AIX Ver. 4
Korn Shell Programming
(Course Code AU23)

Student Notebook

ERC 2.3

IBM Learning Services
Worldwide Certified Material

Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AT	IBM
OS/2	BookMaster	RISC System/6000
RS/6000		

Linux is a registered trademark of Linus Torvalds in the United States and other countries.

Windows Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

November 2000 Edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an "as is" basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk. The original repository material for this course has been certified as being Year 2000 compliant.

Contents

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AT	IBM
OS/2	BookMaster	RISC System/6000
RS/6000		

Linux is a registered trademark of Linus Torvalds in the United States and other countries.

Windows Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Course Description

AIX Ver. 4 Korn Shell Programming

Duration : 5 days

Purpose

This course will teach you how to use shell scripts and utilities for practical system administration of the IBM RISC System/6000.

Audience

Support staff of AIX® for RISC System/6000.

Prerequisites

An understanding of programming fundamentals: variables, flow control concepts such as repetition and decision. A working knowledge of AIX including the use of the vi editor, find and grep commands. Students without this experience should attend *AIX Version 4 Basics Plus*.

Objectives

After completing this course, students should be able to:

- Distinguish Korn Shell specific features
- Use utilities such as sed and awk to manipulate data
- Understand system Shell Scripts such as /etc/shutdown
- Write useful Shell Scripts to aid system administration.

Contents

- Basic Shell concepts
- Flow control in a Shell Script
- Functions and typeset
- Shell features such as arithmetic and string handling
- Using regular expressions
- Using sed, awk and other AIX utilities.

Practical Work

This course has a high practical content.

Agenda

Course Times: 09:30 - 17:00 (16:00 on the last day)

Day 1

Course and student introductions
Unit 1 - Basic Shell Concepts
Lab 1 - Introduction to Labs
Unit 2 - Variables
Lunch
Lab 2 - Using Shell Basics
Unit 3 - Return Codes and Traps

Day 2

Lab 3 - Testing
Unit 4 - Flow Control
Lunch
Lab 4 - Shell Programming Constructs
Unit 5 - Shell Commands
Lab 5 - Shell Commands and Features (part 1)

Day 3

Lab 5 - Shell Commands and Features (part 2)
Unit 6 - Arithmetic
Lab 6 - Shell Arithmetic
Lunch
Unit 7 - Korn Shell Types, Commands and Functions
Lab 7 - Typeset and Functions

Day 4

Unit 8 - More on Shell Variables
Lab 8 - More on Shell Variables
Lunch
Unit 9 - Regular Expressions and Text Selection Utilities
Lab 9 - Regular Expressions and Data Selection
Unit 10 - Utilities for Personal Productivity (part 1 - sed only)

Day 5

Unit 10 - Utilities for Personal Productivity (part 2)
Lab 10 - Personal Productivity
Unit 11 - The AWK Program
Lunch
Lab 11 - Using Awk
Unit 12 - Putting It All Together
Lab 12 - Getting It Together (including discussion on Instructor given example)
Unit 13 - Good Practices and Review
Close

Unit 1. Basic Shell Concepts

What This Unit Is About

This unit introduces the Korn Shell and its environment.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Recognize file types
- Identify metacharacters
- Use various quoting mechanisms
- Redirect file input and output

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

To review basic Shell concepts in order to:

- Describe the AIX Shells
- Use the AIX file-system
- Create a Shell Script
- Use metacharacters
- Use I/O redirection
- Use pipes and tees
- Group commands
- Run background processes
- Use Korn Shell job control
- Use command line recall and editing

Figure 1-1. Objectives AU232.3

Notes:

Shells

What is a Shell ?

- User interface to AIX
- Command interpreter
- Programming language

AIX Shells:

• Korn	- ksh	
• Bourne	- bsh	
• Restricted	- Rsh	
• C	- csh	
• Trusted	- tsh	
• POSIX	- psh	
• Default	- sh	*link to ksh in AIX V4*
• Remote	- rsh	

Figure 1-2. Shells AU232.3

Notes:

Any of the AIX Shells can be the initial login Shell for a user. Each has different features and syntax. Shells have some builtin commands which we will cover in later units. The AIX operating system provides a number of useful commands that are available from all Shells — examples of these will appear in this and later units.

The Korn Shell adds C Shell features to the Bourne Shell, to produce the most user-friendly and powerful Shell. It is also faster than the other Shells. The Korn Shell is more recent than the other shells, but retains backward compatibility with the Bourne Shell. David G. Korn wrote the Korn Shell at AT&T's Bell Labs (now Lucent) where it is now widely used.

Bourne Shell is the oldest Shell, it was written at AT&T's Bell Labs by Steven Bourne.

Restricted Shell provides a limited sub-set of the commands in Bourne Shell:

- you can't change your working directory
- you may not run operating system commands unless they are in the working directory
- the command search path cannot be changed
- re-direction is not allowed

The C Shell has a completely different syntax to Bourne Shell. It provides some advanced features such as job-control and command line editing. It was written by Bill Joy at Berkeley University of California. It's primary use is as an interactive shell and is not usually used in writing shell scripts.

The Trusted Shell is a sub-set of the Korn Shell, but it is AIX-specific, and is one of the enhanced security features of AIX Version 3:

- only "trusted" and Shell builtin commands can be executed
- the internal field separator characters cannot be reset
- functions may not be defined
- there is no command history
- the command search path is fixed in a special start-up profile file (/etc/tsh_profile)

The IEEE POSIX 1003.2 Shell and Utilities Language Committee report is the Open Systems definition of a Shell. The Korn Shell conforms to this document. A POSIX Shell is implemented under AIX Version 4 as a link to the Korn Shell.

The default login Shell for each user (in */etc/passwd*) is the */bin/ksh* Korn Shell. Bourne Shell is the default login Shell for older UNIX systems, and early versions of AIX.

The default Shell is */bin/sh*. For AIX Version 3 this was a link to the */bin/bsh* Bourne Shell program, in AIX Version 4 it is a link to /bin/ksh the Korn Shell.

A Remote Shell is used to login via a remote terminal, it uses the user's default login Shell.

This course will concentrate on the Korn Shell, pointing out differences from the Bourne Shell.

POSIX is Portable Operating System Interface — Xopen.

Directories

The file-system comprises directories in a hierarchical structure

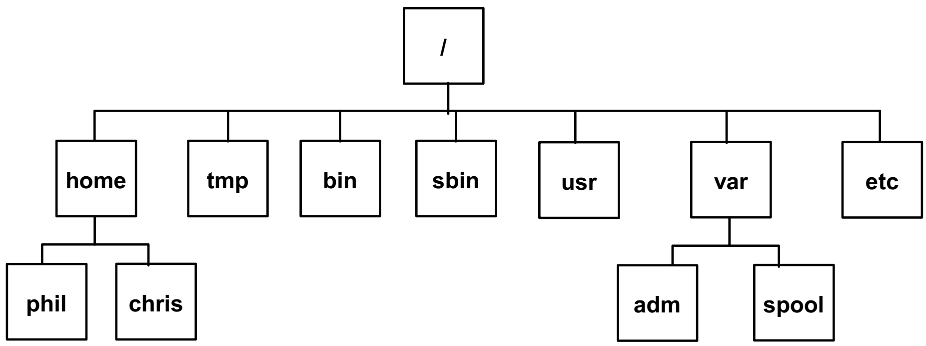


Figure 1-3. Directories AU232.3

Notes:

Each user on the system has a home directory with their portion of the tree underneath: like */home/phil* for user phil. In AIX Version 3.2, */home* replaces */u* in Version 3.1.

Below is a summary table of commands to manipulate the File-System.

Command:	*Argument*	Function:
`mkdir`	*directory*	Create new directory *"directory"*
`rmdir`	*directory*	Delete empty directory *"directory"*
`rm -r`	*directory*	Delete directory *"directory"* and any sub-directories
`cd`	*directory*	Change working directory to *"directory"*
`ls -ld`	*directory*	Give a long listing of *"directory"* - shows permissions, owner, etc.
`pwd`		Print working directory - where you are in the tree now
`mv`	*old* *new*	Rename a file or directory - *"new"* can be a new file name, or a directory in which to place the file
`cp`	*old* *new*	Copies a file to a new name
`ln`	*name* *copy*	Creates another name without copying the contents

The current directory is referred to by "." or the "." notation, and is used to specify a relative pathname to a directory or file from the current directory e.g. from */home*, *./chris* refers to Chris's home directory. Entering *cd* with no directory, changes the working directory to your home directory.

To refer to the parent of the current directory (go up a layer) we use the ".." notation: e.g. from */home/chris*, *../phil* is Phil's home directory.

The Korn Shell provides *cd* and *pwd* as builtin commands. AIX provides *pwd* as an operating system command. Additional features are provided with the Korn Shell *cd*:

`cd -` changes to the last working directory

`cd old new` replaces the string "old" with "new" in the current directory pathname, and tries to change directory to the resultant path. e.g. if /home/pat is the working directory, *cd pat chris* will change to /home/chris.

A File

Definition:

- collection of data, located on a portion of a disk.
- stream of characters or a "byte stream".

No structure is imposed on an ordinary file by the operating system.

Examples:

- Binary executable code – /bin/ksh
- Text data – /etc/passwd
- C program text – /home/john/prog.c
- Device special file – /dev/null
- Directory special file – /home

`$ file filename` ***– to find out which file type***

Figure 1-4. A File AU232.3

Notes:

Directories and devices are known as "special files" — the operating system controls their use.

Some other operating systems impose a record structure on all files — AIX does not have this restriction. You can have whatever you like in an ordinary file.

One special file that we'll be using a lot is */dev/null* — this is a bottomless pit where output can be directed if you want to lose it.

The *file* command can be used to find out what type a particular file is: i.e. binary executable, C program text, etc.

AIX File Names

- Should be descriptive of the content
- Are case-sensitive
- Should use only alphanumeric characters:

  UPPERCASE   lowercase   digits
  #   .   @   -   _

- Should not begin with "+" or "-" sign
- Should not contain embedded blanks or tabs
- Should not contain shell "special" characters:

```
 *   ?   >   <   /   ;   &   !   ~
 [   |   ]   $   \   '   "   `   {   }   (   )
```

Figure 1-5. AIX File Names AU232.3

Notes:

Remember *".filename"* files (dot files) are hidden from the normal *ls* and *li* commands, unless you use the *-a* option, or you are root.

Unlike DOS, AIX does not impose limitations on file name structure — you can have a 20 character file name with a ".*pat*" on the end if that takes your fancy.

There is a limit of 256 characters on the length of a Shell command line, and 255 characters on file names. As complicated and lengthy commands are sometimes necessary, it is usually wise to avoid very long file names.

What is a Shell Script?

- A readable text file which can be edited with a text editor
  - /usr/bin/vi *shell_prog*
- Anything that you can do from the Shell prompt
- A program, containing:
  - System commands
  - Variable assignments
  - Flow control syntax
  - Shell commands

and Comments !

Figure 1-6. What Is a Shell Script? AU232.3

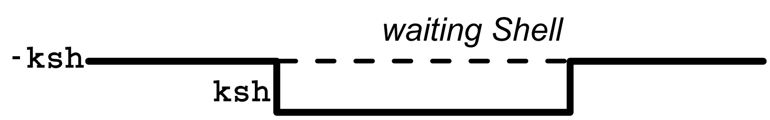
Notes:

The first line of a Shell Script can be read as an instruction to the Shell to run the script in a new specified type of Shell. This ensures that scripts are correctly run when you have switched your login to another Shell type.

#]/usr/bin/ksh as the first line ensures that the script is always run in the Korn Shell.

Invoking Shells

`$ ksh`	*begins a new Shell, interrupting the current one*
`$ ksh -c commands`	*runs commands in a Shell*
`$ ksh -r`	*starts a restricted Shell*



`$ exec ksh`	*terminates the current Shell and replaces with new Shell*

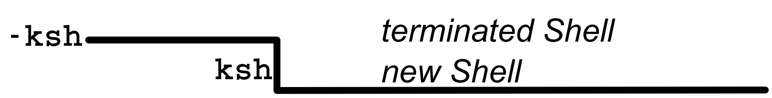


Figure 1-7. Invoking Shells AU232.3

Notes:

There are many options for invoking the Korn Shell, these are described fully in Unit 5. The Bourne Shell shares the options shown above with the Korn Shell.

With the "*-c*" option, multiword *commands* must be enclosed in quotes, so that they are treated logically as a single word.

A waiting shell is "sleeping" until its new shell signals that it has completed.

The *exec* command is a Shell builtin command.

Invoking Scripts

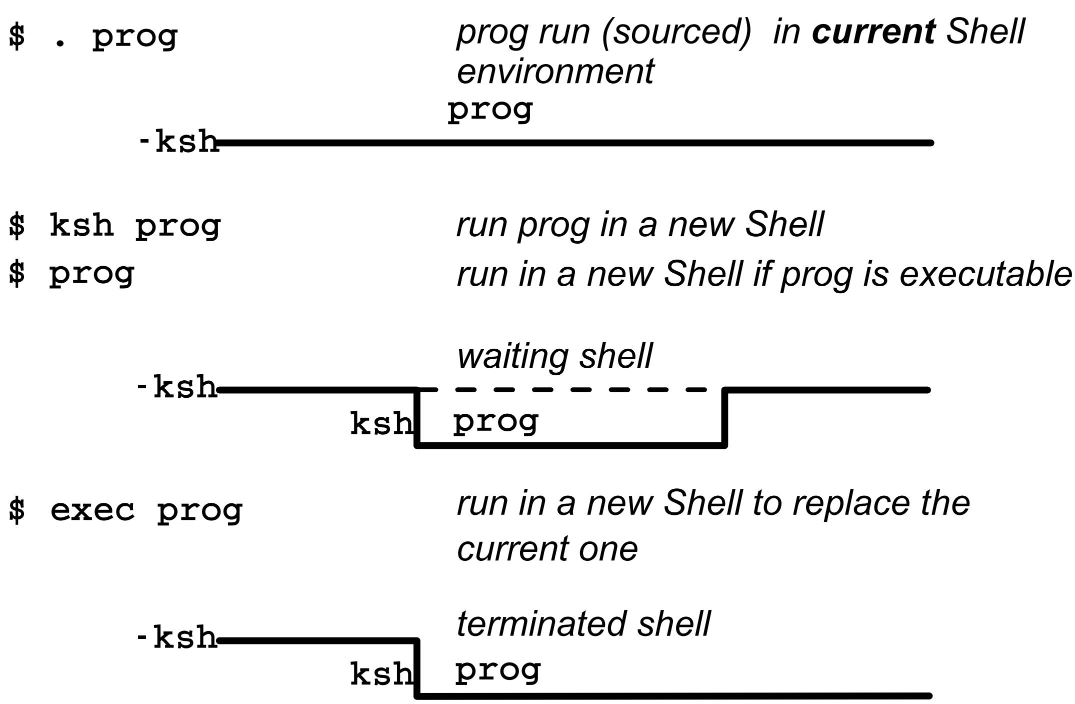


Figure 1-8. Invoking Scripts AU232.3

Notes:

The "." method (sourcing) causes the entire "*prog*" file to be read by the Korn Shell before it executes any of it. Such pre-processing may have unexpected side-effects. Other methods of invoking Scripts execute each line of code as it is read in.

A "./" in front of a file name refers to the current directory — that shown by *pwd*.

An executable Shell Script is executed by the user's login Shell, just like any operating system command.

Files can be made executable with the *chmod* command: e.g.

```
chmod +x prog
```

Korn Shell Configuration Files

Invoking the Korn Shell sources:

/etc/environment	Sourced by all AIX processes
/etc/profile	Sourced by login Shells
.profile	Login Shells source this file in the user's home directory
ENV file	A resource file listed in the ENV Environment Variable will be sourced by Korn Shells

time

Each new **explicit** Korn Shell sources the ENV file again

Figure 1-9. Korn Shell Configuration Files AU232.3

Notes:

If you use the Korn Shell as your login shell, your *.profile* file should contain settings for ENV. For example, it is typical to include the following lines in the script (explained in the next Unit):

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

For "privileged" Korn Shells, run with the "*-p*" option, the user's *.profile* and ENV files are replaced by */etc/suid_profile*. A "privileged" Shell is automatically invoked if your effective user id (UID) is different from your real UID, or your effective group (GID) is different from your real GID.

The AIX Windows Common Desktop Environment (CDE) provides access to Korn Shell windows. Normally these are not login shells. A .dtprofile file will be sourced if found in the home directory.

The Trusted Shell uses */etc/tsh_profile* in place of */etc/profile* and the user's *.profile* file.

The C Shell sources *.login* and *.cshrc* files in the user's home directory, instead of */etc/profile* and the users' *.profile* and *.kshrc* files.

Only Korn Shells source the ENV file. You invoke an explicit shell when you use the Korn Shell directly or explicitly. For example when you use commands like

```
ksh, ksh prog, ksh -c "commands"
```

When you run a program (other than by the dot method) that has the special comment "*#]/usr/bin/ksh*" as its first line, you also invoke an **explicit** Shell.

Another common file used is *.exrc*. This file contains commands used to control your vi editor environment. For example:

```
set showmode
set tabstop=4
ab IBM International Business Machines, Inc.
```

in your .exrc file. You need to use the colon before the command in the *vi* interactive form of the command.

What Are Metacharacters?

Characters with special meaning
- 3 types
  - Wildcard (or expansion)
  - Korn Shell
  - Quoting

- Shell processes metacharacters before executing a command

- There are several different Shell metacharacters

- Metacharacters can be mixed

They can be turned off by Shell options

Figure 1-10. What Are Metacharacters? AU232.3

Notes:

Metacharacters do not represent themselves. The three types are a way of classifying the metacharacters. Wildcards are the most commonly used (like *, ?). Korn Shell uses metacharacters, like ? and +. The third type are quotes like double, single and the \ escape character.

Unit 5 shows how wildcard metacharacters can be turned off using Shell options.

Wildcard Metacharacters

Metacharacters that form patterns that are expanded into matching filenames from the current directory

*****	-	Match any number of any characters
?	-	Match any single character
[abc]	-	Match a single character from the bracketed list
[!az]	-	Match any single character except those listed
[a-z]	-	Inclusive range for a list

Character Equivalence Classes can be used in place of range lists, to avoid National Language collation problems:

[**[:upper:]**]	-	range list of all upper case letters
[**[:lower:]**]	-	all lower case letters: a, b, c,... z
[**[:digit:]**]	-	digits: 0, 1, 2,... 9
[**[:space:]**]	-	spacing characters: tab, space, etc.

Figure 1-11. Metacharacters AU232.3

Notes:

Filenames beginning with a "." must be matched explicitly, with a "." as the first character in your pattern.

There are many more Character Equivalence Classes: **[:alpha:], [:alnum:], [:cntrl:], [:graph:], [:print:], [:punct:], [:xdigit:]** and **[:blank:]**. Further description of these is in the AIX Commands Reference manual, under *ksh*, *bsh*, *csh*, and especially *ed*.

Commands and utilities such as *grep*, *sed* and *awk* also use pattern matching metacharacters and Character Equivalence Classes. These have similar functions but are not identical (see units 9, 10 and 11):

*****	to match any number of the preceding character (so it must always follow something),
.	the dot matches any single character, rather than the **?**,
[^ab]	with a **^** in place of a **!** to signify an exclusion list,
^	can be used to signify the beginning of a line,
$	will signify the end of a line.

Take care not to be confused using **sed** and **awk!**

Sample Directory

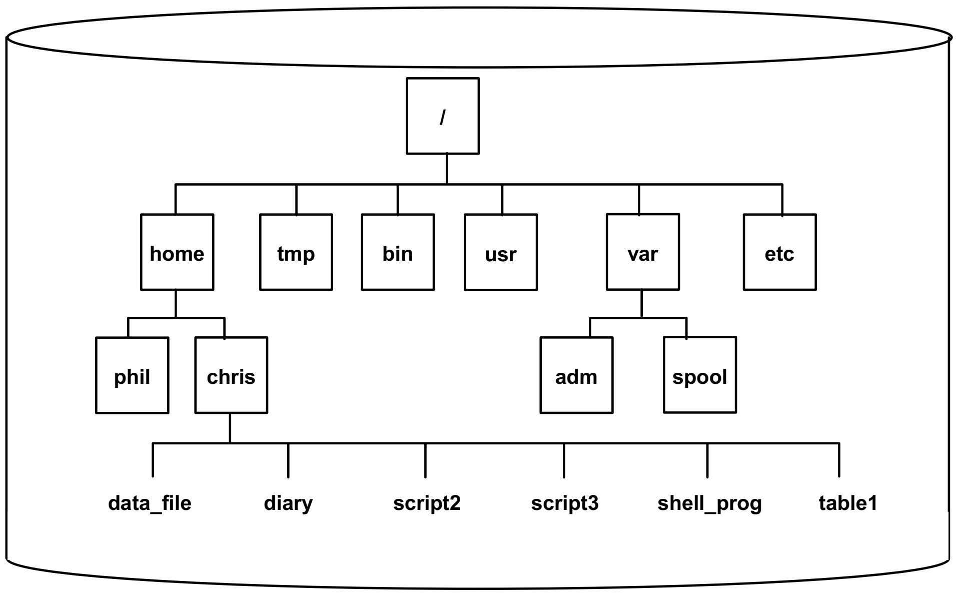


Figure 1-12. Sample Directory AU232.3

Notes:

These files will be used for the examples of metacharacter file name expansion on following pages.

Expansion Examples

`$ rm d*y`	*removes the diary file*
`$ file script*`	*identifies script2 and script3*
`$ head script[345]`	*displays the top lines of script3*
`$ more script[3-6]`	*displays script3 screen by screen*
`$ tail script[!12]`	*displays the last lines of script3*

Now your turn...

```
$ touch ?a*

$ pg [st][ah]*

$ lpr [a-z]*t[0-9]
```

Figure 1-13. Expansion Examples AU232.3

Notes:

Assume the current directory is */home/chris*.

Korn Shell Metacharacters

The Korn Shell can match multiple patterns

`*(pattern|pattern...)`	zero or more occurrences
`?(pattern|pattern...)`	zero or one occurrence
`+(pattern|pattern...)`	one or more occurrences
`@(pattern|pattern...)`	exactly one occurrence
`!(pattern&pattern...)`	anything except

One or more patterns, separated with "|" for "or", "&" for "and"

Examples:

`*([0-9])`	*0 or more consecutive digits*
`?(warning)`	*0 or 1 occurence of "warning"*
`+([[:upper:]]|[a-z])`	*1 or more consecutive letters*
`@([0-9]|abc)`	*1 digit or "abc"*
`!(err*&fail*)`	*Word cannot start with "err" or "fail"*

Figure 1-14. Korn Shell Metacharacters AU232.3

Notes:

Notice the & and | combination. Since you want to check two conditions (words not starting with err or fail) the logic needed is NOT ... AND ... . But Boolean logic (the !, & and |) mean that not X or not Y is given by not (X and Y).

Quoting Metacharacters

Stops normal Shell metacharacter processing, including metacharacter expansion

- To form strings

"double quotes"	group characters into a string, and allow variable and command substitution

- To form literal strings

'single quotes'	remove any special meaning for the characters within them

- For a literal character

\character	removes the special meaning of the character following the \

Figure 1-15. Quoting Metacharacters AU232.3

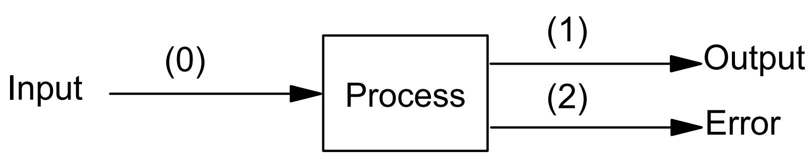
Notes:

In Unit 2 we shall see how to refer to variables, and in Unit 7 we shall look at command substitution.

