REFERENCE_LOCATION = a list of LOCATION mnemonics to uniquely identify #
# a Server Site. Used to populate DATABASE_LOCATIONS. #
# #
# REFERENCE_DBPREFIX = a list of 2 character strings for use by Oracle to #
# create the DATABASE_PREFIX strings used for GEA #
# HOME_DATABASE Indicators within CAPE. #
# #
# REFERENCE_DBSERIES = a list of 1 character numeric database SERIES flags #
# used to generate the last character of the database #
# prefix string used for GEA HOME_DATABASE Indicators #
# within CAPE. #
# #
# REFERENCE_DB_SIZES = a list of integer Generic_Database_Unit SIZE #
# parameters used by GENERATE_DB when setting the #
# sizes of database TABLESPACES. This value should #
# be increased by 50% for a server who's database #
# has flagged a space shortage. #
# #
# REFERENCE_COUNTRY = a list of COUNTRY Names to locate the CAPE #
# Database with its $COMPANY Operation. Used to populate #
# the DESCRIPTION Field of DATABASE_LOCATIONS #
# #
# REFERENCE_BUILDING = a list of $COMPANY BUILDING Names to locate the CAPE #
# Database with its $COMPANY Operation. Used to populate #
# the DESCRIPTION Field of DATABASE_LOCATIONS #
# #
# This file is split up into sections for each Country. The Variables are #
# in lower case and are prefixed by a CAPE Country Code and identified by #
# a single letter code for usage: #
# #
# MapChar ListName Domain Description #
#-----#
# h = REFERENCES_DBSERVER (Host) Host IP Alias Name #
# d = REFERENCES_IPDOMAIN (Domain) Host Domain Name #
# l = REFERENCES_LOCATION (Location) Database_SID (Chars 1-4) #
# f = REFERENCES_DBPREFIX (preFix) Database_ID (Chars 1&2) #
# s = REFERENCES_DBSERIES (Series) Database_ID (Char 3) #
# b = REFERENCES_BUILDING (Building) DATABASE_LOCATIONS (Descr) #
# c = REFERENCES_COUNTRY (Country) DATABASE_LOCATIONS (Descr) #
# z = REFERENCES_DB_SIZES (siZe) Generic Database Unit Mb #
# REFERENCES_DATABASE (TYPE) generate_db TYPES #
# #
# When creating new Country sections follow this format and place Country #
# in its alphabetic location in the list. Then at the end of the file add #
# the variable names into the lists which create the exported REFERENCE #
# variables (in the correct position or internal mapping will be wrong). #
# #
# The IP aliases must = HOSTNAME from `uname -n` #
# #
# All names in these lists must be delimited by SPACE's Only. No TAB's are #
# allowed in this format. Server Names can be any length but must agree #
```

R2's Shell Tutorial

```
# with the IP Alias for the server. Location Names must be 4 characters #
# long and be unique globally. Flag Names must be 2 characters long and be #
# unique globally. #
#####
# 960415 RHR Genesis
# 960515 RHR Added DB_SIZES for GENERATE_DB usage
# 960618 RHR Added DBSERIES for GEA Database_ID usage
# 960619 RHR Re-Ordered Variables for ease of maintenance
# 960731 RHR Added Domains

#####
# EXPORTED VARIABLES #
#####
export REFERENCE_DATABASE
export REFERENCE_DBSERVER
export REFERENCE_IPDOMAIN
export REFERENCE_IPDOMAIN
export REFERENCE_LOCATION
export REFERENCE_DBPREFIX
export REFERENCE_DBSERIES
export REFERENCE_BUILDING
export REFERENCE_COUNTRY
export REFERENCE_COUNTRY
export REFERENCE_DB_SIZES

#####
# Deutschland #
#####
de_h_map="duf021 "
de_d_map="black.$COMPANY.com "
de_l_map="fok_ "
de_f_map="de "
de_s_map="3 "
de_b_map="Black_House "
de_c_map="Germany "
de_z_map="11 "

#####
# Espanya #
#####
es_h_map="esf011 "
es_d_map="purple.$COMPANY.com "
es_l_map="fop_ "
es_f_map="es "
es_s_map="3 "
es_b_map="Purple_house "
es_c_map="Spain "
es_z_map="11 "