Where \ is nested inside double quotes, it only removes the special meaning of four characters: \, `, " and $.

Process I/O

- Every process has a file descriptor table associated with it



File descriptor table

Defaults	0<	Standard in	- keyboard
	1>	Standard out	- screen
	2>	Standard error	- screen
User-defined	3		
	⋮		
	9		

Figure 1-16. Process I/O AU232.3

Notes:

You can define how the file descriptors 3 to 9 are handled. You might want to use descriptor 3 to output to a named file, while 4 outputs to a printer device file. Remember that your screen is addressed through its device file, e.g. */dev/tty0*, for both reading of input and displaying of output.

Remember that the device file */dev/tty* always refers to your keyboard or screen.

The defaults for the first three file descriptors can be changed as we will see next...

Input Redirection

Redirecting standard input from a file: <

command < filename

```
$ mail gene
Subject: Hello
A letter to see if you are still with us.
<Ctrl-d>
$ _

$ mail -s "Hello" gene < letter
$ _
```

Input may also be given inline. This is called a HERE document.

command << END
text
…
END

Figure 1-17. Input Redirection AU232.3

Notes:

In this example, the file *letter* has been created using an editor such as */usr/bin/vi*.

In a Shell Script the first method could not be used, because the mail command takes its input from standard input by default (but see below also).

In the second example the file descriptor "`0`" is changed so that input is taken from the named file. It is possible to write "*0*<", but the file descriptor number is usually omitted.

HERE documents are seen in scripts. You could use the HERE document syntax for the first *mail* example. In this case

```
$ mail -s "Hello" gene << END
> A letter to see if you are still with us.
> END
```

The ">" in front of each HERE document line is the Shell secondary prompt. Shell prompts are configurable (see unit 2 for example).

This will work in a Shell Script, allowing input to come from the text of the script between the "END" markers rather than from a file. The file descriptor is not usually included, but "*0*<<" would work.

Note that the final *END* marker is on a line by itself. You could use any string of characters to mark the ends, but the word *END* seems appropriate. A space must separate the chosen marker from "`<<`".

If "-" follows the "`<<`", i.e. "`<<- END`", leading tabs are ignored in the input text. A "\" will prevent substitutions from taking place — otherwise you can refer to variables and substitute command values.

Output Redirection

Redirecting standard output to a file: >

command > filename

```
$ ls /home/chris
data_file script2 script3 shell_prog table1
$ _

$ ls /home/chris > listing
$ _
```

Redirecting standard error output to a file: 2>

command 2> filename

```
$ cat /home/chris/printout
cat: 0652-050 Cannot open printout.
$ _

$ cat /home/chris/printout 2> errors
$ _
```

Figure 1-18. Output Redirection AU232.3

Notes:

In this example, the files *listing* and *errors* are created, or overwritten if the file already exists.

It is permissible to write `command 1> filename`, but the `1` is usually omitted. However, for re-directing error output, the `2` is mandatory.

To redirect other I/O descriptors, use the syntax `n>`, where `3<=n<=9`

Note that the number in the error message is unique for each type of message and product.

Output Appending

Appending standard output to a file: >>

command >> filename

```
$ wc -l /home/chris/script3
     42  /home/chris/script3
$ _

$ wc -l /home/chris/script3 >> line_count
$ _
```

Appending standard error output to a file: 2>>

command 2>> filename

```
$ wc -c /home/chris/characters
wc: 0652-755 Cannot open characters.
$ _

$ wc -w /home/chris/words/ 2>> errors
$ _
```

Figure 1-19. Output Appending AU232.3

Notes:

The *line_count* file is appended to — the original contents remain intact.

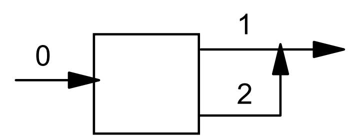
It is again permissible to write `command 1>> filename`. Again, appending to other I/O descriptors uses the `n>>` syntax.

Association

File descriptors can be joined, so that they output to the same place

command > file 2>&1

Redirects standard error to join with standard out



What do you think this command does?

```
$ cat Message_file 1>&2
```

Figure 1-20. Association AU232.3

Notes:

The order of association is significant. If we had put *command 2>&1 > file*, the error output would appear at the default destination for the standard output, while the standard output goes to the file.

Setting I/O or File Descriptors

The built-in Shell command exec allows you to

- open
- associate
- close

file descriptors

`$ exec n> of`	*Opens output file descriptor n to file "of"*
`$ exec n< if`	*Opens input file descriptor n to read file "if"*
`$ exec m>&n`	*Associates output file descriptor m with n*
`$ exec m<&n`	*Associates input file descriptor m with n*
`$ exec n>&-`	*Closes output file descriptor n*
`$ exec n<&-`	*Closes input file descriptor n*

Figure 1-21. Setting I/O or File Descriptors AU232.3

Notes:

Once executed, each of the above settings remains active for the duration of the Shell. Settings for file descriptors 0, 1 and 2 remain active in subsequent Shells. They are re-set by using *exec* to run a replacement Shell or command.

There is no way to list the current configuration of file descriptors for the Shell.

Setting I/O Descriptor Examples

To open file descriptor 3 for output to Dale's out file and 4 to Dale's err file

```
$ exec 3> /home/dale/out
$ exec 4> /home/dale/err
$ date >&3
$ ls /home/gale >&4
```

To associate output to file descriptor 3 with file descriptor 4

```
$ exec 3>&4
$ wc -l /home/gale/script3 >&3
$ wc -l /home/gale/table1 >&4
```

To close file descriptors 3 and 4

```
$ exec 3>&-
$ exec 4>&-
```

Figure 1-22. Setting I/O Descriptor Examples AU232.3

Notes:

File descriptor 3 is re-directed by the association step, so that output to file descriptor 3 is logged in Dale's *err* file — rather than the original *out* file destination. At the end of the example, Dale's *out* file contains only the date command output. Dale's *err* file contains both the listing of Gale's home directory and *wc* command outputs.

Pipes

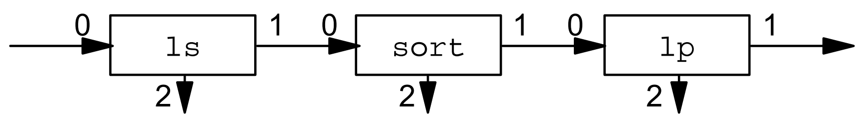
Commands can be joined, so one inputs into the next

`command1 | command2 | command3`

Gives a command *pipeline*

```
$ ls /home/robin | sort -r  |  lp
```

sorts the file list into reverse order, and prints it



Pipelines may have a branch using *the tee* command

- duplicates the standard input to the branch and to standard out

```
$ ls /home/francis | tee raw_list | sort -r | lp
```

saves the unsorted list in the file raw_list

Figure 1-23. Pipes AU232.3

Notes:

A command which takes input from its standard input and outputs to standard output after processing is called a filter. All but the last command in a pipeline is run in a sub-shell.

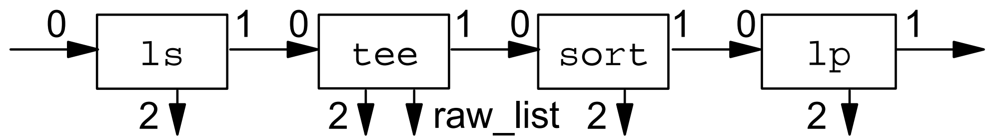
There is a 32k limit on the amount of data passing along the pipeline. If a command generates more than 32k of output it must sleep until the next command processes some of the data; then it can awaken.

Commands can be sequenced with semi-colons, but there is no interaction between them:

```
cd /home/robin ; pwd
```

Tee commands are quite useful particularly if you want to view output and keep it for later use.

The *tee* command in the above pipeline looks like this:



To append to an existing file with tee, use the *-a* option.

Command Grouping ( )

To combine the output of several commands: ( ) or { }

(command ; command ...)

- Runs commands in a Sub-Shell

For root to alter Lynn's files:

```
# ( cd /home/lynn ; chown lynn:bin d* )
```

leaves the working directory unchanged on completion

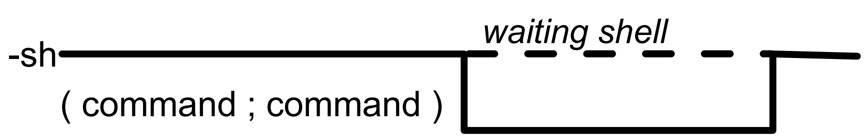


Figure 1-24. Command Grouping () AU232.3

Notes:

Even Shell builtin commands can be run in the Sub-Shell if they appear in "*( )*" parentheses. As it is a Sub-Shell, changes to the environment do not affect the main Shell.

Input and output re-direction can be applied to the grouped commands after the parentheses, e.g.:

```
( command1 ; command2 ) > /dev/null 2>&1
```

The semi-colons allow the commands to appear on the same line, you could have new lines instead:

```
( command1
command2 )
```

The *chown* command can only be run by root. With *user.group* or *user:group* specified instead of just a user name, the *chown* command also performs a *chgrp*.

Command Grouping {}

{ command ; command ... ; }

- Runs commands in the current Shell
- Directory (or environment) changes remain in effect
- Must leave spaces around the braces

Either have the braces on separate lines
or include a final "; " before the closing brace

```
# { cd /home/lynn ; chown lynn:bin s* ;}
```

{ command ; command ; }
-ksh

Figure 1-25. Command Grouping {} AU232.3

Notes:

Note that here commands in the braces do affect the "Shell". Input and output re-direction can be applied to the grouped commands after the braces, e.g.:

```
{ command
command
} > /dev/null 2>&1
```

Background Processing

Execute command in the background: &

command &

```
$ sleep 999  &
```

Waiting for the end...

```
$ date
Fri Dec 31 11:59:59 EST 1999
$ wait
```

When all background processes have finished

```
$ _
```

Figure 1-26. Background Processing AU232.3

Notes:

You can specify a process id number or Korn Shell job number to wait for — instead of waiting for all background processes. The wait command is a Shell builtin command, it completes with the same exit status as the background task. Wait can also wait for a specific job to complete and return its status. We shall learn about a command's exit status in Unit 3.

Korn Shell Job Control

Korn Shell assigns job numbers to background or suspended processes

- The **jobs** command lists your current Shell processes and their job ids
- **Ctrl-z** suspends the current foreground job
- **bg** runs a suspended job in background
- **fg** brings to foreground a suspended or background job
- Jobs can be stopped with the **kill** command

kill, fg and bg work with the following arguments:

`pid`	process id
`%job_id`	job id
`%%` - or - `%+`	current job
`%-`	previous job
`%command`	match a command name
`%?string`	match string in command line

Figure 1-27. Korn Shell Job Control AU232.3

Notes:

The *jobs* command has three options:

-l	Lists process ids along with the job ids,
-n	Lists only jobs that have stopped or exited since last notified,
-p	Lists only the process group.

Job Control Example

```
$ cc -o RUNME program_in.c
...
```

After some time running this long compilation...

```
Ctrl-z
[2] + 5692 Stopped (SIGTSTP)   cc -o RUNME program_in.c
$ jobs
+ [2] Stopped (SIGTSTP)        cc -o RUNME program_in.c
- [1] Running                  sleep 999 &
$ bg %+
[2] cc -o RUNME program_in.c
$ jobs
+ [2] Running                  cc -o RUNME program_in.c
- [1] Running                  sleep 999 &
$ kill %cc
[2] + 5692 Terminated          cc -o RUNME program_in.c
$ fg %1
sleep 999
$ _
```

Completing the sleep in the foreground...

```
$ jobs
$ _
```

Figure 1-28. Job Control Example AU232.3

Notes:

Command Line Editing and Recall

Vi option for the Korn Shell gives:

- Command line editing
- Command recall

```
$ set -o vi
```

Then simply press **ESC** to enter editing mode:

- **h** to move the cursor left
- **l** to move the cursor right
- **-** or **k** fetches commands from the history file
- **+** or **j** if you go too far back
- Plus other *vi* commands to perform line editing

Figure 1-29. Command Line Editing and Recall AU232.3

Notes:

Appendix A, at the back of your notes, contains a detailed reference for the "*vi*" command. Below are the special "*vi*" sub-commands that work only with `"set -o vi"` editing of a command line:

\	Filename completion. Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended. If the match is unique, a "/" is appended if the file is a directory and a space is appended if the file is not a directory.
*	Appends an asterisk to the current word and attempts filename generation. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.
=	Lists the file names that match the current word as if an asterisk were appended to it.

_	(Underscore) Optionally preceded by a *Count*, e.g. "*5_*". Causes the *Count*th word of the previous command line to be appended and input mode entered. The last word of the previous command line is used if *Count* is omitted.
/	Command search Searches command history for this string. Use "n" to go to the next, "N" to go to the previous.
@Letter	Searches the *alias* list for an *alias* named Letter. If an *alias* of this name is defined, its value is placed into the input queue for processing.
#	Sends the line after inserting a "#" in front of the line. Useful for causing the current line to be inserted in the history without being executed.
Ctrl-c	Terminates the `"set -o vi"` edit
Ctrl-j	(New line) Executes the current line, regardless of the mode.
Ctrl-l	Line feeds and prints the current line. Has effect only in control mode.
Ctrl-m	(Return) Executes the current line, regardless of the mode.

Checkpoint

1. What type of file is **/dev/tty3**?
2. How could we find out a file type?
3. How can we get .kshrc to run in an explicit Korn Shell?
4. How can we specify the first character in a file name to be uppercase?
5. How can we ignore error messages from a command?
6. How do you make the normal output of a command appear as error output?
7. How can we group commands, in order to re-direct the standard output from all of them?
8. What will **kill 1** do?
9. If you have submitted a job to run in foreground, how could you move it to background?
10. How would you set up a command line recall facility?

Figure 1-30. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

Summary

- AIX Shells
- Hierarchical file-system
- File names and types
- Shell Scripts
- Invoking Shells
- Shell metacharacters: expansion, Korn and quoting
- < and << input redirection
- > and >> output redirection
- 2> and 2>> error redirection
- Setting file descriptors
- Pipes and tees
- Command grouping
- Background processes
- Korn Shell job control
- Korn Shell command editing

Figure 1-31. Summary AU232.3

Notes:

Unit 2. Variables

What This Unit Is About

This unit describes how to set and reference variables. In addition, we present positional parameters and variable inheritance.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Set and reference variables
- Access positional parameters
- Analyze variable inheritance

How You Will Check Your Progress

- Checkpoint questions
- Machine exercises

Objectives

How to use Shell variables and parameters:

- Setting variables
- Referencing variables
- Using Positional Parameters
- Shifting arguments
- Setting Positional Parameters
- Using Shell parameters
- How inheritance works
- Listing Shell variables
- Listing Environment variables

Figure 2-1. Objectives AU232.3

Notes:

Setting Variables

To assign a value to a variable: **name=value**

```
$ var1=Fri
$ _
```

To protect a variable against further changes:

`readonly name=value`

- or -

`typeset -r name=value`

```
$ readonly var1=Sun
$ var1=Mon
ksh: var1: This variable is read only
$ _
```

Figure 2-2. Setting Variables AU232.3

Notes:

There are no spaces around the "=". Variable assignments remain in effect for the duration of the Shell.

It is a good idea not to use uppercase names for your variables — the Shell does, and there could be conflicts. There are no Korn Shell limitations on the length of a variable name, or the length of its contents.

A *readonly* variable cannot be assigned a new value or be *unset*. The Shell itself can change *readonly* variables — e.g. if you make any Shell-set variable *readonly*. The command is a Shell builtin. To initialize a *readonly* variable, set the value when declaring the variable. The *typeset* command is a Korn Shell builtin (not available in other Shells) — more in later Units. You cannot assign values with the *readonly* command in the Bourne Shell.

With no further arguments, both *readonly* and *typeset -r* list the variables that are readonly.

With AIX Version 4, a new option *readonly -p* gives a list of readonly variables in the format "`readonly var=val`".

Referencing Variables

To reference a variable, prefix name with a **$**

```
$ print $var1
Fri
$ _
```

To separate a variable reference from other text use: **${ }**

```
$ print The course ends on $var1day
The course ends on
$ print The course ends on ${var1}day
The course ends on Friday
$ _
```

Figure 2-3. Referencing Variables AU232.3

Notes:

The *print* command is a Korn Shell builtin command. You can get the same functionality by using either the */bin/echo* command provided by the AIX operating system, or the *echo* command builtin to the Shells.

Unset variables have no value, and so nothing is printed when you reference them in a *print* command.

Positional Parameters

Parameters can be passed to Shell Scripts as arguments on the command line

```
$ params.ksh arg1 arg2
```

- "arg1" is Positional Parameter number 1
- "arg2" is Positional Parameter number 2
- Others are unset

They are referenced in the script by:

- $1 to $9 for the first nine
- ${10} to ${n} for the remainder (Korn Shell only!)

Figure 2-4. Positional Parameters AU232.3

Notes:

In the Bourne Shell you cannot reference more than nine arguments at once.

If you want to pass arguments that begin with a "-" or "+", you can use the convention that "--" marks the end of options for a command or script. You will see how to use this in unit 5 with the option processing command *getopts*. For example:

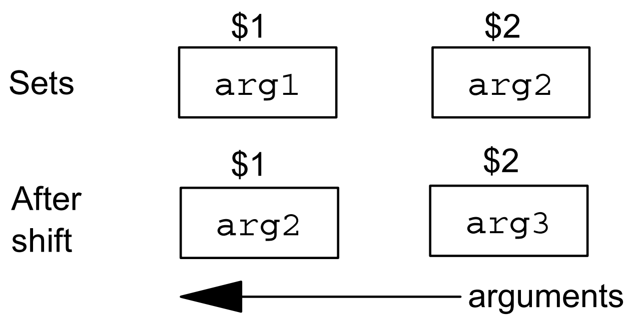
```
params.ksh -- -arg1 +arg2 arg3
```

This will prevent "*-arg1*" being treated as an option rather than an argument.

Shifting Arguments

In a Shell Script the **shift** command moves arguments "to the left":

```
$ params.ksh arg1 arg2 arg3
```



- Discarding the first or "leftmost" argument
- Decrementing the number of Positional Parameters
- Allowing Bourne Shell to reference more than 9 arguments

Figure 2-5. Shifting Arguments AU232.3

Notes:

You can specify a number of parameters for *shift*, e.g.

```
shift 3
```

moves three parameters to the left, discarding the leftmost three. The Shell provides *shift* as a builtin command.

Setting Positional Parameters

In a Shell Script the **set** command can:

- Change the values of Positional Parameters
- Unset Positional Parameters previously set

```
 $ cat first.ksh
print $1 $2 $3
set value1 value2
print $1 $2 $3

$ first.ksh a b c
a b c
value1 value2

$ _
```

Figure 2-6. Setting Positional Parameters AU232.3

Notes:

Set is a Shell builtin command. Here parameter 3 was cleared (or unset) by the use of the *set* command.

The Shell command *unset* can be used to clear a variable from memory and so remove it:

```
unset var1
```

or

```
unset -v var1
```

AIX Version 4 introduced the "-*v*" option for *unset*. This option corresponds to the POSIX standard recommendation.

Variable Parameters

Shell Scripts set a number of other Shell Parameters:

$#	The number of Positional Parameters set
$@	Positional Parameters in a space separated list
$*	Positional Parameters in a list separated by the first Field Separator (the default is a space)

In double quotes, $@ and $* behave differently:

```
"$@"  =   "$1" "$2" "$3" . . .

"$*"  =   "$1 $2 $3 . . . "
```

Figure 2-7. Variable Parameters AU232.3

Notes:

The *IFS* (Internal Field Separator) variable contains the Field Separator characters. In most Shells these characters default to Space, Tab and Newline.

We shall see more of *IFS* later...

Some Shell Parameters

Shell Parameters that remain fixed for the duration of the Script:

$0 The (path)name used to invoke the Shell Script

$$ The Process Id (PID) of current process (shell)

$- Shell Options used to invoke the Shell, e.g. -r

Parameters set as the Script executes commands:

$! The PID of the last background process

$? The return code from the last command executed

Figure 2-8. Some Shell Parameters AU232.3

Notes:

As *$0* remains fixed for the duration of a Shell Script, it is not affected by the *shift* command seen earlier. It is the pathname used to invoke the script.

If you see "*ism*" in the *Shell Options* these are the usual default options for a command login Shell. The option letters mean the Shell is in interactive mode, it uses STDIN for commands, and it has job control (m=monitor) enabled respectively. We shall see all of the options in Unit 5.

We shall see more of *$?* in the next Unit. You should note that PID is a very common abbreviation used in documentation and commands.

Parameter Code Example

So let's put all of it into action in a Shell Script...

```
$ cat second.ksh
print $$
print $0
print "$# PPs as entered"
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
shift
print $0
print "$# PPs after a shift"
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
set "$@"
print 'Set "$@" - parameters in double quotes'
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"
set "$*"
print 'Set "$*" - parameters space separated'
print "PP1=$1 PP2=$2 PP3=$3 PP4=$4"

$ _
```

Figure 2-9. Parameter Code Example AU232.3

Notes:

On the next page we shall see what this does...

Parameter Output Example

Here's what it does...

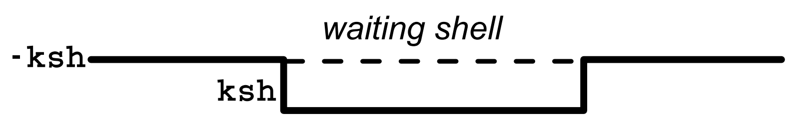
```
$ second.ksh arg1 arg2 "arg3 and text"
4687
second.ksh
3 PPs as entered
PP1=arg1 PP2=arg2 PP3=arg3 and text PP4=
second.ksh
2 PPs after a shift
PP1=arg2 PP2=arg3 and text PP3= PP4=
Set "$@" - parameters in double quotes
PP1=arg2 PP2=arg3 and text PP3= PP4=
Set $* - parameters in double quotes
PP1=arg2 arg3 and text PP2= PP3= PP4=
$ _
```

Figure 2-10. Parameter Output Example AU232.3

Notes:

This Shell and the Next

What happens to variables when you spawn a Sub-Shell?



Unless you export variables, they will not be passed on.

`$ set`	*to list all variables and values*
`$ export var` - or - `$ typeset -x var`	*export variable var so that it will be inherited by Sub-Shells, or use typeset in the Korn Shell*
`$ export` - or - `$ typeset -x`	*to list variables that are exported, other variables will be unset in a Sub-Shell*

Figure 2-11. This Shell and the Next AU232.3

Notes:

Attributes of variables are also inherited — like a *readonly* attribute for example.

In the Korn Shell you can use the *export* command to set variable values and export them in one step: e.g.

```
$ export var=value
```

or

```
$ typeset -x var=value
```

With AIX Version 4 "`export -p`" gives a list of exported variables in the format "`export var=val`".

The *set* command also reports variable settings in single quotes.

The *env* command performs a similar function to the "*export*" builtin command above, but it is an external operating system command.

You will see more about *typeset* in later units.

Inheritance Example

Let's see inheritance in action...

`$ x=324`	*We can set a variable x*
`$ print "$$: X=$x"`	*in our current shell*
`4589: X=324`	
`$ ksh`	*In a Sub-Shell, x is unset*
`$ print "$$: X=$x"`	*- there is no value to print*
`4590: X=`	
`$ _`	
`Ctrl-d`	*Returning to the main Shell...*
`$ print "$$: X=$x"`	
`4589: X=324`	*x will have its value restored*
`$ export x`	*If we export x, a Sub-Shell*
`$ ksh`	*can inherit the value of x*
`$ print "$$: X=$x"`	
`4591: X=324`	
`$ x=3`	*If we change x from the*
`$ _`	*Sub-Shell, the change does*
`Ctrl-d`	*not affect the main Shell*
`$ print "$$: X=$x"`	
`4589: X=324`	

Figure 2-12. Inheritance Example AU232.3

Notes:

Important points to note here are:

- To use a value in a script or sub-shell it **MUST** be *export*ed.
- You can **never** pass a value back (or up) from a sub-shell to a calling shell with an exported variable.
- Unset or unexported variables have a NULL (string) value.

Korn Shell Variables

Korn Shell sets certain variables each time they are referenced:

`SECONDS`	seconds since Shell invocation
`RANDOM`	random number in the range 0 to 32767
`LINENO`	current line number within a Shell Script or function
`ERRNO`	system error number of the last failed system call – a system-dependent value!

Figure 2-13. Korn Shell Variables AU232.3

Notes:

Every variable above holds integer values. None of the above are exported by default.

Notice that each variable name is in upper case. Shell variable names are generally upper case. To avoid conflicts, you should avoid using upper case variable names.

You can set *SECONDS* to an initial value, so that subsequent references yield that value plus the number of seconds since Shell invocation,
e.g. `$ SECONDS=35`

You can initialize the *RANDOM* number sequence by assigning a value to the variable,
e.g. `$ RANDOM=$$`

You can clear the ERRNO variable by assigning the value zero to it,
i.e. `$ ERRNO=0`

Other Shell variables (which we shall see next) also lose their special meanings if they are *unset*.

Environment Variables

Several variables define the environment of a Shell:

`CDPATH`	a search path for the cd command
`HOME`	your home directory
`IFS`	input field separators (defaults to: space, tab, newline)
`MAIL`	the name of your mail file
`MAILCHECK`	mail check frequency (default 600 seconds)
`MAILMSG`	the "you have new mail" message
`PATH`	the system command search path
`PS1`	the primary Shell command prompt
`PS2`	a secondary prompt for multi-line entry
`SHELL`	the pathname of the Shell
`TERM`	the terminal type (selects terminfo file)

Figure 2-14. Environment Variables AU232.3

Notes:

MAILCHECK holds an integer value, *unset* removes the special meaning.

The Shell sets default values for *IFS* and *MAILCHECK*. The *login* program sets up the *HOME* variable. The Shell normally does not set a value for *MAIL*.

The Shell sets default values for *PATH*, *PS1* and *PS2*. The Shell normally does not set a value for *SHELL*. The AIX login process sets the value for *TERM* — this is taken from the Object Data Manager (ODM).

You can customize the Shell prompts

- In PS1 "!" is replaced by the command number
- Use single quotes to include Shell set variables

  $ PS1=!' $SECONDS : '

Shell defaults for PS1 and PS2 are:

```
PS1='$ '
PS2='> '
```

Korn Environment Variables

Korn Shell specific features require environment variables:

`COLUMNS`	screen width
`EDITOR`	the editor for command line editing
`ENV`	program/script to be sourced for each new Shell
`FCEDIT`	an editor for the `fc` command
`FPATH`	a search path for function definition files
`HISTFILE`	your history file
`HISTSIZE`	limit of history commands accessible
`LC_COLLATE`	sorting sequence for pattern ranges
`LINES`	screen length
`OLDPWD`	previous working directory for `cd -`

Figure 2-15. Korn Environment Variables AU232.3

Notes:

None of the above are exported by default.

`COLUMNS` defaults to 80, `LINES` to 24. Both of these variables control window editing and, as we shall see in Unit 4, the *select* command.

By default, `ENV` is not set.

`HISTFILE` implicitly defaults to `$HOME/.sh_history`, while `$HISTSIZE` has the value 128.

`LC_COLLATE` is normally set to "*En_GB*" or "*en_GB*" in the UK, and "*En_US*" or *"C(POSIX)"* in America.

Unit 5 describes the *fc* command, and Unit 7 the function of `FPATH`.

Korn Environment Variables (Cont.)

`OPTARG`	required value for an option – `getopts`
`OPTIND`	index of the next argument for `getopts` to process
`PPID`	the parent process id
`PS3`	prompt for the `select` command
`PS4`	debug prompt for ksh with the -x option
`PWD`	the current working directory
`REPLY`	set by `select` command and the `read` command if no argument is given
`TMOUT`	seconds to Shell timeout
`VISUAL`	a visual editor – overrides `EDITOR`

Figure 2-16. Korn Environment Variables (Cont) AU232.3

Notes:

`OPTIND` is set to 1 for each Shell script or function that executes. `OPTARG`, `OPTIND` and `TMOUT` lose their special meaning if they are *unset*. `PWD` is exported by default in the Korn Shell.

`TMOUT` holds an integer value. The Shell default value of zero means no timeout. The Korn Shell waits one minute before dying after issuing a warning message and a beep.

We shall see more of *PS3* and *REPLY* in Unit 4; *OPTARG* and *OPTIND* in Unit 5.

The Bourne Shell provides further environment variables:

`NLFILE`	file with extended character set details
`NCLTAB`	sort collating sequence
`SHACCT`	command history for use by system accounting
`TIMEOUT`	minutes to Bourne Shell timeout — which is without warning!

Checkpoint

1. How could we use positional parameter 3 in a shell script?
2. Which variable contains the number of positional parameters?
3. How can we change the value of a variable set in a different process?
4. What is the variable ***IFS***?
5. How can we reset ***PS1*** to show the current directory?
6. By setting a variable, how can we have a command recall facility?

Figure 2-17. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.
2.
3.
4.
5.
6.

Summary

- Setting variables
- Referencing variables
- Positional Parameters
- Shifting arguments
- Setting Positional Parameters
- Shell parameters
- Inheritance
- Shell variables
- Environment variables

Figure 2-18. Summary AU232.3

Notes:

Unit 3. Return Codes and Traps

What This Unit Is About

This unit provides the student with the opportunity to review basic testing concepts and explore shell scripting using return codes, signals, and traps.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Identify conditional execution statements
- Analyze return codes and signals
- Test variables or files for specified conditions
- Handle signals in a script with traps

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

In this unit we will learn about:

- Return values
- Exit Codes
- Conditional execution
- The test command
- Compound expressions
- File test operators
- Numerical expressions
- String expressions
- Korn Shell test operators
- Korn Shell [[ ]] expressions
- Signals
- Sending signals
- Catching signals

Figure 3-1. Objectives AU232.3

Notes:

Return Values

Each command, pipeline or group of commands returns a value to its parent process

$? contains the value of the return code

- **zero** means success
- **non-zero** means an error occurred

The single value returned by a pipeline is the return code of the last command in the pipeline

For grouped commands – that is, ( ) or { } – the return code is that of the last command executed in the group

Figure 3-2. Return Values AU232.3

Notes:

Exit Status

A Shell script provides a return code using the *exit* command

`$ print $$`	*check the Shell process id*
`879`	
`$ ksh`	*start a new Sub-Shell*
`$ print $$`	*and check its process id*
`880`	
`$ exit`	*quit the Sub-Shell*
`$ print $?`	*and print the return code*
`0`	
`$ print $$`	
`879`	
`$ ksh`	*begin another Sub-Shell*
`$ print $$`	
`890`	
`$ exit 101`	*exit with a value to set*
`$ print $?`	*the return code*
`101`	
`$ print $$`	
`879`	
`$ _`	

Figure 3-3. Exit Status AU232.3

Notes:

The *exit* command is a Shell builtin command.

Conditional Execution

A return code (or exit status) can be used to determine whether or not to execute the next command

- if command1 is successful execute command2

```
command1 && command2

$ rm -f file1 && print file1 removed
```

- if command1 is not successful execute command2

```
command1 || command2

$ who|grep marty || print Marty logged off
```

Figure 3-4. Conditional Execution AU232.3

Notes:

The *-f* option to the *rm* command prevents interactive questions being displayed when file permissions do not allow read or write for the named file. The command returns status "0" only if the named file is deleted.

The operating system command *who* lists the users logged on to the system. The *grep* operating system command searches standard input for the pattern specified — only if a match is found will it return an exit status "0" (the return code).

The test Command

The test command is used for expression evaluation

```
test expression
```

- or -

```
[ expression ]
```

- returns zero if the expression is true
- returns non-zero if the expression is false

The Korn Shell provides an improved version

```
[[ expression ]]
```

- easier syntax
- includes same functionality as *test*
- additional operators
- Shell expansions prevented

Figure 3-5. The test Command AU232.3

Notes:

Test operators form expressions that we shall see later.

The keywords *true* and *false* have their obvious meanings.

If you use metacharacters with *test* or `[ ]` they will be expanded: with `[[ ]]` they are only expanded if they appear as a pattern in a string expression — refer to "Korn Shell `[[ ]]` Expressions" later in this unit.

The Korn Shell provides additional operators for use with the *test* command compared to the Bourne Shell, as well as further operators for use with the `[[ ]]` syntax.

Compound Expressions

For the [ ] or test command

`exp1 -a exp2`	binary and operation
`exp1 -o exp2`	binary or operation
`! exp`	logical negation
`\(   \)`	to group expressions

For the [[   ]] syntax

`exp1 && exp2`	true if both expressions are true - the second is only evaluated if the first is true
`exp1 || exp2`	true if either expression is true - the second is only evaluated if the first is false
`! exp`	logical negation
`(   )`	to group expressions

Figure 3-6. Compound Expressions AU232.3

Notes:

Notice that with *test* or `[ ]` you need to escape Shell metacharacters (like parentheses). Compound expressions are valuable with multiple test operators and tests.

File Test Operators

File status can be examined using several operators

Operator:	*True if ....:*
`-s file`	file has a size greater than zero
`-r file`	file exists and is readable
`-w file`	file exists and is writable
`-x file`	file exists and is executable
`-u file`	file exists and has the SUID bit set
`-g file`	file exists and has the SGID bit set
`-k file`	file exists and has the SVTX sticky bit set
`-e file`	file exists
`-f file`	file exists and is an ordinary file
`-d file`	file exists and is a directory
`-c file`	file exists as a character special file
`-b file`	file exists as a block special file
`-p file`	file exists and is a named pipe file
`-L file`	file exists and is a symbolic link

Figure 3-7. File test Operators AU232.3

Notes:

Note a file will appear to be writable even though it is within a read-only file system. Only the file access control list is examined, not the file system status.

An executable directory file is a directory that can be searched — you may *cd* to the directory.

The operator "`-e`" was added with AIX Version 4.

Numeric Expressions

For arithmetic expressions and integer values use

Expression:	*True if ....:*
`exp1 -eq exp2`	`exp1 is equal to exp2`
`exp1 -ne exp2`	`exp1 is not equal to exp2`
`exp1 -lt exp2`	`exp1 is less than exp2`
`exp1 -le exp2`	`exp1 is less than or equal to exp2`
`exp1 -gt exp2`	`exp1 is greater than exp2`
`exp1 -ge exp2`	`exp1 is greater than or equal to exp2`

Figure 3-8. Numeric Expressions AU232.3

Notes:

Numerical values are compared using the above operators. If variable *x* has been assigned a numerical value, you test *x* as follows:

```
$ x=2                -or-      $ [ $x -eq 2 ]
$ test $x -eq 1
$ [[ $x -eq 3 ]]
```

String Expressions

To examine strings use one of the following

Expression:	*True if ....:*
`-n str`	`str is non-zero in length`
`-z str`	`str is zero in length`
`str1 = str2`	`str1 is the same as str2`
`str1 != str2`	`str1 is not the same as str2`

Figure 3-9. String Expressions AU232.3

Notes:

Character strings are compared using the above operators. If variable *h* has been assigned a character string, you test *h* as follows:

```
$ h=Hello              -or-       $ [ $h = Welcome ]
$ test $h = Hello
$ [[ $h = Hi ]]
```

To avoid syntax errors from *test* or the Shell, you usually surround the $variable with double quotes — as in "$h". This avoids problems testing with NULL strings in particular (why?).

Korn Shell Test Operators

The Korn Shell provides a number of additional `test` operators

Expression:	*True if ...:*
`file1 -ef file2`	file1 is another name for file2
`file1 -nt file2`	file1 is newer than file2
`file1 -ot file2`	file1 is older than file2
`-O file`	file exists and its owner is the effective user id
`-G file`	file exists and its group is the effective group id
`-S file`	file exists as a socket special file
`-t des`	file descriptor *des* is open and associated with a terminal device

Figure 3-10. Korn Shell test Operators AU232.3

Notes:

You can use metacharacters in filenames.

Korn Shell [[ ]] Expressions

When using the Korn Shell [[ ]] syntax there are a few extra expressions...

Expression:	*True if ...:*
`str = pattern`	str matches pattern
`str != pattern`	str does not match pattern
`str1 < str2`	str1 is before str2 in the ASCII collation sequence
`str1 > str2`	str1 is after str2 in ASCII collation
`-o opt`	option *opt* is on for this shell

You may use Shell metacharacters in the patterns

Figure 3-11. Korn Shell [[ ]] Expressions AU232.3

Notes:

Examples:		
	`abc = *c`	true
	`abc != ?c?`	true
	`abc < def`	true

Remember that Shell metacharacters may be used in patterns.

Also, due to locale settings, some string comparisons may not give the answers you expect. This is particularly true if `LANG` is not set to `en_US`.

Practice Test

```
$ [[ -s /etc/passwd || -r /etc/group ]]
$ print $?                  True or False?

$ test -f /etc/motd -a ! -d /home
$ print $?                  True or False?

$ x="005"
$ y=" 10"
$ test "$y" -eq 10
$ print $?                  True or False?

$ [ "$x" = 5  ]
$ print $?                  True or False?

$ [[  -n "$x"  ]]
$ print $?                  True or False?


$ test  -S  /dev/tty0
$ print $?                  True or False?

$ [[  1234 = +([0-9])  ]]
$ print $?                  True or False?
```

Figure 3-12. Practice Test AU232.3

Notes:

Signals

The kernel sends *signals* to processes during their execution

- certain system events issue signals when they
  - run out of paging space
  - receive special key sequences like <Ctrl-c>
- The **kill** command sends a specific signal to a process

Figure 3-13. Signals AU232.3

Notes:

To terminate a foreground process you can press the Interrupt key sequence (normally <Ctrl-c>). Your input causes the relevant *signal* to be sent to your foreground process by the system.

The *kill* command is the only way to terminate a background process.

What You Can Do with Signals

Signals sent to processes may be

- Caught — the process deals with it
- Ignored — nothing happens
- Defaulted — use default *handlers*

Figure 3-14. What You Can Do with Signals AU232.3

Notes:

Signals are a form of simple interprocess communication. If a process takes default action on a signal, this normally means terminate (die!). If you do not want the default you can either ignore or trap the signal.

The Kill Command

- To send a signal to a process:

```
kill -sig pid    -or-    kill -s sig pid
```

- To signal the current process group:

```
kill -sig 0      -or-    kill -s sig 0
```

- To send a signal to all of your processes, except those with PPID 1 (do not use if you are root):
-

```
kill -sig -1     -or-    kill -s sig -1
```

- To list all defined signals

```
kill -l
```

- To list the signal that caused an exit error

```
kill -l $?
```

Figure 3-15. The kill Command AU232.3

Notes:

The current process group means all processes started from, and including, the current login Shell.

The "*-s sig*" and "*-l $?*" options were introduced with AIX Version 4.1.

The full signal list is held in /usr/include/sys/signal.h.

We know in many cases the default action is for the process to die upon receipt of the signal. However, some signals are ignored. A list of useful signals follows on the next pages.

Signal List

Here is a list of some useful signals

Signal:		Event:
0	EXIT	issued when a process or function completes (Shell specific)
1	HUP	you logged out while the process was still running – sent to Sub-Shells too
2	INT	interupt pressed (Ctrl-c)
3	QUIT	quit key sequence pressed (Ctrl-\)
15	TERM	default kill command signal
18	TSTP	process suspend (Ctrl-z)

Figure 3-16. Signal List AU232.3

Notes:

The Bourne Shell issues the *EXIT* (0) signal only upon completion of a Shell process.

The INT (2) signal key sequence may vary with terminal type. For AIX Version 3 and IBM-3151 ASCII terminals it is <Ctrl-c>; other common sequences are <Ctrl-Backspace> and <Delete>.

The default key configurations for a terminal can be changed through *smit* — terminal attributes — or by using the *stty* command for the session. To change the *QUIT* sequence to <Ctrl-t>:

```
$ stty quit ^t
```

Signal names include a "SIG" prefix to the signal codes listed above, i.e. *SIGDANGER*. By default background processes stop if they attempt to read from a terminal. To set this behavior for background processes that attempt to write to a terminal, use:

```
$ stty tostop
```

You should avoid the `KILL` signal except as a last resort. If you send a `KILL` to a process it can never be caught so it is impossible to perform cleanup actions (like removing lock files etc.).

Signals KILL (9), SEGV (11), STOP (17) and SAK(63) may not be trapped under AIX V3 or V4.

Signal List (Cont.)

Signal:		Event:
19	CONT	continue if stopped – issued by `kill` to a suspended process before TERM or HUP
29	PWR	power failure imminent – save data now!
33	DANGER	paging space low
63	SAK	you pressed <Ctrl-x> and <Ctrl-r> the SAK sequence

Figure 3-17. Signal List (Cont.) AU232.3

Notes:

A reserved key sequence, called the secure attention key (SAK), allows a user to request a trusted communication path which is part of TCB (Trusted Computing Base).

Catching Signals with Traps

The `trap` command specifies any special processing you want to do when the process receives a signal:

To process signals

```
$ trap 'rm /tmp/$$; print signal!; exit 2' 2 3
```

To ignore signals

```
$ trap '' INT QUIT
```

To reset signal processing

```
$ trap - INT QUIT
```

- or -

```
trap 2 3
```

To list traps set

```
$ trap
```

Figure 3-18. Catching Signals with Traps AU232.3

Notes:

The Shell *trap* command allows your script to catch specific signal.

You should use single quotes to enclose the action — to protect it from Shell expansions, although double quotes may also work. In the Korn Shell signal names or numbers may be used, but names are more portable. For the Bourne Shell only numbers are allowed.

The signals trapped can be system or user initiated. Once a signal is set to be ignored, Sub-Shells also ignore that signal, and cannot then trap the signals themselves.

Notice that you need to explicitly use *exit* if you want to terminate the script from within a *trap*.

Trap Example

```
#!/usr/bin/ksh
# ps_monitor
# monitor processes using ps -elf at intervals
# of 30 seconds for 2 minutes.  If interrupted,
# a summary report is produced by executing
# psummary.
#
trap 'print $0: interrupt received ;
        ./psummary ;
        exit' 2 3 15
ps -elf > /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
sleep 30
ps -elf >> /tmp/pdata
trap - 2 3 15
```

Figure 3-19. Trap Example AU232.3

Notes:

Which directory does the `trap` command use for the `./psummary` command/script?

Checkpoint

1. How can you tell whether a command you have just entered was successful?
2. How can you test if file *datafile* is non-empty?
3. How can you check if you have been logged on for more than 20 minutes, and if so, print out a suitable message?
4. How could you log off, using the kill command?
5. If you are a DBA is this a desirable command to terminate the <oracle_server>? **kill -KILL <oracle_server>**
6. What does this command do? **trap echo you did <Ctrl-c> 2**
7. How could you get <Ctrl-c> to log you off?

Figure 3-20. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

7.

Summary

- Return values
- Exit status
- Conditional execution
- The *test* command
- Compound expressions
- File *test* operators
- Numerical expressions
- String expressions
- Korn Shell *test* operators
- Korn Shell *[[ ]]* expressions
- Signals
- Sending signals – *kill* command
- Catching signals – *trap* command

Figure 3-21. Summary AU232.3

Notes:

Unit 4. Flow Control

What This Unit Is About

This unit presents flow control using conditional loops and decision making.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Generate if-then-else statements
- Generate while/until loops
- Understand and use for loops
- Create case and select constructs
- Leave loops prematurely

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Machine exercises

Objectives

For practical Shell Scripts we need program logic:

- The *if - then - else* construct
- Conditional loops with *until* and *while*
- Specific value iteration with *for*
- Multiple choice pattern matching with *case*
- The *select* command for menus
- Breaking and continuing loops
- Doing nothing – the null command

Figure 4-1. Objectives AU232.3

Notes:

The *if - then - else* Construct

```
if expression1
then
      commands to be executed if
      expression1 is true
elif expression2
then
      commands to be executed if
      expression1 is false, and
      expression2 is true
elif expression3
then
      commands to be executed if
      expression1 and expression2 are
      false, but expression3 is true
else
      commands to be executed if all
      expressions are false
fi
```

Figure 4-2. The if - then - else Construct AU232.3

Notes:

The italicized text marks optional parts of the syntax — you do not always need an *else* part, but there can be only one! Any number of *elif ... then* segments may be included.

As soon as a true expression is found, the corresponding block of commands is executed. Then the flow of the program will continue after the closing *fi* statement. The return value of the construct is that of the last command block executed, or true if none was executed.

if Example

Here is a simple if construct:

```
#!/usr/bin/ksh
# Usage: goodbye username
#
if [[ $# -ne 1 ]]
then
        print  "Usage is:  goodbye username"
        print  "Please try again."
        exit 1
fi
rmuser $1
print "O.K., $1 is removed."
```

When we run "goodbye", this is what we get ...

```
$  goodbye
Usage is:  goodbye username
Please try again.
$  goodbye pete
O.K., pete is removed.
$ _
```

Figure 4-3. if Example AU232.3

Notes:

We have used the Korn Shell `[[ ]]` syntax for the expressions above, but it could just as easily have been the older `[ ]` or *test* command. In fact any command, or even group of commands, could be used as an expression. Metacharacters are expanded and variable references are allowed. It is the return value of the expression that is used to decide true or false: zero = true.

Conditional Loop Syntax

until *expression* **do**   commands executed   when *expression* is false **done**   # *optional* < *file*
while *expression* **do**   commands executed   when *expression* is true **done**   # *optional* < *file*

Figure 4-4. Conditional Loop Syntax AU232.3

Notes:

The *while* loop will be executed only if the expression evaluates true. An *until* loop follows the reverse logic — executing only if the expression is false.

If the expression is a command or a group of commands, rather than a logical *test* or `[[ ]]` expression, you can redirect input from a file into the command expression.

Both *until* and *while* return the value of the last loop command executed, or true if no loops were executed. The program continues after the *done* statement.

until Loop Example

The C compiler returns a non-zero exit code **until** its compilation is successful:

```
$ until  cc prog.c
> do
>   vi prog.c
> done
$ _
```

Figure 4-5. until Loop Example AU232.3

Notes:

The example was so simple, that it was entered at the command line — not as a Shell Script. For this reason we get secondary prompts for the second and subsequent lines of the command entry.