#####
# Gt Britain #
#####
gb_h_map="ukf123 ukf124 ukf125 ukf171 ukf212"
gb_d_map="blue.$COMPANY.com green.$COMPANY.com red.$COMPANY.com yellow.$COMPANY.com white.$COMPANY.com"
gb_l_map="tst_ dev_ for_ foy_ fow_ "
gb_f_map="ts dv gb du jw"
gb_s_map="3 3 3 3 3"
gb_b_map="Blue_House Green_House Red_House Yellow_House White_House"
gb_c_map="Britain Britain Britain Britain Britain"
gb_z_map="11 11 11 11 11"

#####
# United States #
#####
us_h_map="cpd700 tc0295 "
us_d_map="abc.$COMPANY.com bbd.$COMPANY.com "
us_l_map="foa_ fob_ "
us_f_map="us lv "
us_s_map="3 3 "
us_b_map="Building_Nine Building_Seven "
us_c_map="America America "
us_z_map="40 30 "

#####
# Global Mapping #
#####
REFERENCE_DATABASE="Administration Development Evaluation Guidance Master Production Test"

REFERENCE_DBSERVER="$de_h_map $es_h_map $gb_h_map $us_h_map "
REFERENCE_IPDOMAIN="$de_d_map $es_d_map $gb_d_map $us_d_map "
REFERENCE_LOCATION="$de_l_map $es_l_map $gb_l_map $us_l_map "
REFERENCE_DBPREFIX="$de_f_map $es_f_map $gb_f_map $us_f_map "
REFERENCE_DBSERIES="$de_s_map $es_s_map $gb_s_map $us_s_map "
REFERENCE_BUILDING="$de_b_map $es_b_map $gb_b_map $us_b_map "
REFERENCE_COUNTRY="$de_c_map $es_c_map $gb_c_map $us_c_map "
REFERENCE_DB_SIZES="$de_z_map $es_z_map $gb_z_map $us_z_map "
```

Oracle's .profile_paths

```
#####
# .profile_paths (c) R. H. Reepe 19th April 1996 Version 1.0 #
#####
# 960419 RHR Genesis

#####
# Selected Short Paths for SQL SCRIPT Use #
#####
export PROJECT=/ $COMPANY/ $THISHOST/ unix/ cen
export ORACLE_BASE=$PROJECT/oracle_base
export ORACLE_HOME=$ORACLE_BASE/product/ $ORACLE_VERSION
export ORACLE_BIN=$ORACLE_HOME/bin
export ORACLE_SQL=$ORACLE_BASE/SQL
export ORACLE_SCRIPTS=$ORACLE_BASE/scripts
export ORACLE_CRON=$ORACLE_BASE/cron
export ORACLE_LOG=$ORACLE_BASE/logfiles
export ORACLE_LOD=$ORACLE_BASE/loaders
export ORACLE_PAC=$ORACLE_SQL/PACKAGES
export ORACLE_PRO=$ORACLE_SQL/PROCEDURES
export ORACLE_FUN=$ORACLE_SQL/FUNCTIONS
export ORACLE_DBA=$ORACLE_SQL/DBA
export ORACLE_MIG=$ORACLE_BASE/MIGRATION
export ORACLE_GEN=$ORACLE_BASE/GENERIC

#####
# Main Command Paths for CRON & SCRIPT files #
#####
export PATH=$PATH:/bin:/usr/sbin
export PATH=$PATH:$ORACLE_BASE:$ORACLE_HOME:$ORACLE_BIN
```

The next two files called by .profile are not reproduced as they have no relevance for other users. See the section on Functions for a full listing of some very useful code.

Oracle's .profile_READ_ME

Lastly here is the .profile_readme which contains information about all the .profiles for this User ID and how they are used. This file allows all users to be kept up-to-date with new developments as they are added and how best to take advantage of the available facilities.

```
#!/bin/sh
#####
# .profile_READ_ME (c) R. H. Reepe 29th February 1996 Version 1.0 #
#####
exit
```

There are a group of files belonging to Oracle, situated in the oracle_base directory (\$HOME for UserID oracle) which start with the characters ".profile". These files are read into Bourne Shell scripts with the dot (.) command at the start of each Bourne Shell script. The files contain information (paths, environment variables, subroutine definitions, etc.) which the shell scripts can use. The profiles are organised as follows:

1	.profile	General Environment Variables
2	.profile_paths	Commonly Used Paths
3	.profile_oracle	Oracle UserID/Password Coded Strings
4	.profile_copyres	Table and Key information for Oracle
5	.profile_references	Database & Server Naming Protocol
6	.profile_subroutine	Bourne Shell Subroutine Suit
7(a-z)	.profile_fun_xxxx	Bourne Shell Subroutine Files
8	.profile_READ_ME	This Document
9	.profile_sql_net_2	Additional DB utilities for SQL-Net 2.0

Currently, all the profiles are read into all scripts at run time. This may change in the future if the size of the profiles grows too large and startup time for the scripts starts to slow down. The special profile for copyres only really needs to be available inside the copy_resources_db script and so could be moved outside of the general execution frame for other scripts. However, there is a plan to create an Oracle Procedure to handle the copy_resources_db functionality at some future date. There is also the distinct possibility that the subroutines for the ftp process may be broken out into a new profile containing networking functions only, as they are quite large and must put additional time into all scripts not using them directly. There are some simple guidelines to be followed when creating or editing the profiles:

- 1 Only Exported Variables will show up in the Parent Shell
- 2 All Exported Variables should be in UPPER_CASE
- 3 All External Subroutines should start with s_
- 4 Group Like Objects together
- 5 Files should be Identical on All Servers

A similar set of guidelines exists for creating and editing Bourne Shell scripts which is as follows:

- 1 All Local Variables should be in lower_case
- 2 All Local Subroutines should be declared before calling
- 3 All Local Subroutines should NOT start with s_
- 4 No amount of internal documentation is too much
- 5 Files should be Identical on All Servers

=====
INDIVIDUAL PROFILE DETAILS
=====

.profile

- 1 Called by All Bourne Shell Scripts by using [. \$HOME/.profile]
- 2 Contains calls to other General Profiles.
- 3 Contains General Setup Variables for CAPE Databases.

R2's Shell Tutorial

----- .profile_paths -----

- 1 Used by All Scripts to locate Paths where other Objects live.
- 2 Contains a list of Oracle Paths.

----- .profile_oracle -----

- 1 Used by All Scripts to "Aquire" UserID's/Password for Oracle.
- 2 Contains Coded Strings of UserID's & Passwords - See Oracle Security.
- 3 Strings contain Control Characters - Do NOT Corrupt this file.

----- .profile_copyres -----

- 1 Used by the copy_resources_db cron script to find the list of Tables to copy.
- 2 Contains a list of tables and some primary keys (machines only).
- 3 Structure is self evident. Update when new tables are required.

----- .profile_references -----

- 1 Used by All Scripts that need to know about Servers and Databases.
- 2 Contains mapping lists for Databases, Servers, Sites, etc.
- 3 See embeded documentation for details.

----- .profile_functions -----

- 1 Used by All Scripts during execution.
- 2 Contains the names of all Function Files loaded during script execution.
- 3 Each fun file contains related Functions and Subroutines.

----- .profile_fun_xxxx -----

- 1 Used by All Scripts during execution.
- 2 Suit of files containing sets of related Functions and Subroutines.
- 3 Be carefull to only call functions after they are defined when nesting.
- 4 For a full listing of available funtions type fun at the OS prompt.
- 5 Current File List:

a	.profile_fun_date	Date and Time Utilities
b	.profile_fun_file	File Utilities
c	.profile_fun_dbas	Database Utilities
d	.profile_fun_menu	Menu Utilities
e	.profile_fun_strg	Text String Utilities
f	.profile_fun_move	FTP and Mail Utilities
g	.profile_fun_util	Other Utilities

----- .profile_READ_ME -----

- 1 This File!
- 2 Add to it if required - It's no good if it's out of date!

----- .profile_sql_net_2 -----

- 1 Used by All Scripts during execution.
- 2 Dedicated subroutines for SQL-Net 2.0 support.
- 3 Will be integrated into .profile_sub_dbas when complete.