Any of the program logic control commands can be entered in this way — but you might find it confusing with long or complicated constructs.

while true Example

The Script "forever" is a tough cookie!

```
#!/usr/bin/ksh
# An endless loop with a trap for INT QUIT TSTP
trap 'print "hasta la vista - baby!"' 2 3 18
while true
do
        print  "I'll be back."
        sleep 10
done
```

`$  forever`	
`I'll be back.`	*every ten seconds*
`I'll be back.`	*the script speaks!*
`I'll be back.`	
`Ctrl-c`	*an attempt to stop it...*
`hasta la vista - baby!`	*invokes the trap, and*
`I'll be back.`	*it carries on.*
`I'll be back.`	

Figure 4-6. while true Example AU232.3

Notes:

The *true* and *false* are shell builtins that are available for use as expressions.

This script traps normal keyboard kill sequences, so that you must *kill* it from another terminal.

for Loop Syntax

```
for  identifier  in  word1  word2  ...
do
     commands  using  $identifier
     more  commands
done
```

```
for  identifier
# equivalent to: for identifier in "$@"
do
     commands  using  $identifier which takes
     values from the positional parameters
done
```

Figure 4-7. for Loop Syntax AU232.3

Notes:

Perhaps a better description of the *for* loop is a specific value iteration command — it iterates over a parameter list (the set of values).

The *for* command sets the *identifier* variable to each of the values from the *word* or positional parameter list in turn, and executes the command block. Execution ends when the *word* or positional parameter list is exhausted. The return value is that of the last block command executed, or true if none were.

The word list in the first form of the *for* command can contain metacharacters for file name expansion.

for - in Loop Example

Here we have a quick tidy-up to delete files:

```
$  for  file  in  *.tmp
>  do
>   rm -f $file
>  done
$  _
```

Why use the option -f ?

Figure 4-8. for - in Loop Example AU232.3

Notes:

The word list in the *for* command has been formed by metacharacter expansion into the file names from the current directory that end in "*.tmp*".

for Loop Example

The sample Script "getprice.ksh" will look up the price list:

```
#!/usr/bin/ksh
# getprice.ksh - select price from "pricelist" file
# for each item entered on the command line
# Usage: getprice item1 item2 ...
#
for item
do
        grep -i "$item" pricelist
done


$ getprice.ksh  "Shock Absorbers"  "Air Filter"
Front Shock Absorbers      49.99
Rear Shock Absorbers       59.99
Air Filter                 10.99
$ _
```

Figure 4-9. for Loop Example AU232.3

Notes:

By omitting the *in word1 word2 ...* part of the *for* command syntax, the command takes its list from the positional parameters — as if you had specified `in "$@"`.

The case Statement

case word **in**
(**pattern1** | *pattern2* | ...)
action ;;
(*) *default* ;;
esac

case $identifier **in**
(**pattern1**) command1
more_commands ;;
(pattern2 | pattern3) *commands* ;;
(*) *commands* ;;
esac

Figure 4-10. The case Statement AU232.3

Notes:

The *case* statement compares the *word* with each *pattern* in turn. If a match is found, the corresponding action is performed. The double semi-colon syntax marks the end of an action. Null actions are allowed. Multiple patterns can be associated with an action — each separated by a pipe character. Patterns can contain metacharacters. Spaces around a pattern are ignored.

There must be at least one pattern block and it is a good idea to include a final "catch-all" pattern the metacharacter "*". Once a match is found, or after all patterns have been checked, the program continues after the *esac* statement.

The Korn Shell allows an optional open bracket "(" at the start of each pattern group, so that you can use the command grouping ( ) syntax around a *case* construct. The Bourne Shell does not allow this.

case Code Example

A guessing game of sorts:

```
#!/usr/bin/ksh
# Usage:  match string
# To see how lucky you are feeling today

case "$1" in
     Ace  )        print "You are really close."  ;;
     King  )       print "Missed it by that much."  ;;
     Queen  )      print "Finally!" ;;
     Jack  )       print "I hope you'll get it next time." ;;
     *  )          print "Guess again." ;;

esac
```

Figure 4-11. case Code Example AU232.3

Notes:

You can use combinations of variable references and fixed text to form a *word* to be matched if you like.

Note where you specify the catch-all pattern.

Case Code Output

A casino dealer in the making?

```
$ match Three
Guess again.

$ match Jack
I hope you'll get it next time.

$ match Ace
You are really close.

$ match King
Missed it by that much.

$ match Queen
Finally!
```

Figure 4-12. case Code Output AU232.3

Notes:

Mini Quiz

1. There can be any number of *elif* statements in an *if – then – else* construct.
2. *while - true* and *until - false* — are they equals or opposites?
3. The statement: "*for identifier* " takes its input from positional parameters.

Figure 4-13. Mini Quiz AU232.3

Notes:

The Korn Shell select Syntax

```
select identifier in word1 word2 ...
do
     commands using $identifier usually
     containing a case statement
done
```

```
select identifier
# equivalent to: select identifier in "$@"
do
     commands using $identifier from positional
     parameters usually containing a case
     statement
done
```

Figure 4-14. The Korn Shell select Syntax AU232.3

Notes:

The Korn Shell *select* command displays the *word* or *positional parameter* list as items in a numbered menu, output is to standard error. The environment variables *LINES* and *COLUMNS* control output size.

The *PS3* prompt is displayed as a prompt for you to enter the number of your choice. The variable *REPLY* is set to the character string that you enter. The variable *identifier* is set to the *word* or *positional parameter* value corresponding to your selection. If you choose an unlisted item, or enter any other unidentified text, *identifier* is set to null.

The command block is executed for each selection. A null selection re-displays the menu and *PS3* prompt without executing the command block.

The select command only terminates if it encounters an end-of-file (<Ctrl-d>) input, *exit*, *break* or *return*. (We shall learn about the *break* command next, and the *return* command in unit 7.) The program continues after the *done* statement. The return value is that of the last block command, or true if no commands were executed.

select Code Example

To help identify animals we have a "barn.ksh" Shell Script:

```
#!/usr/bin/ksh
# usage: barn.ksh
PS3="Pick an animal: "
select  animal  in  cow  pig  dog  quit
do
        case $animal in
        (cow)     print "Moo"
                ;;
        (pig)     print "Oink"
                ;;
        (dog)      print "Woof"
                ;;
        (quit)      exit
                ;;
        ('')    print "Not in the barn"
                ;;
        esac
done
```

Figure 4-15. select Code Example AU232.3

Notes:

The environment variable *LINES* defaults to 24, while *COLUMNS* is 80 by default. This is fine for the screen we are using, so they were left at their default values. The *PS3* prompt default is "#? ".

The *case* "catch-all" is executed when the *select* command doesn't recognize your selection, and the animal variable is set to null.

select Output Example

Running "barn.ksh" we can choose an animal to examine ...

```
$ barn.ksh

1)  cow
2)  pig
3)  dog
4)  quit

Pick an animal: 1
Moo
Pick an animal: 2
Oink
Pick an animal: 3
Woof
Pick an animal: 8
Not in the barn
Pick an animal: 4
$
```

Figure 4-16. select Output Example AU232.3

Notes:

The menu would be re-displayed if we just press return without making a selection. As we make more and more selections, the menu is of course disappearing as the screen scrolls upward.

exit The Loop

In the Korn Shell script /usr/sbin/snap

```
...
if [ "$badargs" = n ]
then
  for choice in $cmplist
  do
  if [ "$component" = "$choice" ]
  then found=y ; break ;
  fi
  done
  if [ "$found" = y ]
  then
    if [ -r "$destdir/$component/$component.snap" ]
    then
    more $destdir/$component/$component.snap
    else
    echo "^Gsnap:  $destdir/$component/$component.snap not found"
    exit 25
    fi
  fi
else
    usage
    exit 26
fi
...
```

Figure 4-17. exit The Loop AU232.3

Notes:

The ***exit*** causes the script to end. A status number can be attached to the *exit* to inform a calling script of its success, failure, or otherwise.

break The Loop

The break command jumps out of **do . . . done** loops:

- exits from the smallest enclosing loop
- jumps out a specified *number* of layers/loops

break *number*

```
select  choice  in  Backup  Restore  Quit
do
  case $choice in
  (Backup)  find . -print|backup -iqf /dev/rfd0
  ;;
  (Restore) restore -xqf /dev/rfd0
  ;;
  (Quit)    break
  ;;
  ('')      print "What ?" 1>&2
  ;;
  esac
done
```

Figure 4-18. break The Loop AU232.3

Notes:

Following a *break* the program continues after the *done* statement — just as if the command was complete.

This is applicable to *until*, *while*, *for*, and *select* constructs.

continue The Loop

The **continue** command begins the next iteration of a **do . . . done** loop:

- starts at the top of the smallest enclosing loop
- begins again a specified *number* of layers/loops out

continue *number*

```
$ for File in *
> do
> if [[ -d $File ]]
> then
>       continue
> fi
> file $File
> done
$ _
```

Figure 4-19. continue The Loop AU232.3

Notes:

Following a *continue* the command block is aborted, the next value is selected and the next iteration of the command block is begun — just as if it had completed the command block in full. So in the above example, when a directory file is found in the current directory, it is ignored: all other files are classified using the *file* command.

Continue is applicable to *until*, *while*, *for* and *select* constructs.

In the example above, the commands are entered against the dollar prompt, rather than in a Script. Clearly there are no files in the current directory.

If the number provided to the *continue* command is greater than the current block nesting depth, the Shell prints a warning and execution continues at the outermost block.

null Logic

Sometimes you require a command, but you don't actually want to do anything – a NULL command

```
:                    # a COLON character
```

```
sys_call parameter1 parameter2
if  [[  $?  -eq  0  ]]
then
        # Debug slot
        :
else
        print $0: Error: command failed
        exit $ERRNO
fi
```

without the null command ":" this would be illegal syntax

Figure 4-20. null Logic AU232.3

Notes:

Constructs like *if*, *until*, *while*, *for* and *select* require at least one command block. When you're debugging a program, null command slots can be handy — you can easily put in another print command without needing to change the logic of the enclosing construct.

You can have arguments to the null command, which will be expanded, and thus may affect the current environment. The return value is zero (true), so you can use the null command in place of the *true* keyword.

Program Logic Constructs Example

Here's a Script to delete empty files:

```
#!/usr/bin/ksh
# Usage: delfile file1 file2 ...
while [[ $# -gt 0 ]]
do
        if [[ -f "$1" ]]
        then
                if [[ ! -s "$1" ]]
                then
                        rm $1 && print $1 deleted
                else
                        print $1 not deleted 1>&2
                fi
        elif [[ -d "$1" ]]
        then
                print $1 is a directory
        else
                print "$1" is a special file
        fi
        shift
done
```

Figure 4-21. Program Logic Constructs Example AU232.3

Notes:

Here's delfile in action...

```
$ delfile /dev/null /tmp/john1 file1 file2 $PWD
/dev/null is a special file
/tmp/john1 deleted
file1 deleted
file2 not deleted
/home/john is a directory
$ _
```

A file can be deleted without write permission to it — write permission on its directory is all that is required. An attempt to delete a file will fail if its directory has no write access. The above example attempts to delete empty files and will report successful deletions.

No allowance is made for the non-existence of the named file — a special file is assumed.

Checkpoint (1 of 2)

1. What is wrong with this fragment of shell script?

```
if [ "$x" -eq 5 ]
then
      echo $x
elif [ "$x" -eq 3 ]
else
      echo "x is only 3"
      exit
fi
```

2. What is the fundamental difference between a **while** and an **until** construct?

3. How could we write an endless loop?

4. What syntax would we use to perform a loop a finite number of times, resetting an identifier each time?

5. Which construct is best suited to allow conditional processing, based on pattern matching?

Figure 4-22. Unit Checkpoint (1 of 2) AU232.3

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

Checkpoint (2 of 2)

6. What would the following lines produce?

```
select word in To be or not to be
do
      :
done
```

7. Which construct is best used within the previous **do-done**? block?

8. How can we terminate one iteration of a loop and commence the next?

9. How can we abruptly terminate all iterations of a loop but continue further processing in a shell script?

Figure 4-23. Unit Checkpoint (2 of 2) AU232.3

Notes:

6.

7.

8.

9.

Summary

- The *if – then – else* construct
- Conditional loops with *until* and *while*
- Specific value iteration with *for*
- Multiple choice pattern matching with *case*
- The *select* command for menus
- Leaving loops – *Exit* and *Break*
- Begining again – *Continue*
- Doing nothing — the null command – **:**

Figure 4-24. Summary AU232.3

Notes:

Unit 5. Shell Commands

What This Unit Is About

Creating an interactive script is a common activity for Korn Shell programming. This unit focuses on the print and read interactive commands as well as the set command.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Use the print and read commands
- Understand and use getopts
- Control the programming environment using the fc and set commands
- Use additional shell commands

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

We shall learn in this unit about some special built-in Shell commands:

- The Korn Shell print command
- Special printing characters
- The read command
- Option and argument processing with getopts
- Command line re-evaluation with eval
- History manipulations with fc
- The set command
- Shell options with set
- Shell invocation
- Built-in commands
- Shell commands provided by AIX

Figure 5-1. Objectives AU232.3

Notes:

The Print Command

The **print** command is the Korn Shell output mechanism:

print argument ...	prints arguments to standard output separated by spaces
print - argument ...	to print arguments that look like options
print -r argument ...	RAW mode – do not interpret special characters
print -R argument ...	equivalent to "-" and "-r"
print -n argument ...	no trailing newline after output
print -uN argument ...	output sent to file descriptor **N**
print -s argument ...	output to the shell history file only

Figure 5-2. The print Command AU232.3

Notes:

The Bourne Shell provides the *echo* command as an equivalent for *print*. The Korn Shell provides *echo* as a builtin command for backward compatibility. With AIX Version 4 *echo* has no options, while at Version 3 the "-*n*" option was provided with *echo*.

Arguments are optional — if you omit them, a blank line is printed — except with "-*n*".

Re-directing output with the "-*u*" option can be more efficient than using individual re-direction.

Options may be mixed in the usual way, except: no option can follow "-*r*", only "-*n*" can follow "-*R*".

There is another *print* option: `print -p argument ...` to output to a co-process. We shall not be looking further at this.

Special Print Characters

Backslash character sequences have special meaning (except in raw mode)

\a	Alarm - ring the terminal bell
\b	Backspace
\c	Print without trailing newline (same as print -n)
\f	Form feed
\n	Newline
\r	Return
\t	Tab
\v	Vertical tab
****	Backslash
\0xxx	Character with octal code **xxx** (up to three octal digits)

Figure 5-3. Special Print Characters AU232.3

Notes:

The backslash character is used to escape the special meaning of the following character. The Shell removes it when an unquoted command line is processed, so that you need two successive backslashes to pass a single backslash to the *print* command. The *print* command interprets the Shell processed line following the conventions listed above.

If you surround *print* arguments with quotes (single or double), the Shell does not strip away backslashes.

All of the above special characters work with the Bourne Shell *echo* command. However, *\a* was not provided with *echo* in AIX Version 3.

When you use *\c*, it must be the last option specified, that is *\r\c* not *\c\r*.

print Examples

When you use the **print** command, here's what you get...

```
$ print "Line 1\n\tLine2"
Line1
        Line2

$ print 'One quarter = \0274'
One quarter = ¼
$ print 'Backslash = \0134'
Backslash = \
$ print -r 'hi\\\\there 1'
hi\\\\there 1
$ print -r hi\\\\there 2
hi\\there 2
$ print 'hi\\\\there 3'
hi\\there 3
$ print hi\\\\there 4
hi\there 4
$ _
```

Figure 5-4. print Examples AU232.3

Notes:

In the '`hi\\\\there 1`' example, raw mode and single quotes prevent backslash interpretations by both the Shell and the *print* command. For the second example, there are no quotes used so the Shell processes the line and removes two backslashes. *print* processes the resulting line but as raw mode was specified the output is two backslashes.

For the third ('`hi\\\\there 3`') example, the Shell passes the input without processing to the `print` command. The command interprets the four backslashes passed to it from the Shell and prints two — since two backslashes input result in a single backslash output from *print*. Finally, the fourth example has both the Shell and then the *print* command interpreting the entered line — the Shell removes two backslashes and, without raw mode, two backslashes result in a single backslash output.

The read Command

To get input while a Shell Script is running, use **read:**

```
read variable ...
```

The read command reads a line from its standard input

- Assigns input words to the variables
- Set remaining variables to null if too few words
- Set last variable to the remainder of the words if too few variables

For the Korn Shell, if no variables are specified, the **REPLY** variable is set to the whole input line

Figure 5-5. The read Command AU232.3

Notes:

Standard input for the *read* command is normally the keyboard.

Words are delimited by a character from the *IFS* environment variable — space, tab or newline.

Apart from not using the *REPLY* variable, the Bourne Shell *read* command works in the same way.

read Examples

We can use **read** from the Shell prompt as well...

```
$ read var1 var2
123 456 789
$ print "var1 = $var1 \tvar2 =
$var2"              var2 = 456 789
var1 = 123
$ read var1 var2
abc
$ print "var1 = $var1 \tvar2 =
$var2"              var2 =
var1 = abc
$ read
hi there
$ print $REPLY
hi there
$ _
```

Figure 5-6. read Examples AU232.3

Notes:

Remember that you cannot change the value of a *readonly* variable.

The AIX Operating System provides the "*line*" command as an equivalent to the Shell commands:

```
( read ; print -R "$REPLY" )
```

If you require input to be taken from a terminal one character at a time — without the need to press return at each input — the *dd* command can be applied:

```
dd if=/dev/tty bs=1 count=$charcount > inputread
```

Here */dev/tty* is a link to the current terminal you are using, and *$charcount* has the number of characters you wish to take as input. In Unit 7 we will learn how to store the output of a command in a variable.

read Command Options

The Korn Shell **read** command has some options:

`read -r variable ...`	raw mode – \ is not taken as a line continuation character
`read -s variable ...`	record the input line in the history file and set variables
`read -uN variable ...`	read from file descriptor **N**

You can specify a prompt for the command to display on standard error
Add a "?prompt" to the first variable

```
read  variable?prompt  variable ...
```

For example, to request a user for a text string:

```
read string?'Please enter a text string'
```

Figure 5-7. read Command Options AU232.3

Notes:

Options above may be mixed in the usual way.

Before AIX Version 4, *read -r* did not require a variable — the *REPLY* variable would be used by default.

It may be more efficient to use the "*-u*" option, rather than normal command input re-direction.

There is another option where you can read from a co-process instead of standard input which we do not discuss further:

```
read -p variable ...
```

read Options Examples

```
#!/usr/bin/ksh
# Usage: readrun
# Prompt the user when asking for input.
read word1?"Enter some text : "  word2
print "Word1 = $word1 Word2 = $word2 \n"

$ readrun
Enter some text : The cursor appeared here
Word1 = The Word2 = cursor appeared here
$ _
#!/usr/bin/ksh
# Usage: readraw
# Read & print text_file in raw mode until EOF.
while read -r line
do
   print -R  "$line"
done  <  text_file
$ readraw
The first line of \ttext_file
-now the second
The last line of \ttext_file\t-\tend of file!\a
$ _
```

Figure 5-8. read Options Examples AU232.3

Notes:

You cannot use the *print* command special characters in the *read* command prompt string. In the top example, notice that the cursor stays on the same line as the prompt string. The prompt itself prints on standard error rather than standard output.

The bottom example shows how input terminates. End of file (*EOF*) for terminal input is normally <*Ctrl-d*>. When the *read* command gets *EOF*, it returns false. The example also shows that in raw mode read does not process the data.

Processing Options

Parameters on a script command line are of two types

- arguments – used in script

- options – used to tell the script things

General parameter/argument processing is difficult

Consider

```
$ myscript -a -f optionfile argfile
$ myscript -foptionfile -va argfile
```

Shell provides **getopts** as a solution

Figure 5-9. Processing Options AU232.3

Notes:

There is a general convention that options are prefaced by a "-"(sometimes a "+"). Arguments are the remainder of the parameters supplied to the program or script.

Processing arguments passed to programs and scripts is not too difficult provided that you have to parse only a small number of cases. The examples indicate two of the possible combinations of permitted options for myscript. Creating code for the two examples given is relatively easy but what happens if a new option is added?

The getopts Command

The **getopts** command processes options and associated arguments from a parameter list

```
getopts optionstring variable parameter...
```

- Each invocation of **getopts** processes the next option in the `parameter` list
  - usually called within a loop
- The `optionstring` lists expected option identifiers
  - if an option identifier requires an associated argument, add a colon (:)
  - a leading colon in the list suppresses "invalid option" messages by `getopts`

Figure 5-10. The getopts Command AU232.3

Notes:

Usually no *parameters* are specified on the *getopts* command line, so that the positional parameters are processed.

The *getopts* command uses your chosen variable "*variable*" and *OPTARG* and *OPTIND* to store the results of each processing operation on the parameters. *variable* contains the current option being processed or a "?" if it is not recognized as a valid option. *OPTARG* contains the string for an associated argument where a ":" has been added to an option identifier in *optionstring*.

The index variable *OPTIND* is not normally examined until the end of processing. Whenever a Korn Shell, or Shell Script, is invoked, the value of *OPTIND* is set to 1. When *getopts* recognizes the end of the options or reads a "--" option, processing of the parameters stops. At this point *OPTIND* indexes the first non-option parameter. By convention (see previous notes) this is the first proper argument.

Option parameters begin with a "+" or "-" and may contain several option identifiers, i.e. *-abc*. By convention, identifiers with a minus are used to set options: a plus means unset that option.

A "--" option can be used to mark the end of your option list. If you have argument parameters that look like options, you can use the "--" to prevent *getopts* from treating them as options. Unusually, this command returns zero (true) even when an error condition occurs.

getopts Syntax Example

How are options processed when passed to a script?

Assume

- The possible options are a, b and c
- Option b is to have an associated argument
- Suppress normal OpSys error messages

Inside the script **getopts** will be used early on:

```
while  getopts  ':ab:c'  flag arguments
do
          identify the values set by getopts
done
```

A correct command line to the script might be

```
$ prog.ksh  +c  -ab  barg  --  arg1  arg2
```

What about?

```
$ prog.ksh  -c -b -a -- arg1 arg2
```

Figure 5-11. getopts Syntax Example AU232.3

Notes:

The example shows three possible valid options — *a,b,c*. Option "b" should have an associated argument. Invoking `prog.ksh` in the first case gives no problems. Notice how *getopts* is able to deal with combined option flags like "-ab" without difficulty.

The second example is more tricky. At first glance it looks OK but there is a problem — what is it?

getopts Example

```
#!/usr/bin/ksh
#  Example of getopts
USAGE="usage:  example.getopts.ksh [+-c]  [+-v]  [-a argument]"

while getopts :a:cv arguments
do
case $arguments in
      a)   argument=$OPTARG ;;
      c)   compile=on ;;
     +c)   compile=off ;;
      v)   verbose=on ;;
     +v)   verbose=off ;;
     : )   print "You forgot an argument for the switch called a."; exit ;;
     \?)   print "$OPTARG is not a valid switch" ; print "$USAGE" ; exit ;;
     esac
done

print "compile is $compile; verbose is $verbose; argument is $argument "

#END
```

Figure 5-12. getopts Example AU232.3

Notes:

The problem on the previous example was that it is not clear whether the "-a" is the associated argument to "-b" or not.

Notice how using a leading ':' in the *getopts optionstring* means doing your own error processing. You do not have to exit with an invalid option but it's usually the best or safest course of action.

How do we get to the actual or proper arguments? Recall that *OPTIND* contains an index to the parameters processed. In particular, after all options have been processed it is "pointing" to the first non-option argument. The usual practice is then to use *shift* to shift over the option parameters by using the index.

```
shift (( OPTIND - 1 ))
```

works for Korn Shell. For the Bourne Shell, you can use

```
shift `expr $OPTIND - 1`
```

You will see these constructs in the next unit.

The eval Command

The Shell processes each command line read before invoking the relevant command(s).

If you want to re-read and process a command line, use **eval**:

- **Eval** processes its arguments as normal
- The arguments are formed into a space separated string
- The Shell then executes that string as a command line
- The return value is that of the executed command line

Figure 5-13. The eval Command AU232.3

Notes:

The *eval* command works in the Bourne Shell in the same way.

eval is a very powerful feature. It has been known for programmers to emulate their favorite command interpreters with a script based on using argument processing and *eval*.

eval Examples

Here are some eval command lines...

```
$ eval print '*sh'
getopts.example.ksh  eval.ksh         try.sh

$ message1=Goodbye                    print the message
$ message10=Hello                     named by $variable
$ variable=message10
$ eval print '$'$variable
Hello

$ print "ls | sort -r" > cmd_file
$ read -r line < cmd_file             read a cmd_file line
$ eval "$line"                        - run as a command
zfile
afile

$ cmd='ps -ef | grep tommy'           run a string command
$ eval $cmd                           to list tommy's processes
...
$ _
```

Figure 5-14. eval Examples AU232.3

Notes:

From a Shell Script, you can use *eval* with the positional parameters...

```
#!/usr/bin/ksh
# Usage: put [options] filename
# Test that the last argument is a filename.
if eval  [[ ! -f \${$#}  ]]
then
    print -u2  "File not found:"
    exit  1

fi
```

The fc Command

The Korn Shell fc command interactively edits and then re-executes portions of your command history file:

`fc start end`	edits and executes a command range – *start* defaults to the last command – *end* defaults to the value of *start*
`-e editor`	to specify an editor other than **$FCEDIT** - Shell default is /bin/ed

To re-execute a single command with automatic editing:

```
fc -e - old=new
command
```

- *old=new* to swap string *old* with string *new*
- *command* to specify a command - default last

Figure 5-15. The fc Command AU232.3

Notes:

The *HISTSIZE* environment variable sets the maximum *start finish* range size you can specify — 128 commands by default.

The *fc* command returns the value of the last command executed.

The *r* command is equivalent to *fc -e -*. With AIX Version 4, *fc -s* is also equivalent to *fc -e -*.

The *fc* command is less often used now but some of the inline command editing may be of interest.

fc Examples - Edit and Execute

Ranges may be strings, absolute or relative numbers...

`$ fc`	*edit the last command with the $FCEDIT editor, and then re-execute*
`$ fc pwd cc`	*edit with $FCEDIT from the most recent command starting with pwd, to one beginning with cc*
`$ fc -e vi 10 20`	*use vi to edit history lines 10 to 20*
`$ fc -e ex -3 -1`	*edit the last three commands with ex*

Automatic editing can specify a command in a similar way

`$ fc -e -`	*re-execute last command as was*
`$ fc -e - cc`	*re-run most recent cc command*
`$ fc -e - 2=3 10`	*swap 3 for 2 in command 10*
`$ fc -e - s=\? -2`	*change "s" into "?" in the command before last*

Figure 5-16. fc Examples - Edit and Execute AU232.3

Notes:

fc Examples - Lists

The Korn Shell fc command lists portions of your command history file:

fc -l start end	list the specified command range - the default is the last 16 commands
-n	suppress command numbers in list
-r	reverses the order of commands

For example...

`$ fc -l pg grep`	*lists commands from the last pg to a grep*
`$ fc -l 15 20`	*lists commands 15 to 20*
`$ fc -l -5 -1`	*lists the last five commands*

Figure 5-17. fc Examples - Lists AU232.3

Notes:

The *fc* command always returns true when commands are listed. It is equivalent to the *history* command. Indeed, you will see in unit 7 that it is an *alias*.

The set Command

We have seen three functions performed by the **set** command:

`set`	lists set variables with their values
`set value ...`	re-sets the positional parameters
`set -o vi`	enables Korn Shell line recall and editing

This last form sets a Korn Shell option. There are several more options to set:

- Korn Shell options and settings are listed by **`set -o`**
- Turn option on with **`set -o option`** or **`set -L`**
  (where **L** is an option identifier)
- Turn option off using **`set +o option`** or **`set +L`**

Figure 5-18. The set Command AU232.3

Notes:

The Bourne Shell also has some of the same options as the Korn Shell. The Bourne Shell *set* command does not have a "*-o*" option syntax; it uses the single letter option identifiers. Most of the option identifiers explained in the following pages are provided by the Bourne Shell — those that are not are noted in the text below.

Shell Options With Set

Option:	*L*	*Description:*
`allexport`	`a`	automatically export each variable set
`bgnice`		run all background jobs at a lower priority – this is on by default for interactive Shells
`ignoreeof`		stops an interactive Shell exiting on Ctrl-d – you must use the exit command
`noclobber`	`C`	stops the Shell overwriting existing files with > re-direction (>| works instead)
`noexec`	`n`	for a non-interactive Shell to check syntax without executing commands
`noglob`	`f`	disables metacharacter pathname expansion

Figure 5-19. Shell Options With set AU232.3

Notes:

The Korn Shell provides a *monitor* or *m* option. The "*C*" option was introduced with AIX Version 4 — for earlier systems "*noclobber*" has no option letter equivalent. The "*notify*" or "*b*" option is also new with AIX Version 4 — other systems have no equivalent.

The Korn Shell also provides a privileged or "*p*" option. However, this is not supported by AIX, as AIX does not allow *SUID* (set user id) shell scripts. The *privileged* option is very similar to the *protected* option that was only available with the 6/3/86 version of the Korn Shell (the same option by a different name for that version only).

On systems that do operate *SUID* shell scripts, the *privileged* or "*p*" option is on if the effective user or group ids differ from the real ones. It disables the processing of *$HOME/.profile* and *$ENV* — using */etc/suid_profile* instead. Turning the option off resets the effective ids to the real ones.

The Bourne Shell does not provide a *privileged* or "*p*" option.

Shell Options With Set (Cont.)

Option	*L*	*Description*
`notify`	`b`	to notify asynchronously of background job completions
	`s`	to sort positional parameters
`trackall`	`h`	set-up a tracked alias for each new command – on for non-interactive Shells
`verbose`	`v`	to display input on standard error as it is read
`vi`		turns on history line recall and **vi** editing
`xtrace`	`x`	the debug option – the Shell displays PS4 with each processed command line
`errexit`	`e`	exits if any command returns a non-zero return code
`nounset`	`u`	displays an error message when an unset variable is used

Figure 5-20. Shell Options With set (Cont) AU232.3

Notes:

The Bourne Shell does not provide an "*s*" sorting option.

Unit 7 deals with command aliases and the use of the "*trackall*" or "*h*" option. The Bourne Shell provides command hashing instead of aliases, which is where the "*h*" originates.

In addition to the "*vi*" option, "*emacs*" and "*gmacs*" options are available if the Korn Shell was compiled with these editors.

There is a "*keyword*" or "*k*" option, that allows "keyword parameters" to be used. These are variable assignments placed in front of a Shell Script invocation, that are passed to the Script:

```
$ variable=value ...  shell_prog argument ...
```

The use of "keyword parameters" is **strongly discouraged** — it is provided only for Bourne Shell compatibility, and may be withdrawn from future versions of the Korn Shell.

One important use for the *set* command is to assign values to Korn Shell arrays, using the *+A* and *-A* options. Arrays are covered in Unit 7, so we will leave this for later.

In the Korn Shell, to turn off the *xtrace* or "*x*" and *verbose* or "*v*" options, and prevent further arguments being interpreted as options, use:

```
set - argument ...
```

To simply prevent arguments being treated as options (without affecting any Shell options), use the "--" syntax:

```
set -- argument ...
```

The "*interactive*" option is listed by a *set* command. However, this option is a Shell invocation option, and cannot be altered with the *set +o option* or *set -o option* syntax.

Set Quiz

1. What command would you use to re-set the positional parameters to "one" "two" "three"?

2. What lists the Shell options with settings?

3. Which *set* option ensures that each variable assignment will be inherited by a sub-Shell?

4. What would stop <Ctrl-d> from logging me out?

5. How can I use *set* to protect my files from being overwritten by output re-direction?

Figure 5-21. set Quiz AU232.3

Notes:

Shell builtin Commands

We have seen the following builtin Shell commands:

`.`	`:`	*`bg`*	`break`
`cd`	`continue`	**`echo`**	`eval`
`exec`	`exit`	`export`	*`fc`*
`fg`	*`getopts`*	*`jobs`*	**`kill`**
`print`	**`pwd`**	**`read`**	`readonly`
`set`	`shift`	**`test`**	**`[ ]`**
`trap`	`typeset`	**`unset`**	**`wait`**

In the later units we will see:

`alias`	*`command`*	*`let`* or *`(( ))`*	`return`
`times`	**`ulimit`**	*`unalias`*	*`whence`*

All builtin commands can run in the current environment

Special builtin commands may terminate the Shell if an error occurs

Figure 5-22. Shell builtin Commands AU232.3

Notes:

Variable assignments made with the underlined special builtin commands remain effective after the commands complete — that would not be the case for regular builtin commands. Command re-directions are processed after parameter assignments with special builtin commands only. The "." and ":" special builtin commands won't terminate the current shell when in error — other special builtin commands will. Italicized commands above are not available in the Bourne Shell. The "*command*" command was introduced with Korn Shell for AIX Version 4.

Bourne Shell has builtin commands for its special features too (these are beyond the scope of this course): *hash*, *login*, *setxvers*, *type*. The *wait* command is a special builtin for the Bourne Shell.

The Korn and Bourne Shells also provide the following commands (explained in AU14):

`umask`	to set and display default file creation permissions
`newgrp`	to change the effective group id, so that created files are associated with that group.

Operating System commands must always run in a Sub-Shell.

AIX Shell Commands

Some built-in Korn Shell commands are also provided as AIX commands – accessible from all Shells:

`alias`	`bg`	`cd`	`command`
`echo`	`fc`	`fg`	`getopt`
`jobs`	`kill`	`newgrp`	`read`
`umask`	`unalias`	`wait`	

AIX commands are also provided for the logical words:

`false` **`true`**

Most of these commands are shell scripts in /usr/bin – they are provided for POSIX compliance

Figure 5-23. AIX Shell Commands AU232.3

Notes:

By default, the Korn Shell will use its own builtin commands instead of AIX ones of the same name. To specify the AIX ones, you could use a full pathname: e.g. */usr/bin/jobs.*

Before AIX Version 4, the following commands were **NOT** normally implemented by the operating system: *alias*, *bg*, *cd*, *command*, *fc*, *fg*, *getopts*, *jobs*, *read*, *umask*, *unalias* and *wait*. It should however, be an easy matter to write missing mini-Shell-Scripts for a system.

As we shall see in Unit 7, *true* and *false* are not Shell builtin commands as such.

The operating system also provides a *getopt* command (note spelling) that performs a similar function to the Korn Shell *getopts* builtin command. Because it is provided by the operating system, it is accessible in all Shells.

Checkpoint

1. Without using redirection, how could we print information to file descriptor 2?

2. What is wrong with the following command?
   `read speed?"mph" distance?"miles"`

3. What **getopts** statement would allow you to process options **p**, and **a**, with option **t** expecting an associated value?

4. What command would print out the first and last positional parameters?

5. Which **set** option disables metacharacter pathname expansion?

6. Which **set** options would be most useful in helping to debug a shell script?

Figure 5-24. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

Summary

- The Korn Shell print command
- Special printing characters
- The *read* command
- Option and argument processing with *getopts*
- Command line re-evaluation with *eval*
- History manipulations with *fc*
- The *set* command
- Shell options with *set*
- Shell invocation
- Builtin commands
- Shell commands provided by AIX

Figure 5-25. Summary AU232.3

Notes:

Unit 6. Arithmetic

What This Unit Is About

This unit presents the three ways of doing arithmetic operations in Shell — expr, let and bc.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Calculate using expr, let or (( )) and bc

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

In this unit we will learn how to do arithmetic in the Shell.

- The expr utility
- Expr arithmetic and logical operators
- Korn Shell let or (( ))
- Number bases
- Let logical operators
- Integer variables
- Implicit let
- The bc utility

Figure 6-1. Objectives AU232.3

Notes:

expr Arithmetic

AIX provides the **expr** utility to perform ***integer*** arithmetic

```
expr  argument1  operator  argument2 …
```

expr features

- runs in a Sub-Shell – not a Shell builtin command
- writes results to standard output
- exit code is 0 for non-zero evaluations
- exit code is 1 for zero or null evaluations
- exit code is ≥ 2 if an expression is invalid

Mostly used for control flow in shell scripts – loop counters

Figure 6-2. expr Arithmetic AU232.3

Notes:

As the *expr* utility is provided by the operating system, it is available from all Shells.

In AIX Version 4 error conditions result in an exit code greater than 2, while AIX Version 3 gives 2.

Internally numbers are treated as 32-bit two's complement integers, but are held and output as character strings.

Remember that there are two "results" — that on standard output and the command exit status. *Expr* also performs pattern matching and string manipulations — we will not be covering these aspects. See the man page if you are interested.

expr Arithmetic Operators

To group expressions use:

`( )` fixes evaluation order - otherwise normal rules of precedence apply

The integer operators result in mathematical evaluations:

`*`	multiplication
`/`	integer division
`%`	remainder
`+`	addition
`-`	subtraction (also unary minus sign)

NOTE: Use of backslash?

Figure 6-3. expr Arithmetic Operators AU232.3

Notes:

Expr only does **integer** arithmetic.

You must use a backslash or quotes to protect special characters from the Shell, e.g. `\*`.

Spaces are required between operators and expressions — except for the unary minus with a literal value: e.g. `-3`.

Operators are shown here in order of precedence: highest to lowest.

expr Logic Operators

For integers or strings the following result is 1 for true, 0 for false:

=	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Logic operators & (and) and | (or) give different output:

`expr LHS \& RHS`	"and" - results in LHS if both sides are non-zero, 0 otherwise
`expr LHS | RHS`	"or" - evaluates to LHS if it is non-zero, otherwise to RHS

Figure 6-4. expr Logic Operators AU232.3

Notes:

For the logical operators, if both expressions are integers, numerical evaluation is performed. If character strings are present, ASCII character order is used. Notice the odd standard output values — opposite to the true=0, false=non-zero command exit codes.

expr Examples

Here is some simple integer arithmetic...

```
$ var1=6; var2=3
$ expr  $var1  /  $var2
2
$ expr  $var1  -  $var2
3
$ expr  \(  $var1  +  $var2  \)  \*  5
45
$ _
```

What is the result of the following?

```
$ expr  10  %  3

$ expr  10  /  3
```

Figure 6-5. expr Examples AU232.3

Notes:

Notice that ***everything*** is an argument to *expr*. Make sure you have whitespace around parameters.

expr Examples (Cont.)

Some logical examples...

```
$ expr  abc  \<  def
1                          meaning true with expr
$ expr  3  \>=  4
0                          meaning false
$ value=4
$ expr  5  !=  $value
1
$ _
```

What is the result of the following?

```
$ expr  10  \|   3

$ zero=0
$ expr 10 \&   1  +  $zero
```

Figure 6-6. expr Examples (Cont) AU232.3

Notes:

For those who understand C programming, the result on standard out has the same value as in a C program. You need to be careful using *expr* with logical expressions. The meaning (the semantics) of the "and" and "or" operators is different from that found in strict logic.

The Korn Shell let Command

```
let argument ..

-or-

((  argument  ))
```

- The *let* built-in Shell command performs long integer arithmetic approximately 10 times faster than *expr*
- Evaluates each argument as an arithmetic expression
- No quotes for special characters, or arguments with spaces or tabs in them, within (( ))
- Variables need no $
- The exit code is 0 (true) for non-zero, and 1 (false) for zero evaluations

Figure 6-7. The Korn Shell let Command AU232.3

Notes:

As multiple arguments are space or tab separated for the ordinary *let* form, you must quote such characters if they appear in an expression.

The *(( ))* form of the command may have only one argument.

let Arithmetic Operators

For simple arithmetic:

(	)	overrides normal precedence rules
	*	multiplication
	/	division
	%	remainder
	+	addition
	-	subtraction (or unary minus)
	=	assignment

`var op= exp` means `var = var op exp`

Upto nine levels of nested processing will be evaluated:

```
$ z=2  ;  y="z + 1"
$ ((  x=3*y  ))
$ print  $x
9
$ _
```

Figure 6-8. let Arithmetic Operators AU232.3

Notes:

A null variable equates to zero. Shell variables do not need the $. When using the (( )) form, there is no standard output. To keep the result you must save it in a variable.

Operators are listed in order of precedence. The unary minus is evaluated after *( )* and both are evaluated before the other simple operators above. The assignment operator has the lowest precedence of all.

base#number Syntax

With **let** you are not limited to just decimal (base ten) integers:

- **let** constants are of the form ***base*#number**
- ***base*** is an integer in the range 2 to 36 (10 default)
- **number** may include upper or lower case letters for bases greater than 10

2#100 in binary	=	4 in base 10
8#33 in octal	=	27
16#b in hexadecimal	=	11
16#2A in base16	=	42

Figure 6-9. base#number Syntax AU232.3

Notes:

Ways to do your octal or hexadecimal arithmetic perhaps?

let Arithmetic Examples - 1

Some simple
arithmetic...

```
$ a=1
$ b=2
$ ((  z = 2#10 + -b ))        unary minus needs a space before it
$ let  c=a+b  d=b\*b          no spaces, but \ needed for *
                              multiple arguments
$ (( e  =  9 / 2#10 ))        integer division
$ (( e  +=  a ))              assignment: addition
$ print $z $a $b $c $d $e
```

What do you think we get?

Figure 6-10. let Arithmetic Examples - 1 AU232.3

Notes:

What is the difference between *$((...)) and ((...))*?

let Logical Operators

Logical expressions evaluate to 1 if true, 0 if false
(the exit code is 0 for non-zero, 1 for zero – as expected):

`!`	logical negation
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`!=`	not equal to
`&&`	logical "and" = 1 if both LHS and RHS are true (RHS not evaluated if LHS is false)
`||`	logical "or" = 1 if either LHS or RHS are true (if LHS is true, RHS not used)

Figure 6-11. let Logical Operators AU232.3

Notes:

let 0 (zero) returns 1 (false) — which is equivalent to the Korn Shell *false*.

Operators are listed in order of precedence. The logical negation operator has the highest order of precedence after *( )* and the unary minus. Other operators above have a lower order of precedence than the simple arithmetic operators. Notice that these operators have correct logic semantics.

let Logical Examples

```
$  (( p = 9 ))

$  (( p = p * 6 ))
$  print $p
54

$ (( p > 0 && p <= 10 ))
$ print $?
1

$ q=100
$ (( p < q || p == 5 ))
$ print $?
0

$ if (( p < q && p == 54 ))
> then
> print TRUE
> fi
TRUE

$ _
```

Figure 6-12. let Logical Examples AU232.3

Notes:

Follow the flow of the variables.

In the first two examples, the variable is assigned to a value. Numeric expressions are tested in the other examples, using "both true" and "either - or" operators. Finally, an *if* statement precedes the *let* command used for conditional testing.

Korn Shell integer Variables

Korn Shell variables are stored as character strings unless defined with the *integer* command

```
integer variable=value ...
        -or-
typeset -iN variable=value ...
```

- Sets the **integer** attribute for each variable
- *typeset* can define a base ***N***, variables then print in the specified base (2 to 36)
- Assignment to an **integer** variable causes expression evaluation – an implicit *let* command
- **let** does not have to convert **integer** variables from character strings to numerical values

Figure 6-13. Korn Shell integer Variables AU232.3

Notes:

We shall see more of the *typeset* command in the next Units. Both *typeset* and *integer* are Korn Shell commands.

integer Examples

Some examples of **integer** and **typeset -i**...

```
$ integer x                   x can hold only integers
$ x=string
ksh:  string: 0403-009 The specified number is
not valid for this command.
$ x=5+10                      implicit let command
$ print $x
15
$ (( x = 5 + 100 ))
$ print $x
105
$ typeset -i8 nums0 nums1 nums2
$ nums0=8#5                   define an octal integer variable
$ nums1=8#10
$ (( nums2=8#3*nums0 ))       assign value
$ print ${nums2}
8#17
$ x=${nums2}
$ print $x                    print gives answer in base 10
15
$ _
```

Figure 6-14. integer Examples AU232.3

Notes:

An ordinary *integer* variable assumes the base of its first value assignment — base 10 for *x* in above example.

Implicit let Command

integer variable assignments are an implicit *let* command
Other implicit let commands are:

- Values for the Korn Shell **shift** command

```
shift OPTIND-1
```

- Resource limits with **ulimit**

```
ulimit -t TMOUT+60
```

Figure 6-15. Implicit let Command AU232.3

Notes:

The *ulimit* command is a Shell builtin command: *ulimit -a* displays current settings. Other options are:

`-c N`	core dump size limit (512 byte blocks),
`-f N`	file size limit for all child processes (512 byte blocks),
`-d N`	data area size limit (kilobytes),
`-s N`	stack area size limit (kilobytes),
`-m M`	physical memory limit (kilobytes),
`-t N`	time limit in seconds.

You have already seen the implicit let usage with *OPTIND*. There is one other use — in connection with arrays which we cover in the next unit.

bc - Mathematics

The AIX system provides the bc utility

```
bc   [file]
```

- performs floating point arithmetic
- acts as a filter command or interactively
- reads arithmetic expression strings from standard input or a specified file
- semicolons or new lines separate expressions
- set the **scale** variable inside **bc** to define the required number of decimal places
- prints results to standard output

Figure 6-16. bc - Mathematics AU232.3

Notes:

The *bc* command works in decimal, octal or hexadecimal. Set the variables *ibase* and *obase* to specify the input and output number bases respectively.

Caution: base conversion will not work for Hexadecimal to decimal, or octal to either of the other bases.

Another caution: *bc* is not good for financial figures.

bc Operators

For simple arithmetic and logical evaluations, use:

`(, ), +, -, *, /, %, =`	as for **let** arithmetic operators
`==, !=, <, <=, >, >=`	as for **let** logical operators
`x^y`	raise x to the power y
`sqrt (x)`	square root
`x++ ++x`	post and pre increment x
`x-- --x`	post and pre decrement x
`x op= y  ≡  x = x op y`	for +=, -=, *=, /=, %=, ^=

A library provides complex mathematical functions:

`s(x)`	sine of x
`c(x)`	cosine of x
`e(x)`	natural exponential of x
`l(x)`	natural log of x
`a(x)`	arctangent of x
`j(n,x)`	Bessel function

Precision functions:

`length(n)`	number of significant digits	E.g. 123.456 has n=6
`scale(n)`	number of digits after decimal point	E.g. 123.456 has n=3

Figure 6-17. bc Operators AU232.3

Notes:

To use the complex mathematical functions, you may need to specify the "*-l*" option to *bc* on the command line.

It is also possible to define complex functions of your own, in a C-language like syntax.

Logical flow control is also provided in *bc* — again in C-language structures.

Comments may be included in complicated files using the /**comment* */ C notation.

bc Examples

Here are some examples of **bc** working both as a filter and interactively...

`$ print '1/4' | bc`	*integer division without a scale*
`0`	
`$ print 'scale = 3 ; 1/4' | bc`	*explicit scale value set*
`0.250`	
`$ print '5.5 * 2.2' | bc`	*scale set implicitly from input*
`12.1`	
`$ bc`	
`sqrt( 4 )`	*no prompt – this is my input*
`2`	*the result from the command*
`Ctrl-d`	*to end interactive mode*
`$ _`	

Figure 6-18. bc Examples AU232.3

Notes:

Checkpoint

1. Multiply together variables **a** and **b**, using **expr**.
2. Use **expr** to multiply variable **a** by the sum of **b** and **c**.
3. Set variable **hex** to contain the hexadecimal value **7c**.
4. Write a **let** statement to test whether variable **a** is smaller than variable **b**.
5. Define a variable **num** as numeric only.
6. Increment a numeric variable **numvar**, by three.
7. How would you calculate 6/7 to 6 decimal places?
8. How would you calculate the square root of 178356025?

Figure 6-19. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.
2.
3.
4.
5.
6.
7.
8.

Summary

- The *expr* utility
- *Expr* arithmetic and logical operators
- Korn Shell *let* or *(( ))*
- Number bases
- *let* logical operators
- Integer variables
- Implicit *let*
- The *bc* utility

Figure 6-20. Summary AU232.3

Notes:

Unit 7. Korn Shell Types, Commands and Shell Functions

What This Unit Is About

This unit describes Korn Shell arrays, command substitutions, functions and variables, and aliases.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Use array variables
- Use command substitution
- Define and call functions
- Use typeset variables
- Process aliases
- Understand Shell command line processing

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

This unit describes Korn Shell arrays and takes an in-depth look at commands and their use

- Korn Shell arrays
- Command substitution
- Functions
- Typeset command
- Autoload functions
- Command aliases
- Pre-set aliases
- Tracked aliases
- The whence command
- Command line processing

Figure 7-1. Objectives AU232.3

Notes:

Defining Arrays

The Korn Shell supports one-dimensional arrays:

- arrays need not be "declared"
- access an element of an array by a subscript to a variable name
- any variable with a valid subscript becomes an array
- a subscript is an expression enclosed within [ ]
- subscripts should lie in the range 0 to 4095
- variable attributes (e.g. **readonly**) apply to all elements of the array

Caution: an entire array cannot be exported, only the 0th element

Figure 7-2. Defining Arrays AU232.3

Notes:

All variables are arrays in Korn Shell but because the "default" is element zero then

```
VAR1 == VAR1[ 0 ]
```

Assigning Array Elements

Just like ordinary variables, values can be assigned, and later referred to:

- assign contents to an array element using

  array[N]=*argument*

- to **unset** an array and assign new values sequentially, use

  set -A array *argument* ...

- to simply replace existing array values with new ones, use

  set +A array *argument* ...

Figure 7-3. Assigning Array Elements AU232.3

Notes:

Korn Shell variable names and contents are not limited in length — this applies to array elements also.

You can unset an array by: *unset array* — specifying the array name is enough.

Referencing Array Elements

The $ notation is used to refer to the value in a variable:

- when referencing an array element use { } notation

```
print  ${array[N]}
```

- to refer to all the elements of an array use an `*` or @ subscript (to give a space separated list)

`${array[*]}`   or   `${array[@]}`

- if you omit a subscript, it means the zeroth element

`${array[0]}`   ==   `$array`

Figure 7-4. Referencing Array Elements AU232.3

Notes:

Just as for positional parameters, where:

```
        "$@" = "$1" "$2" ...
and     "$*" = "$1 $2 ..."
```

with array elements:

```
        "${array[@]}" = "${array[0]}""${array[1]}" ...
and     "${array[*]}" = "${array[0]}${array[1]} ..."
```

Array Examples

`$ list[0]="Line 0"` `$ list[1]="Line 1"` `$ list[3]="Line 3"`	*Fill the array list.*
`$ print $list` `Line 0`	*Print the zeroth element.*
`$ print ${list[*]}` `Line 0 Line 1 Line 3`	*Print all elements.*
`$ print ${list[0]}` `Line 0` `$ print ${list[1]}` `Line 1`	*Print elements individually.*
`$ print ${list[2]}`	*Element [2] is null.*
`$ print ${list[3]}` `Line 3`	
`$ print $list[1]` `Line 0[1]` `$ _`	*Without { } notation, we get "$list" + "[1]".*

Figure 7-5. Array Examples AU232.3

Notes:

Another Array Example

Here we have the beginnings of a card game...

```
#!/usr/bin/ksh
# Usage: pickacard.ksh
# To choose a random card from a new deck
integer number=0
for suit in CLUBS DIAMONDS HEARTS SPADES
  do
  for n in ACE 2 3 4 5 6 7 8 9 10 JACK QUEEN KING
    do
      card[number]="$n of $suit"
      number=number+1
    done
  done
print  ${card[RANDOM%52]}

$ pickacard.ksh
QUEEN of DIAMONDS
$ _
```

Figure 7-6. Another Array Example AU232.3

Notes:

The lines picked out in italic or bold italic have implicit lets which were covered in the Unit 6. With an implicit *let* you don't need the dollar to reference shell variables.

Command Substitution

Command substitution allows you to use the output of a command or group of commands:

- in a variable assignment
- in part of an argument list

Bourne	`variable=`command``
	- or -
Korn	`variable=$(command)`

Nesting is possible:

```
var=`cmd1 \`cmd2 \\\`cmd3\\\` \` `
```

- or -

```
var=$(cmd1 $(cmd2 $(cmd3) ) )
```

Figure 7-7. Command Substitution AU232.3

Notes:

Both Bourne and Korn Shells have available the first form using grave accents, more usually called back quotes. The second is Korn Shell specific syntax. Clearly nesting is easier with the Korn Shell form!

If you use a *case* statement with the Korn Shell `$( ... )` command substitution, you must use the optional "*(*"in front of each pattern. Be careful to leave spaces around brackets, to avoid confusion with the double parenthesis form of the *let* command (as in (( ))).

Substituted commands run in Sub-Shells, but you can use re-direction in place of a command...

Command Substitution Examples

Here is command substitution in action...

```
$ d=$(date)
$ print  $d
Tue Feb 29 02:29:00 EST 2000
$ print "Contents of a file" > tmp_file
$ c=`cat tmp_file`
$ r=$(< tmp_file)                    no command, no Sub-Shell, so faster
$ print  "Cat: $c \n<: $r"
Cat: Contents of a file
<: Contents of a file
$ print "Most recent file: $(ls -t | head -1)"
Most recent file: tmp_file
$ arg1=1  ;  arg2=2
$ answer=$(expr $arg2 \* $(expr $arg1 + 3) )
$ print $answer
8
$ _
```

Figure 7-8. Command Substitution Examples AU232.3

Notes:

Inside backquotes (grave accents), a backslash normally only removes the special meaning of: \, ' or *$*.

Between backquotes that are themselves double quoted, the backslash also removes the special meaning of a double quote, for example:

```
var="output $(print \"text to print\") "
```

Defining Functions

Commands can group together and be named
The set of commands form the function body
Function definitions look like:

Bourne	***Korn***
`identifier()` `{` *`commands`* `}`	`function identifier` `{` *`commands`* `}`

Functions

- provide a means of breaking down programs into discrete units
- stored in memory for fast access
- executed, like new commands, in the current environment

Figure 7-9. Defining Functions AU232.3

Notes:

A function must be defined before it is used — i.e. put the definitions at the top of a Shell Script.

In the Korn Shell, functions may have the same name as that of a Script variable: in the Bourne Shell, this is not possible.

Don't use reserved words in a function name: *!*, *{*, *}*, *case*, *do*, *done*, *elif*, *else*, *esac*, *fi*, *for*, *function*, *if*, *in*, *select*, *then*, *time*, *until*, *while*, *[[,]]*. You cannot create a function with the same name as a special shell builtin command. If you give a function the same name as a regular builtin command, and use that command within the function definition, recursion occurs.

The Korn Shell "*command*" command (introduced with AIX Version 4) suppresses function lookup — this allows you to avoid recursion within a function. There is an example of *command* usage later.

Functions and Variables

Functions have different variables to the main Script:

- arguments
  - taken as positional parameters to the function
  - calling script **`$1-${n}`** parameters are reset on leaving the called function
- variables
  - declared with the **typeset** or **integer** commands (inside a Korn Shell function) are "local" variables to the function
  - all other variables are "global" in the Script
  - the "scope" of a "local" variable includes all functions called from the current function

Figure 7-10. Functions and Variables AU232.3

Notes:

Inside a function *$** and *$@* refer to the arguments to the function.

$0 in a function refers to the name of that function. Previous versions of the Shell in AIX may not have this behavior.

Local variables do not exist in the Bourne Shell. More on the *typeset* command later in this unit.

Normally all variables in a Shell Script are "global" — that is accessible anywhere in the script.

function Examples

Some useful functions...

```
$ function cd
> {
>     command cd "$@"         - command stops recursion
      PS1="`pwd` : "          - PS1 is set to "/tmp : "
> }
$ cd /tmp
/tmp : cd /                   - the new prompt appears
/ : _                         - and will follow us around
```

```
#  Handy for usage errors in Shell Scripts
#  Invoke function usage with arguments: script
#  followed by arglist.  Note exit status!
function usage
{
    prog="$1"; shift
    print -u2 "$prog: usage: $prog $@"
    exit  1
}
```

Figure 7-11. function Examples AU232.3

Notes:

In the Korn Shell we have *$PWD* set to the current directory always, so in `/etc/profile` or our own login `$HOME/.profile` file we can put:

```
PS1='$PWD : '
```

The single quotes set the literal value of *$PWD*, which is then expanded when *$PS1* is printed. In the Bourne Shell we cannot use *$PS1* in this way, but the *cd* function example above would work.

In earlier versions of the Korn Shell the only way to prevent recursion would be to rename the function. Function names should generally be chosen carefully, so as to be both descriptive and safe. AIX Version 4 also provides *command* as an operating system command available to all Shells.

Ending Functions

A function completes after executing the last command:

- the exit code is normally that of the last command
- **return** can be used to specify an exit code *N*, or just end the function at that point

  `return N`

- **exit** will terminate the current function and current Shell

  `exit N`

- errors within a Korn Shell function cause it to return control and the error exit code to the calling Script

Functions may be deleted from memory using...

```
unset -f functionname
```

Figure 7-12. Ending Functions AU232.3

Notes:

In the Bourne Shell function errors abort the Script, like an *exit* command.

Functions and Traps

The behavior of **trap** with functions is determined by the Shell type:

Bourne: a **trap** is "global" – the same in and out of a function

Korn: a trap is "local" to a function and is reset on completion

a main program trap is not shared with functions

a signal that is not caught or ignored, may cause the script to terminate

a signal that is ignored by a Korn Shell, is also ignored by functions called from it

Figure 7-13. Functions and Traps AU232.3

Notes:

Before AIX Version 4, only main program *ERR* and *EXIT* traps were not shared with functions. Where a signal was neither caught nor ignored, the condition would be passed back to the calling program.

A signal that is ignored by the main Shell cannot be trapped by any Sub-Shell — it is always ignored.

The typeset Command

The Korn Shell typeset command defines or lists variables and their attributes:

```
typeset ±LN variable1=value1 variable2=value2 ...
```

omitting variables lists variables with specified attributes

- sets attributes, or lists names and values

+ unsets attributes, or lists just names

Where **L** is any of ...

`r`	the **readonly** attribute – no modification of variables' value
`i`	sets the **integer** attribute – use with *N* to set number base
`x`	the **export** attribute – the variable will be exported

Figure 7-14. The typeset Command AU232.3

Notes:

Attributes are set, or unset, after assigning optional values to specified variables.

"`-H`" sets the pathname mapping attribute — on non-UNIX systems pathnames are converted into host system names.

We saw the "-*i*" option used in the last Unit.

typeset Examples

Declare arrays to specify:

- size
- attributes

```
$ typeset -xi8 a2[1]          exported & octal integer
$ a2=52
$ a2[1]=25
$ ksh
$ print $a2 ${a2[1]}
8#64                          only element 0 was exported
$ _
```

Inside a Korn Shell function, **typeset** creates a "local" variable...

```
# Function to convert numbers into binary
function  binary_convert
{
     typeset -i2 binary=$1
     print "$1 = $binary"
}
```

Figure 7-15. typeset Examples AU232.3

Notes:

If you create a "local" variable with the same name as a "global" one, the two variables are distinct.

To list variables with the readonly attribute...

```
$ typeset +r
LOGNAME
$
```

typeset With Functions

Other uses of **typeset** are:

- display functions
- set function attributes
- unset function attributes

typeset ±fL function1 function2 ...

- to list functions with specified attributes, omit function list
- ***-f*** sets attributes, or displays function names and definitions
- ***+f*** unsets attributes, or displays only function names

Where ***L*** is any of...

x	the **export** attribute – the function will be available to implicit Shells invoked from the current one
u	to mark a function as undefined
t	the Shell **xtrace** option for a function

Figure 7-16. typeset With Functions AU232.3

Notes:

You must be using your history file for the listing options to work: the Shell *nolog* option must be off when function definitions are read.

Functions that are to be defined across explicit invocations of a Shell should be defined in the *$ENV* file, with the *export* attribute — so that they are available to subsequent shells (implicit or explicit).

The return value is true if you specify "*-u*" or all function names, otherwise it is the number of non-function names you specify.

The "*-u*" will "reserve a slot" for future definitions. The xtrace option ("*-t*") is clearly useful for debugging.

typeset with Functions Examples

```
$ typeset -f                    lists functions in full
function list
{
     while [[ "$1" != "X" ]]
     do
          print  $1
          shift  1
     done
}
$ typeset -fx list              export the list function
$ typeset +f                    lists function names
list
$ _
```

Figure 7-17. typeset with Functions Examples AU232.3

Notes:

In the next unit we will see more uses for *typeset*.

autoload Functions

A Korn Shell function that is defined only when it is first called, is an **autoload** function:

```
autoload function
```

- or -

```
typeset -fu function
```

- using **autoload** functions improves performance
- the Korn Shell searches directories listed in the **FPATH** variable for a file with the name of the called function
- the contents of that file then defines the function
- existing function definitions are not unset

Figure 7-18. autoload Functions AU232.3

Notes:

By putting several related function definitions in a file, and using the operating system *ln* command to create multiple names for the file, you can *autoload* "libraries" of functions. The multiple names are those of the functions in the file of function definitions.

Aliases

The Korn Shell **alias** facility provides:

- a way of creating new commands
- a means of renaming existing commands

Creation:	`alias name=definition`
Deletion:	`unalias name`

An **alias** definition may contain any valid Shell Script or metacharacters

Figure 7-19. Aliases AU232.3

Notes:

Like functions, aliases must be defined before they are used — so put definitions at the top of Shell Scripts.

You may re-define Shell builtin commands using aliases, but don't use aliases for reserved words.

Reserved words are: *! {*, *}*, *case*, *do*, *done*, *elif*, *else*, *esac*, *fi*, *for*, *function*, *if*, *in*, *select*, *then*, *time*, *until*, *while*, *[[*, *]]*.

In AIX Version 4, all aliases can be removed with a single command: *unalias -a*.

Processing Aliases

Command lines are split into words by the Shell:

- check the first word of each command line for a defined **alias**
- a backslash in front of a command name prevents **alias** expansion if the alias exists
- if the **definition** ends in a space or tab, the next command word will also be processed for **alias** expansion
- **resolve alias** names within a function when function definitions are read – not at execution!

Figure 7-20. Processing Aliases AU232.3

Notes:

Definitions must be quoted to include spaces or tabs.

Preset Aliases

Korn Shell uses the following exported aliases

- may be unaliased or redefined

```
alias autoload='typeset -fu'
alias false='let 0'
alias functions='typeset -f'
alias hash='alias -t'
alias history='fc -l'
alias integer='typeset -i'
alias nohup='nohup '          with trailing space
alias r='fc -e -'
alias true=:
alias type='whence -v'
```

Figure 7-21. Preset Aliases AU232.3

Notes:

It is not a good practice to alter the above aliases, it will confuse other programmers if nothing else.

We shall see what *hash* and *whence* do in a moment.

The alias Command

The **alias** command has some options:

`alias -L name=definition`

Where ***L*** is any mix of...

`x` to set, or display exported aliases

`t` to set, or list tracked aliases

If ***definition*** is quoted...

"definition" interpreted when entered

'definition' text stored for later interpretation

Figure 7-22. The alias Command AU232.3

Notes:

A backslash can be used inside a "*definition*" to prevent recursion for a command. Single quotes around the whole *definition* have the same effect.

Tracked aliases are covered in a moment.

An exported alias is passed to Shells invoked from the current one. However, to export an alias across different explicit Shells, you must define and export it from the *$ENV* file. Explicit means wherever you can see "*ksh*" in the invocation — e.g. `ksh`, `ksh -c "commands"`, `ksh prog`. Alternatively, running a script that has the special "*#]/usr/bin/ksh*" comment as its first line will invoke a new explicit Shell.

Notice what happens when you use single or double quotes. In most cases you will want single quotes so that any interpretation occurs when the alias expands later.

alias Examples

```
$ alias -x ls='ls -a'          ls is set and exported
$ x=10
$ alias px="print $x" rx='print $x'
$ x=100
$ px                           prints $x as it was
10
$ rx                           prints the latest $x
100
$ alias od=done                an alias for some flow control
$ for i in lazy done
> do
>       print $i
> od
lazy
done
```

Figure 7-23. alias Examples AU232.3

Notes:

```
$ alias cd=_cd
$ function _cd
> {
>       \cd "$@"              #preventing recursion with a \
>       print "$OLDPWD --> $PWD"
> }
$ _
```

The cd command alias results in the following:

```
$ cd /tmp
/ --> /tmp
$ cd /home/root
/tmp --> /home/root
$ _
```

Tracked Aliases

A "tracked alias" reduces the search time for a future use of a command

`set -o trackall` **or** `set -h`

turns on Shell **trackall** option

First use of a command creates tracked alias

Force creation with

```
alias -t name
```

List all "tracked aliases"

```
alias -t
```

NOTE: the value of a "tracked alias" becomes undefined when the PATH variable is reset

Figure 7-24. Tracked Aliases AU232.3

Notes:

Once created, a "tracked alias" will obscure a new command of the same name if it is placed in the command search *PATH*, in a directory that is before that of the original command.

Some "tracked aliases" are pre-defined for the Korn Shell. What these are varies from system to system.

The Bourne Shell provides command "hashing" instead of tracked aliases, which is where the "*h*" originates.

The whence Command

Whence reports how a command will be carried out by the Korn Shell

whence -pv command

- **-v** for a verbose report
- **-p** to force a PATH search even if the command is an alias or function (AIX only option)

```
$ whence vi
/usr/bin/vi
$ whence -v vi                    executable program
vi is a tracked alias for /usr/bin/vi
$ whence -v print
print is a shell builtin
$ whence type                     so type is an alias
whence -v
$ type for
for is a reserved word
$ _
```

Figure 7-25. The whence Command AU232.3

Notes:

The whence command reports: aliases, exported aliases, keywords (Shell reserved words), builtins, functions, undefined functions (autoload functions), tracked aliases and programs.

With AIX Version 4 the *command* command is provided as both a Korn Shell builtin and as an AIX command accessible from all Shells. *command -v* and *command -V* perform similar functions to *whence* and *whence -v*. When used as an AIX command, *command* operates in a sub-shell, and thus will not report functions or aliases unless they were defined and exported by the *$ENV* file. *command -p* is similar to *whence -p*, but the former uses a default *PATH* for its search, and thus will only find the standard AIX commands.

Command Line Processing

Each command line is processed in the following way by the Korn Shell:

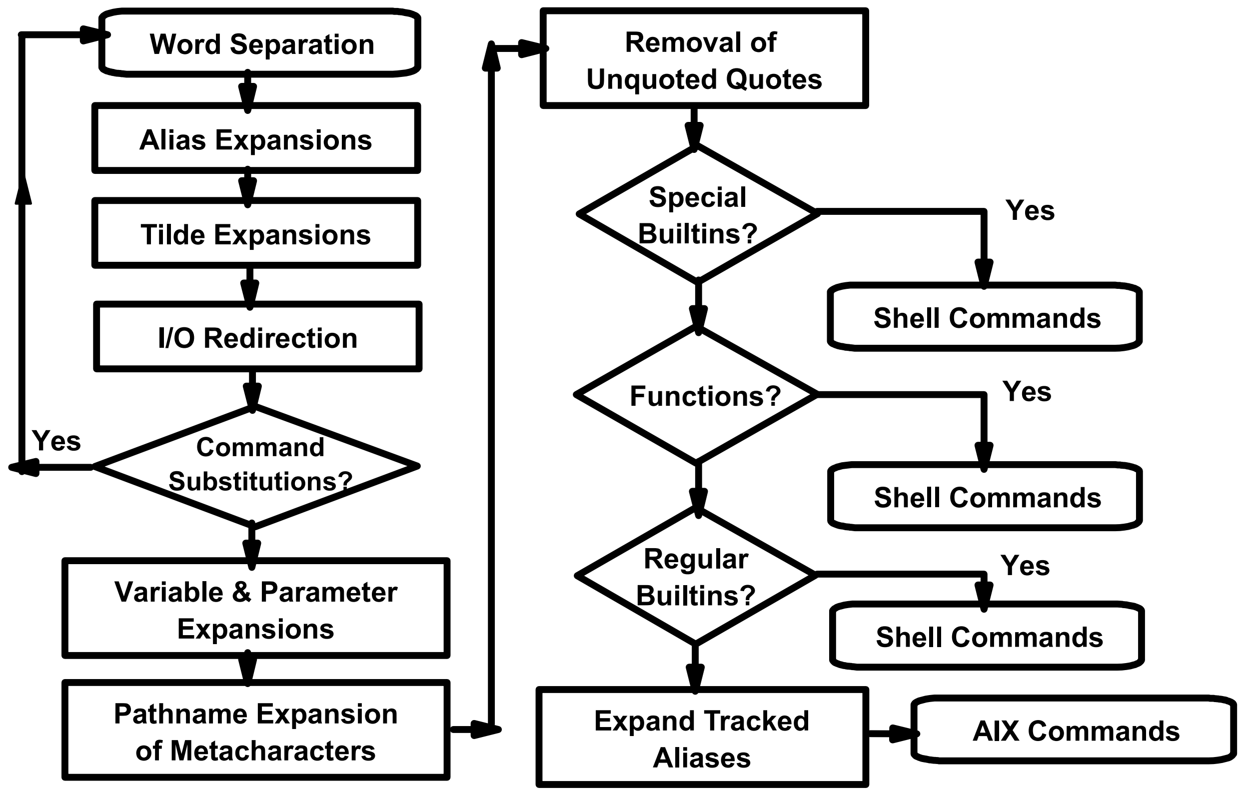


Figure 7-26. Command Line Processing AU232.3

Notes:

In the next Unit we will see what "tilde expansion" does. Other Shells process lines in a different way.

Before AIX Version 4, Shell "regular builtin commands" were handled along with "special builtin commands". "Special builtin commands" are: ".", ":", *break*, *continue*, *eval*, *exec*, *exit*, *export*, *newgrp*, *readonly*, *return*, *shift*, *times*, *trap* and *typeset*.

Checkpoint

1. How is an array defined?
2. How do we refer to array elements?
3. How could we set a variable **users**, to contain the number of users logged onto the system?
4. How would we write a function to check the readability of a file?
5. How would we write a function to print the square root of a number, with 6 decimal places?
6. How do we define local variables within a function?
7. How can we list which functions are defined?
8. Which command would allow you to load a library of functions?
9. How could we create an alias to show how many minutes have elapsed since the current shell began?

Figure 7-27. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.
2.
3.
4.
5.
6.
7.
8.
9.

Summary

- Korn Shell arrays – defining and referencing
- Command substitution
- Functions
- Typeset command
- Autoload functions
- Command aliases
- Preset aliases
- Tracked aliases
- The whence command
- Command line processing

Figure 7-28. Summary AU232.3

Notes:

Unit 8. More on Shell Variables

What This Unit Is About

This unit describes more uses for variables; replacement, changing sub-strings, length "operator" and typeset options.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Evaluate sub-strings
- Provide default or alternate values for variables
- Format strings using typeset options

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

This unit will show how to manipulate text (character) strings using Korn Shell variables:

- Variable replacements
- Variable sub-strings
- Variable lengths
- Further typeset options
- Tilde expansions

Figure 8-1. Objectives AU232.3

Notes:

Variable Replacements

Value of variables can be replaced with alternate values

`${variable:-word}`	value is **word** if **variable** is unset (use default value)
`${variable:=word}`	value is **word** if **variable** is unset and assigns word to **variable** if it is unset (assign default value)
`${variable:+word}`	value is null if **variable** is unset, else value is **word** (use alternate value)
`${variable:?word}`	if **variable** is unset, **word** is displayed on standard error and the Shell script or function terminates with a non-zero exit code (exit 1)

Figure 8-2. Variable Replacements AU232.3

Notes:

These *${ }* forms work in both the Bourne and Korn Shells.

There are no spaces between curly braces, variable, special characters or word.

If you omit *word* from the `${variable:?word}` form the Korn Shell displays the message "*ksh: variable: 0403-040 Parameter null or not set.*" by default, otherwise "*ksh: variable: word*" results.

The behavior of the `${variable:?word}` syntax in functions varies across AIX versions. In Version 3, a function terminates and returns control to the calling program. With Version 4, the Shell Script terminates completely.

The Korn Shell allows extended parameter lists, which enable the generated line to exceed the traditional Bourne Shell line length limit of 5120 characters. *Variable* can be a number — a positional parameter.

The use of the ":" allows you to decide whether a NULL variable is itself valid or not. A NULL variable has the value of the null string (usually written "" or '').

Variable Replacement Examples

Some simple examples...

- To assign the value of TERM_DEF to TERM if it is unset or null:

```
TERM_DEF=ibm3162
...
print "TERM set as ${TERM:=$TERM_DEF}"
```

- Print date and time using command substitution, or what was set earlier (do not allow null date):

```
print ${date:-$(date)}
```

- Using the alternate value "1" if variable has a value:

```
var_flag=${var:+1}
```

- To exit the script if positional parameter 3 was not given (it can be null):

```
${3?"No parameter 3? exit"}
```

Figure 8-3. Variable Replacement Examples AU232.3

Notes:

Remember that the use of a : (colon) means the value of variable may be null. So the second example only allows a string with characters in the variable `date` (but maybe not a valid date string!). In the last example, you allow positional parameter 3 to have a null string value.

Korn Shell Sub-Strings

In the Korn Shell the ${ } syntax also works with patterns:

`${variable#pattern}`	removes smallest matching left pattern from variable
`${variable##pattern}`	removes the largest matching left pattern
`${variable%pattern}`	removes the smallest right matching pattern
`${variable%%pattern}`	removes the largest matching right pattern

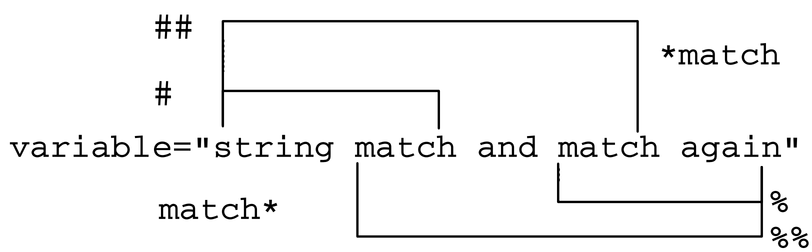


Figure 8-4. Korn Shell Sub-Strings AU232.3

Notes:

Patterns can be composed using Shell metacharacters.

Korn Shell Sub-String Examples

A bit of chopping...

```
$ variable="Now is the time"
$ print ${variable#N*i}             shortest left
s the time
$ print ${variable##N*i}            longest left
me
$ print ${variable%time}            shortest right
Now is the
$ print ${variable%%t*e}            longest right
Now is
$ _
```

Here's a function to strip out the file name from its path and print it...

```
function base
{
     print ${1##*/}          # match what?
}
```

Figure 8-5. Korn Shell Sub-String Examples AU232.3

Notes:

The function *base* says take the first parameter to the function and then applies a leftmost match from the start of the string value. The */ pattern matches up to the last / in the string or none. The result is to remove any such match leaving the last component of the pathname.

For those that are curious and have come across old scripts, the utility *expr* that was seen earlier can do similar work but it is slower and has a trickier syntax.

Korn Shell Sub-String Quiz

Now it's your turn...

1. How can I strip the ".c" extension from a C program file name held in variable "name", and print it?

2. Write a function "path" to print the pathname part of a file name.

Figure 8-6. Korn Shell Sub-String Quiz AU232.3

Notes:

Variable Lengths

A special Korn Shell variant of the **${}** syntax can be used to find the length of a variable:

- to find the number of characters in a variable...

    `${#variable}`

- the number of positional parameters is...

    `${#*}` or `${#@}`

- for the number of elements set in an array (not the highest element subscript)...

    `${#array[*]}` or `${#array[@]}`

Figure 8-7. Variable Lengths AU232.3

Notes:

You can regard the # character here as a (sort of) "length operator" when it appears inside a variable reference.

typeset Options Review

Typeset command used to

- set attributes for variables or functions
- create local variables in functions

typeset ±LN variable=value...

where ***L*** is... *i* integer, ***N*** is a fixed base
r readonly
x to export the variable

typeset ±fL function...

where ***L*** is... *x* to export the function
u for an autoload function
t to set xtrace in the function

- to set attributes, display names and values

+ to unset attributes or display just names

Figure 8-8. typeset Options Review AU232.3

Notes:

In the last Unit we saw the *typeset* command used to set attributes of variables and functions and create local variables in function definitions. There are several more options that allow variables to be formatted upon expansion by the Korn Shell. The *typeset* command is a Korn Shell builtin.

Further typeset Options

Options below allow variables to be formatted upon expansion by the Korn Shell:

typeset ±LN variable=value...

where ***L*** is...

u	convert ***value*** to uppercase when expanded
l	convert ***value*** to lowercase
L	left-justify, pad with trailing blanks to width ***N*** – if value is too big, truncate from the right
R	right-justify, adding leading blanks to width ***N*** – if wider than ***N***, truncate from the left
LZ	left-justify to width ***N*** and strip leading zeros
RZ	right-justify to width ***N***, adding lead zeros if the first character is a digit

Figure 8-9. Further typeset Options AU232.3

Notes:

For some systems there are multi-byte versions of the Korn Shell (using national language support). There the width refers to the number of columns rather than the number of characters. The default width is the width of first assignment.

Option *Z* is identical to *RZ*.

typeset Examples

Here are the different types in action...

```
$ typeset -u var=upper
$ print $var
UPPER
$ typeset -l var=LOWER      # lower case ell
$ print $var
lower
$ typeset -L6 text=SIDE
$ print "${text}="
SIDE  =


$ typeset -R6 text
$ print "=$text"
=  SIDE


$ typeset -LZ4 num=000.1234567
$ print ${num}
.123
$ typeset -RZ5 num=1234567
$ print $num
34567
```

Figure 8-10. typeset Examples AU232.3

Notes:

Tilde Expansions

Following alias expansion the Korn Shell checks for a leading unquoted ~ character to see if it is:

~	tilde by itself is replaced by $HOME
~+	is replaced by $PWD
~-	is replaced by $OLDPWD
~user_name	is expanded into the $HOME value for the ***user_name*** given
~other_text	will be left alone

Examples...

```
cd ~           ≡    cd $HOME
lastdir=~-     ≡    lastdir=$OLDPWD
johns=~john    ≡    johns=/home/john
```

Figure 8-11. Tilde Expansions AU232.3

Notes:

The use of tilde is not often seen now though you may see *~userid*.

Checkpoint

1. What happens when the variable **TMOUT** is set and you enter the following? **TMOUT=${TMOUT:-60}**

2. What would your prompt say if you were in your **bin** directory and you entered this: **PS1='${PWD#$HOME/} $'**.

3. How could you find out the number of characters in the variable HOME?

Figure 8-12. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.

2.

3.

Summary

- Variable replacements
  - *for unassigned/null strings*
- Variable sub-strings
  - *simple pattern matches*
- Variable lengths
  - *the # "operator"*
- Further typeset options
  - *justification and padding*
- Tilde expansions
  - *shortcuts*

Figure 8-13. Summary AU232.3

Notes:

Unit 9. Regular Expressions and Text Selection Utilities

What This Unit Is About

This unit describes regular expressions, and some UNIX text selection utilities.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Understand and use regular expressions
- Use grep, cut, and other text selection and manipulation tools

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

This unit will show how to select and manipulate text (character) strings using:

- Regular expressions
- The *grep* command
- The *tr* command
- The *cut* command
- The *paste* command

Figure 9-1. Objectives AU232.3

Notes:

Sample Data File

To manipulate data, we need to know its format.
The data file we will use in this unit has the following structure:

```
Lastname,<SPC>Firstname<TAB>nnn-mmmm

$ cat  phone.list

Terrell, Terry          617-7989
Franklin, Francis       704-3876
Patterson, Pat          614-6122
Robinson, Robin         411-3745
Christopher, Chris      305-5981
Martin, Marty           814-5587
Llewellyn, Lynn         316-6221
Jansen, Jan             903-3333
Llewellyn, Lee          817-8823
$ _
```

Figure 9-2. Sample Data File AU232.3

Notes:

The *phone.list* file will be used in examples on following pages. There is a single space character after the comma following the *Lastname*.

Regular Expressions

Powerful feature available in many programs

Used to **select** text

- vi, ex, emacs, grep/egrep, sed, awk, perl

What are they?

- An expression representing a pattern of characters
- Contain a sequence of characters/metacharacters

Figure 9-3. Regular Expressions AU232.3

Notes:

You will find the regular expression feature is part of many programs such as in editors and in pattern matching utilities (we see later in this unit). The principles and uses of regular expressions (often abbreviated to RE) appear in many places in AIX and UNIX systems. Once you have grasped the essential techniques you will find that they can be used over and over again.

A RE is just that — an expression that represents a pattern of text. Such an expression can contain simple sequences of characters or more complex sequences that use special characters (*metacharacters*) to describe more complex patterns of text.

Regular Expression Metacharacters

Pattern	Meaning (matches)
aphanumeric character	The character itself (not really a metacharacter)
. (period)	Any single character
[AZ]	One of A or Z
[^AZ]	Any character not A or Z
[A-Z]	Any character in range A to Z
[-AZ]	One of -, A or Z
[0-9]	Any digit 0 to 9

Figure 9-4. Regular Expression Metacharacters AU232.3

Notes:

The Shell interprets metacharacters differently from AIX operating system commands.

When used in a regular expression, the "*" says match zero or more of the previous character. A dot (.) means any single character so to match one or more occurrences of any character use ".*" as the regular expression.

You may also use the *[:class:]* named classes from POSIX and the Shell. For example, for any digit you can use [[:digit:]].

Extending the Pattern

Two ways:

- Anchors
- Multipliers

Anchors are

^	Matches beginning of line
$	Matches end of line

Multipliers apply to patterns. They are

*	zero or more occurences of previous pattern
?	zero or one occurence of previous pattern
+	one or more occurences of previous pattern
{m,n}	at least m and no more than n occurences of previous pattern ("quoted braces")

Figure 9-5. Extending the Pattern AU232.3

Notes:

Two other metacharacters used within regular expressions specify position in the line of the character(s). The caret "^"specifies the beginning of the line; "^*t*" says any line starting with a *t*. The *$* specifies the end of the line; *7$* says match any lines that end with a *7*.

You can also get wildcard effects by extending the pattern with *multipliers*. The most common are the use of * and quoted braces. The next page deals with braces.

You find the other multipliers in programs that have an extended RE syntax such as egrep, awk and perl.

Quoted Braces

To specify the number of consecutive occurences

Syntax 1: `regular_expression\{min, max\}`

To look for two, three or four occurrences of any combination of the characters 3, 4 and 5 consecutively

```
grep '[345]\{2,4\}' phone.list
```

Syntax 2: `regular_expression\{exact\}`

To look for any lines which have two consecutive "r" characters

```
grep 'r\{2\}' phone.list
```

Syntax 3: `regular_expression\{min,\}`

To look for any lines with at least two consecutive "r" characters preceded by an "e"

```
grep 'er\{2,\}' phone.list
```

Figure 9-6. Quoted Braces AU232.3

Notes:

We shall see more on the *grep* command later in this Unit.

Quoted braces offer a more specific wild-card than the asterisk.

`\{min,max\ .`

This will search for lines which contain between the minimum and maximum number of the previous RE in a sequence.

- *`a\{4,5\}`* says look for 4 or 5 repeats of the character "a" in sequence within a line.

- *`[367]\{2,6\}`* says look for 2 to 6 occurrences of any combination of *"3"*, *"6"* or *"7"* in sequence.

- *`.\{6,7\}`* says look for 6 to 7 occurrences of any character.

`\{min\ .`

Here an exact number of repeats are specified, as the maximum number is omitted.

\{min,\ .

Here the minimum number is set, there is no maximum number, it is equivalent to looking for at least "*min*" repeats.

The single regular expression preceding quoted braces can be regular characters or a pattern of metacharacters. Further characters or patterns will be matched in the usual way:

- *ab\{4,5\}* says look for an "*a*" followed by a "*b*" repeated 4 or 5 times.

Quoted Parentheses

To capture the result of a pattern

Syntax: `\(regular expression\)`

- Stores the character(s) that match the regular expression (within parentheses) in a register
- Nine registers are available; characters which match the first quoted parentheses are stored in register one, those that match the second quoted parentheses in register two, etc.
- To reference a register use a backslash followed by a register number:

  \1 to \9

For example, to list any lines in "phone.list" where there are two identical characters together...

```
grep  '\(.\)\1'  phone.list
```

Figure 9-7. Quoted Parentheses AU232.3

Notes:

Quoted parentheses store character(s) from the input line to use as patterns to match against other character(s) from the input line. If you want to know whether the first two characters on the line are the same, but you don't know what the first character is, quoted parentheses allow the first character to be read into a buffer (or *register*) and then the second character to be compared with the buffer's contents.

"`\(.\)`"matches any single character and puts it into register "`\1`". So the pattern "`\(.\)\1`" identifies a two-character sequence where both characters are the same.

Regular Expressions – Quiz

Using the "phone.list" file, what RE gives:

1. People with five-letter surnames?
2. People with first names of at least four characters?
3. All entries where the number before the dash is the same as that after the dash e.g. 3-3456?
4. People whose surnames begin with A, B or C?

Figure 9-8. Regular Expressions — Quiz AU232.3

Notes:

Regular expressions may be quoted so that the Shell does not interpret the metacharacters.

grep Command

- Search file(s) or standard input for lines containing a match for a specific pattern

  `grep [options] pattern [ file1 file2 . . . ]`

- Valid grep metacharacters: `. * ^ $ [-]`

.	any single character
*	zero or more occurrences of the preceding character
^a	any line that begins with "a"
z$	any line that ends with "z"
[a-f]	any ONE of the characters in the stated range

- Valid options:

-c	print only a count of matching lines
-i	ignore the case of letters when making comparisons
-l	print only the names of the files with matching lines
-n	number the matching lines
-s	works silently, displays only error messages
-v	print lines that do NOT match
-w	do a whole word search

Figure 9-9. grep command AU232.3

Notes:

The **grep** command (**g**/**re**/**p**) searches for the specified pattern from STDIN and displays to STDOUT. The search can be for simple strings or regular expressions.

There are other greps in the family :

fgrep only fixed string allowed

egrep allows multiple (either | or) patterns

Historically, early greps did not allow quoted (\) parentheses or braces. Only egrep understood the extended syntax.

grep Examples

```
1.$ grep -i "tech support" phone.list

2.$ grep bob /etc/passwd

3.$ ps -ef | grep tracy

4.$ ls -l | grep '^d'

5.$ grep -n '.*' /etc/passwd > \
  > passwd.file.numbered.lines

6.$ egrep '^b(i|o)' /etc/passwd
```

Figure 9-10. grep Examples AU232.3

Notes:

1. In a file called **phone.list** in the current directory, search for the string 'tech support' and display to STDOUT. The *-i* will allow grep to find the string whether the letters are uppercase or lowercase. This command will NOT find "technical support" or "support line."
2. This will search the **/etc/passwd** file and find "bob" — but will also find "billybob."
3. Find any processes that were started by the user named tracy — but will also find any command with the same string, i.e., `mail tracy <letter`.
4. Display only directories in the current directory.
5. Creates a new file that includes all the /etc/passwd information and numbers the lines.
6. Find a user that begins with either "bi" or "bo" and display to STDOUT. This will find billy OR bob, but also billybob.

tr For Translations

The **tr** command translates one set of characters into another:

```
tr LISTIN LISTOUT < in_file > out_file

          - or -

tr -d LISTIN < in_file > out_file
```

- characters in **LISTIN** are replaced by the corresponding ones in **LISTOUT**
- if **LISTOUT** contains fewer characters than **LISTIN,** ignores extra ones from LISTIN
- if **LISTOUT** contains more characters than **LISTIN,** ignores extra ones from LISTOUT
- with **-d**, characters in **LISTIN** are deleted
- only works with STDIN and STDOUT

Figure 9-11. tr For Translations AU232.3

Notes:

There are two versions of the *tr* command supplied by AIX: the AIX version */usr/bin/tr* (explained above), and a BSD version */usr/ucb/tr* which uses slightly different syntax. The AIX flavor */usr/bin/tr*, will be the one obtained by a default *PATH*. The BSD version pads a short *LISTOUT* to the same length as *LISTIN* using the last character of *LISTOUT*.

Note that *tr* does not allow filename arguments.

tr Examples

Some simple translations...

```
$ print $HOME | tr "/" "-"
-home-team01
$ print "{ { [ ... ] } }" | tr "{}" "()"
( ( [ ... ] ) )
$ print "Lower to upper" | tr "[a-z]" "[A-Z]"
LOWER TO UPPER
$ print "TOP DOWN" | tr '[:upper:]' '[:lower:]'
top down
$ print "vowels and consonants" | tr -d aeiou
vwls nd cnsnnts
$ tr -d '\015' <dos_txt_file >aix_txt_file
$ _
```

Figure 9-12. tr Examples AU232.3

Notes:

The cut Command

Cut extracts fields or columns from text input

```
cut -dS -s -flist [ file ]

        or

cut -cLIST [ file ]
```

`-dS`	where S is the character to take as a delimiter
`-s`	with -dS suppresses lines that do not contain delimiters
`-fLIST`	specifies a **LIST** of fields to cut out and keep
`-cLIST`	is a **LIST** of columns to cut (character positions)

LIST - specifies field or column numbers
- may contain comma separated values (m,n) or a range (m-n)

Figure 9-13. The cut Command AU232.3

Notes:

The *cut* command is provided by the AIX operating system. Standard input can be used in place of a named file. The default delimiter is *TAB*.

cut Examples

Field numbering starts at 1

```
$ cut  -d:  -f1,3  /etc/passwd  |  head  -3
root:0
daemon:1
bin:2
$ cat  /etc/passwd  |  cut  -d'*'  -s  -f1
guest:
$ df  |  cut  -c6-10  |  tail  +2
hd4
hd2
hd3
hd1
$ text="A tasty dish to set before the King!"
$ echo  $text  |  cut  -c-8,32-
A tasty King!
$ _
```

Figure 9-14. cut Examples AU232.3

Notes:

A "-" by itself at the start of a range means from the first column or field; at the end of a range it means to the end of the line.

The paste Command

As name suggests, sticks (merges) things together

Commonly used to create or format a data stream

Default output is

line from file1 <TAB> line from file2

Separator(s) may be changed on command line

Options:

-d [dlist] the delimiter between files (may be a list)

-s make the output a single line of all lines of each file

Figure 9-15. The paste Command AU232.3

Notes:

The *paste* command is complementary to the *cut* command. It assembles files into a single multicolumn file — each column formed from a named file. The *dlist* characters are inserted as delimiting characters — either one character that is used to separate all columns, or a list that will be used sequentially — one character for each column join. You may use the *print* special characters to represent a newline, *TAB*, etc.:

```
paste -dS file1 file2 ...  > joined_file
```

paste Examples

Print a 3 column listing of .ksh files:

```
ls *.ksh | paste - - -
```

Format a listing in 3 columns using <TAB> <TAB> <NEWLINE> as delimiters

```
ls *.ksh | paste -d"\t\t\n" -s -
```

Figure 9-16. paste Examples AU232.3

Notes:

Notice how you can use "-" to represent STDIN and how it may be used more than once (giving three files).

Checkpoint

1. What regular expression can you use to select surnames?
2. What regular expression can you use to select text with repeated characters in the surname?
3. What command can you use to select lines in phone.list with four character first names?
4. How could you count the number of processes whose PIDs are in the range 1000-9999?
5. How would you convert spaces to a tab in phone.list?
6. What would this next command accomplish? **cut -d: -f1,3,4 /etc/passwd**
7. Using the **paste** command, output the /etc/passwd file so that each line of information is separated by a tab and so that the fifth, sixth and seventh fields are on a separate line from the others. (Hint: make each field a line.)

Figure 9-17. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.
2.
3.
4.
5.
6.
7.

Summary

- Understand Regular Expressions
- Using the *grep* command to select text
- Using the *tr* command to translate characters
- Using the *cut* command to select text fields
- Using the *paste* command to merge data streams

Figure 9-18. Summary AU232.3

Notes:

Unit 10. Utilities for Personal Productivity

What This Unit Is About

This unit describes how the sed utility manipulates data. Finally, the unit briefly looks at three utilities to help improve productivity — tar, at and crontab.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Use sed to edit file contents
- Understand sed advanced features
- Make use of tar archives
- Be able to schedule scripts for execution at a later date.

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

This unit will introduce utilities that can improve your personal productivity – sed, tar, at, crontab

- use the stream edit utility – *sed*
- use the archive utility – *tar*
- manipulate when your work gets done – *at* and *crontab*

Figure 10-1. Objectives AU232.3

Notes:

sed

There are several ways of running sed:

- **sed** 'edit-instructions' filename
- command | **sed** 'edit-instructions'
- **sed** -f command.file filename

Note: The input file is not changed or overwritten by **sed**!

Figure 10-2. sed AU232.3

Notes:

The *sed* command can be invoked in a number of ways. The *sed* command takes its input from standard input unless a *filename* is specified on the command line; it writes its output to standard output. Thus *sed* is a filter and can be used within a pipe.

The output of *sed* can be redirected to a file; a word of warning, never try to redirect the output of *sed* back to the original input file as this is not supported by the Shell and due to the order in which the Shell processes the command line, you will end up losing the original contents of the input file.

The edit instruction(s) can be provided on the command line, or in an ASCII file if *sed* is invoked with the *-f* option.

Line Selection

The **sed** instructions operate on all lines of the input, unless you specify a ***SELECTION*** of lines:

```
sed 'SELECTION edit-instructions'
```

SELECTION can be

- a single line number
  - `1` = line 1 of the input
  - `$` = the last line of the input

- a range of line numbers
  - `5,$` = from line 5 to the end of the input

- a regular expression to select lines matching a pattern
  - /string/ = selects all lines containing "string"

- a range using regular expressions
  - /^on/,/off$/ = from the first line beginning with "on" to the first ending in "off"

Figure 10-3. Line Selection AU232.3

Notes:

Regular expressions used for line selection must be delimited by the '/' character.

The Substitute Instruction

This instruction changes data

Syntax: `s/old string/new string/g`

Some examples

1. To replace the first occurrence of "Smith" on each line with "Smythe"

```
sed 's/Smith/Smythe/' phone.list
```

2. To replace all occurrences of "Smith" with "Smythe" using a different delimiter

```
sed 's!Smith!Smythe!g' phone.list
```

3. To precede each phone number with "Tel:"

```
sed '/[0-9]\{3\}-[0-9]\{4\}/s//Tel: &/g' \
phone.list
```

Figure 10-4. The Substitute Instruction AU232.3

Notes:

The data file phone.list is same as that in Unit 9.

A "`\`" can be used to escape any special meanings of characters in your strings or addresses - i.e. "`\.`" is a dot, and "`\ `" a literal ampersand.

Rather than a "*g*", you can specify that the nth occurrence is to be replaced by putting a number "*n*" in place of the "*g*".

A blank "*old string*" section is expanded into whatever matched the line *SELECTION* pattern.

A blank "*new string*" section results in the "*old string*" being deleted. There is a better way of doing this, as we shall see later on.

The "&" is used to redisplay what was previously matched.

Substitutions - Quiz

1. Convert the "phone.list" into just a name list, i.e. get rid of the phone numbers

output:
```
Terrell, Terry
Franklin, Francis
Patterson, Pat
 ...,    ...
```

```
sed  's/______________________//'  phone.list
```

2. Convert the "phone.list" file to a first-name and number list

output:
```
Terry        617-7989
Francis      704-3876
Pat          614-6122
...             ...
```

```
sed  's/______________________//'  phone.list
```

Figure 10-5. Substitutions - Quiz AU232.3

Notes:

sed with Quoted Parentheses

- Repeating the first character

```
$ print "1234" | sed 's/^\(.\)/\1\1/'
11234
$ _
```

any single character to register 1

register 1 is repeated

- Stripping out all but the first and last characters

```
$ print "1234"|sed 's/^\(.\).*\(.\)$/\1\2/'
14
$ _
```

character to register 1

character to register 2

register 1 and 2

Now it's your turn...

Working on the "phone.list" file, abbreviate everyone's first name to an initial and a period (use register 1 to store each initial)

```
sed  's/__________/__________/'  phone.list
```

Figure 10-6. sed with Quoted Parentheses AU232.3

Notes:

Summary for Substitutions

- without a "**g**", **sed** only substitutes the first match

```
$ print xxx | sed ' s/x/y/'
yxx
$ print xxx | sed 's/x/y/g'
yyy
$ _
```

- other delimiters can be used when "/" makes life difficult
  – e.g. converting an AIX to a DOS pathname

```
$ pwd | sed 's/\//\\/g'
\home\kim\desktop
$ pwd | sed 's;/;\\;g'
\home\kim\desktop
$ _
```

Figure 10-7. Summary for Substitutions AU232.3

Notes:

If you have multiple lines, each line of the *text*, apart from the last one, must be followed by the "\" character. This "\" escapes the return character.

Delete and Print

This command removes text

Syntax: SELECTION**d**

- To delete all lines in the output stream

```
$ sed  d  phone.list
```

- Delete from line 5 to the end of the file

```
$ sed  '5,$d'  phone.list
```

By default **sed** writes out every line it reads in

– makes print instruction "**p**" by itself redundant:

```
$ cat in.file
line 1
line 2
$ sed p in.file
line 1
line 1
line 2
line 2
$ _
```

Figure 10-8. Delete and Print AU232.3

Notes:

- Delete the last line of output

```
$ sed  '$d'  phone.list'
```

- To remove any blank lines

```
$ sed '/^$/d'  phone.list
```

Print is of more use with the **-n** option — to suppress normal printing of input lines, and only print a ***SELECTION***

```
$ sed -n in.file   #select all lines
line 1
line 2
$ sed -n '/2/p' in.file#select lines with a "2"
line 2
$ _
```

Append, Insert and Change

These instructions add or modify text

Syntax: SELECTION**x**
text

Where **x** is

i inserts **text** before a single selected line

a appends **text** after a matched line

c changes a range of matched lines into **text**.
SELECTION can be a single line or a range but only one copy of **text** is printed in its place

Figure 10-9. Append, Insert and Change AU232.3

Notes:

SELECTIONS are:

- line number
- regular expression
- range of lines

Example ...

```
$ sed '1a\
> Add after line 1 of the input'  in.file
Line 1
Add after line 1 of the input
Line 2
$ _
```

Command Files

- A **sed** command file consists of one or more **sed** instructions on separate lines
- Command files are useful in many situations:
  - storing multiple instructions
  - storing a long complex command
  - for commands which may need to be modified and reused
- Use the "**-f**" option to use a command file

Example...

```
$ cat sedscript.sed
  s/ GA/, Georgia/
  s/ FL/, Florida/
  s/ IL/, Illinois/
  s/ TX/, Texas/
  s/ MD/, Maryland/
  s/ DC/, District of Columbia/
$ sed -f sedscript.sed addrs.file > new.addrs.file
$ _
```

Figure 10-10. Command Files AU232.3

Notes:

It is sometimes useful to add an extension to a script to denote the type of its contents. You have seen the use of .ksh for script files; here we adhere to the same convention and use .sed for our sed scripts.

A Practical Example

Converting a "BookMaster" script to a "wysiwyg" file

```
:ul.
:li.An unordered list starts with ":ul.".
:li.Each list item is tagged with ":li." - it
appears as an indented bullet point.
:li.The end of the list is marked by ":eul."
:eul.
```

Strategy:

1. Remove lines which contain just ":ul." or ":eul."
2. For lines that start with ":li.", substitute the ":li." with a dash followed by five spaces

```
$ cat  bkm.wysi.sed
/^:e*ul\.$/d
s/^:li\./-     /
$ sed  -f bkm.wysi.sed  bookmaster.file > wysi.file
$ cat wysi.file
-     An unordered list starts with ":ul.".
-     Each list item is tagged with ":li," - it
appears as an indented bullet point.
-     The end of the list is marked by ":eul."
```

Figure 10-11. A Practical Example AU232.3

Notes:

Multiple Editing Instructions

- Multiple instructions can be applied to each line
- Each instruction must be on a separate line

Example 1...

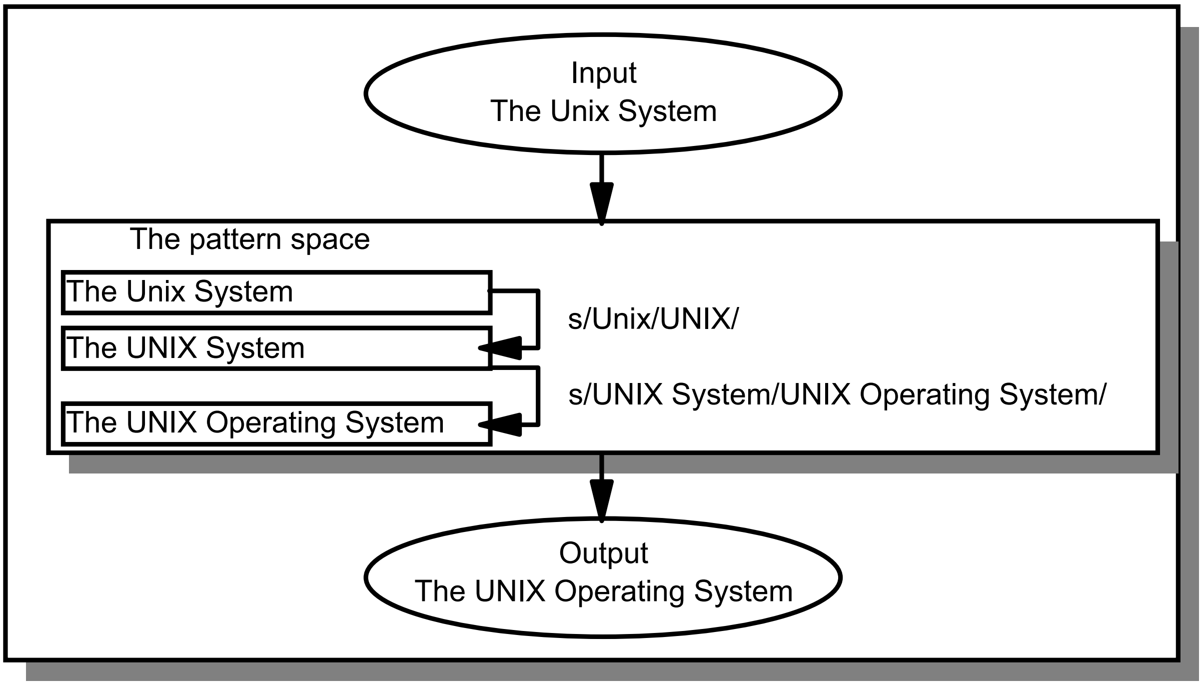
```
$ sed  '/[1-4]-/s/$/ (Bldng 1) /
>      /[5-9]-/s/$/ (Bldng 2) /'  phone.list

Terrell, Terry       617-7989  (Bldng 2)
Franklin, Francis    704-3876  (Bldng 1)
```

Figure 10-12. Multiple Editing Instructions AU232.3

Notes:

Internal Operation



- sed applies all editing instructions to a line before it moves on to the next line
- it holds each input line in a "pattern space" or temporary buffer while editing instructions are applied in sequence

Figure 10-13. Internal Operation AU232.3

Notes:

Internal Operation – Example

Example of sed command/instructions

```
$ print "The Unix System" | sed  's/Unix/UNIX/
>        s/UNIX System/UNIX Operating System/'
The UNIX Operating System
$ _
```

Figure 10-14. Internal Operation - Example AU232.3

Notes:

This is the *sed* script that corresponds to the previous page.

Grouping Instructions

Braces "{" "}" are used for two purposes:

- one *SELECTION* inside another (*nest*)
- to apply multiple instructions to the same *SELECTION* range (*group*)

Example...

SELECTION range

```
/\.ol/,/\.eol/{
        /^$/d
        }
```

instructions for the SELECTION range

- The instruction "/^$/d" (delete blank lines) will be applied to a range of lines between one that contains an ".ol" and up to the first containing an ".eol"
- The special meaning of the dot preceding "ol" and "eol" is escaped by the use of a backslash

Figure 10-15. Grouping Instructions AU232.3

Notes:

sed Advanced Topics

There are two other areas in *sed* that can be useful
- multiple input lines for the pattern space
- use of the hold space (temporary area)

There are three instructions for multiline input
- N Read next line
- P Print line
- D Delete line

Notice they are in UPPER CASE

Figure 10-16. sed Advanced Topics AU232.3

Notes:

You will recall that in the internal operation description, *sed* used a pattern space to work on the input line. In this part of the unit, you will see that *sed* allows multiple lines in that pattern space and also that you can copy (save) the current "buffer" for later use.

Implicit in the *sed* instructions seen so far is that at the end of each script, the pattern space is cleared for new input. Indeed, there is an instruction ***n*** that forces this to happen. The instructions that follow allow the pattern space to contain multiple lines.

Multiple Input Lines - N Instruction

The **N** instruction

- does NOT clear pattern space
- inserts an (embedded) newline ("\n") into the pattern space
- reads a line and appends to the pattern space

Similar to **n** instruction

- BUT n clears pattern first

An embedded newline ("\n") can be matched explicitly

^ and $ refer to the FIRST and LAST character respectively of the pattern space

Figure 10-17. Multiple Input Lines - N Instruction AU232.3

Notes:

The *N* instruction allows additional lines to be added to the pattern space. Each use of *N* means one newline and one input line added to the pattern space. Remember that the *n* instruction does clear the pattern space.

Since there are multiple lines (and therefore newline characters) then the meaning of ^ and $ change.

The P and D Instructions

These also do not clear the pattern space

P prints the pattern space up to the first embedded newline

D deletes the text up to the first embedded newline
- no new input (contrast to the d instruction)
- processing of pattern space continues from top of script

Figure 10-18. The P and D Instructions AU232.3

Notes:

Again, these instructions operate on the text up to the first newline in the pattern space. Using these instructions means that text is printed or deleted from the existing text in the pattern space.

Multiline Pattern Spaces – Example

```
$ sed  '/Adams/{
>       N
>       s/.-[0-9]*/censored/g
>       }'  phone.list
Smith, Terry                 7-7989
Adams, Fran                  censored
StClair, Pat                 censored
Brown, Robin                 1-3745
Stair, Chris                 5-5972
Benson, Sam                  4-5587
Harris, Ford                 6-6221
Phiri, Ray                   3-3333
Llewellyn, Nia               7-8823
$ _
```

Figure 10-19. Multiline Pattern Spaces - Example AU232.3

Notes:

The example above shows how two consecutive lines can have the same sed instructions applied by appending to the input buffer with *N*. In the example, you know that it is the line with **Adams** that is the signal for edits to be applied.

The Hold Space

This is a set-aside or copy buffer
Hold space cannot be directly changed (edited)
It is a temporary storage area

There are three instructions available

- h or H copy or append contents of pattern to hold space (HOLD)
- g or G copy or append contents of hold to pattern space (GET)
- x swap pattern and hold space (EXCHANGE)

An example

```
$ print "1\n2" | sed '/1/{
>                    h       hold line matching 1
>                    d       delete pattern space
>                    }
>                  /2/G'    2 line + hold space
2                              print pattern space
1
$ _
```

Figure 10-20. The Hold Space AU232.3

Notes:

A better name might have been "temp" space. The HOLD and GET instructions that operate on this additional buffer/storage area have two forms; lower and upper case. The lower case form clears the Hold Space before copying the pattern space. The upper case form appends the pattern space to any existing Hold Space data.

The tar Utility

This is an archive/backup command

Historically used tape but now any device

- default to /dev/rmt0

Syntax: `tar options pathname(s)`

Figure 10-21. The tar Utility AU232.3

Notes:

The *tar* utility is very useful for temporary archives and backups. It was originally written to output to a tape device but is now used for virtually any storage device. For AIX the normal default is */dev/rmt0* but as you will see this can be changed by a command line option.

tar Options

Options are of two types
- required
- optional

Should be specified using a leading hyphen

Required options are one of
- c - create an archive
- x - extract file(s) from archive
- t - list (tell) what is in archive

Other (optional) options are
- f - used to specify other than default device
- v - verbose (usually with t or x)
- m - restore/keep modification times

Figure 10-22. tar Options AU232.3

Notes:

tar options are in two groups — required and optional. The original utility did not conform to the normal syntax for parameters and options. Some old scripts using tar may be seen without a leading hyphen (-) before the options. Normal modern practice is to use the correct option syntax.

tar options are many and use of the AIX documentation and/or the man pages may be helpful. As the syntax suggests, there must be a *required* option present. The most common "optional" options are *-f* and *-v.* For example, to read an archive from the default device:

```
$ tar -tv
-rw-r--r-- phil/office     527 2000-02-01 17:13:09 getopts.ksh
-rwxr-xr-x phil/office      50 2000-07-06 13:25:26 group1.ksh
-rwxr-xr-x phil/office      55 2000-07-06 13:25:26 group2.ksh
-rwxr-xr-x phil/office     195 2000-07-06 13:25:26 if-then-elif.ksh
-rwxr-xr-x phil/office     123 2000-07-06 13:25:26 if-then-else.ksh
$ _
```

Notice that using *v* gives the equivalent of a long listing of a directory.

Typically the *-f* option is used to specify a tar file, often called a *tarfile*. For example

```
$ tar -cf au23.tar examples
$ _
```

creates a tarball of the directory examples.

tar Pathnames

tar takes a pathname as one of its parameters

Full pathnames mean that restores (extracts) will be to original directory

Relative pathnames mean that restores may be to any part of filesystem

tar may be used to do recursive copies of data from one directory to another

```
$ cd fromdir; tar cf - . | (cd todir;\
>tar xf -)
```

Figure 10-23. tar Pathnames AU232.3

Notes:

Since a pathname is involved it can be either a full or relative path. With tar, a full pathname will mean that files/directories extracted will be to the original path.

For that reason, relative pathnames are usually preferred for backups or archives. Choose carefully if you think that full paths are necessary.

Working in Absentia

You can submit jobs for execution later

AIX provides two useful utilities
- at
- crontab

Access to these facilities is controlled by the system administrator

Figure 10-24. Working in Absentia AU232.3

Notes:

Suppose you want to process some material but can wait (e.g. overnight). The AIX utilities *at* and *crontab* (with the *cron* daemon) will help you.

It is possible that a tightly controlled system will not allow you to use these facilities until expressly enabled by the system administrator.

The at command

at submits a set of commands (a job) for later execution

Syntax: `at [-r|-l] time`

Commands are read from stdin

`time` can be specified as absolute or relative

- the time may include a date

Options include

- `-l` list your at jobs
- `-r` remove your *at* job(s)

at uses mail to send the stdin and stderr output (unless redirected)

System administrator determines who may use *at*

Figure 10-25. The at Command AU232.3

Notes:

The set of commands (or script) submitted by *at* becomes an *at job*. This is not the same as a job in the Korn Shell.

The *time* syntax can be absolute as in 2200 or relative to some other time. The time specification can also include a date if required. The important point is that the "job" only executes once.

Note that the script (the set of commands) are copied to a spool area. This means that even if the script is subsequently edited, the changes are not made to the submitted script.

at Usage and Examples

Here are some examples (commands excluded)

at 2100
at 10pm
at 4am
at 9am tomorrow
at 10:30 Jul 3
at now + 2 hours
at now + 2 days
at now + 1 year

Figure 10-26. at Usage and Examples AU232.3

Notes:

There are many different formats that you can use to specify the time. The use of *now* and *tomorrow* are useful.

The crontab Command

This command is like *at* but for regular "jobs"

Syntax: `crontab [-e | -l | -r] [job-file]`

The commands executed are in job-file (or from stdin)
The options allow you to edit, list or remove your crontab file

System administrator determines who may use *cron*

cron will mail the output of the command to crontab owner

Figure 10-27. The crontab Command AU232.3

Notes:

crontab allows you to specify both date/time and frequency of a particular "job". The crontab file has a particular format (you will see this next). To create an entry in your crontab, use

```
$ crontab job-file
```

The system daemon *cron* examines crontab files in the spool area every minute and loads any changes. Using crontab to edit your crontab entries is the best way to ensure that cron is informed of any updates.

Like the at command, a system administrator controls which users have access to crontab facilities.

crontab File Format

cron needs crontab files in a particular format
Each line has time(s)/date(s) and the command to run

Format of each line is a set of fields
- minute (0-59)
- hour (0-23)
- day (1-31)
- month (1-12)
- day of week (0-6, 0 = Sunday)

Each of the first five fields may be
- a number
- a comma separated number list (1,3,4,13)
- a range (4-9)
- an asterisk (*)

Sixth field contains the command(s) executed (a % means a newline)

Figure 10-28. crontab File Format AU232.3

Notes:

Each of the six crontab fields are separated by whitespace, usually a space or tab character.

crontab Examples

Here are some possible crontab file entries/lines

```
# Run command at 0900 and 1200 Mon-Fri
15 9,12 * * 1-5 /home/sa/games_off
# Do some backups at 0200 Tue-Sat
0 2 * * 2,3,4,5,6 /home/sa/backup daily

# What does this one do?
13 5 * * 0 find $HOME -name ,\* -exec rm -f {} \;
```

Figure 10-29. crontab Examples AU232.3

Notes:

Checkpoint

1. Write a command line script that displays a **ps -ef** with your username as the owner of init.

2. How can I make phone.list appear double spaced?

3. How could you backup files in your HOME directory to tape?

4. Suppose you needed to run a script but don't need the results until the next day, what command might you use?

5. As an administrator you decide that every weekend you will check the disk usage of users' directories in /home. Which method is preferable — using *at* or *cron/crontab*?

6. Construct a suitable crontab entry for the previous question.

Figure 10-30. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.

2.

3.

4.

5.

6.

Summary

- Use of *sed* to automate repetitive editing tasks
- Archiving using *tar*
- Batching commands for later execution:
  - One off using *at*
  - Regular or repeated using *crontab*

Figure 10-31. Summary AU232.3

Notes:

Unit 11. The AWK Program

What This Unit Is About

This unit describes how to use and program in *awk*.

What You Should Be Able to Do

You should be able to:

- Use awk to generate formatted output from input files
- Create and use a simple awk script
- Be aware of the more advanced and powerful features of awk programming that are available

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on exercises

Objectives

This unit will show you how to use the awk utility by looking at:

- Regular expressions in awk
- Basic awk programming
- BEGIN and END processing
- Flow control – if, while, and for
- Leaving loops – continue, next and exit
- Awk arrays
- Better printing
- Awk functions

Figure 11-1. Objectives AU232.3

Notes:

What Is Awk?

- **Awk** is a programming language used to manipulate text
- **Awk** sees data as words (**fields**) in a line (**record**)
- An **awk** command consists of a ***pattern*** and an **action** comprising one or more statements

```
awk '/pattern/ { action }' file ...
```

- **Awk** tests every **record** in the specified ***file(s)*** for a ***pattern*** match. If a match is found, the specified **action** is performed
- **Awk** can act as a filter in a pipeline or take input from the keyboard (standard input) if no ***file(s)*** are specified

Figure 11-2. What is Awk? AU232.3

Notes:

awk is sometimes called a report generator tool.

awk program text may be thought of as a data driven program.

Sample Data – awk

```
Lastname,<SPC>Firstname<TAB>nnn-mmmm

$ cat phone.list

Terrell, Terry                      617-7989
Franklin, Francis                   704-3876
Patterson, Pat                      614-6122
Robinson, Robin                     411-3745
Christopher, Chris                  305-5981
Martin, Marty                       814-5587
Llewellyn, Lynn                     316-6221
Jansen, Jan                         903-3333
Llewellyn, Lee                      817-8823
$ _
```

The same file as in the RE and sed units

Figure 11-3. Sample Data - awk AU232.3

Notes:

The *phone.list* file will be used again. There is a single space character after the comma and a *tab* after the Firstname.

awk Regular Expressions

- Like sed, regular expressions are "**/**" delimited – "**/x/**"
- All of the previous regular expression metacharacters can be used with **awk**

Awk has the following extensions

`/x+/`	for one or more occurrences of x
`/x?/`	zero or one occurrence of x
`/x|y/`	matches either "x" or "y"
`(string)`	groups a string – for use with **+** or **?**

Example:

```
/t[i|o]?n[iey]+/
```

matches: tiny, tony, toni, toney, tone (and others...)

Figure 11-4. awk Regular Expressions   AU232.3

Notes:

The programming language perl has similar extensions.

awk Command Syntax

- Basic syntax

```
pattern  { actions }
pattern
         { actions }
```

- Multiple statements in an action
  - use a line break or a semi-colon

```
$ awk '/Ll/ { print $1 ; print $3 }'\
> phone.list
```

- Comments start with a # until the end of a line

```
$ awk '/Ll/ { print $1 # prints field 1
>       print $3 }' phone.list
```

Figure 11-5. awk Command Syntax AU232.3

Notes:

The three basic syntax awk program lines work as follows:

- If *pattern* is present then do the actions
- If *pattern* is present but no actions are specified, this defaults to printing the complete current line (record) to stdout.
- If *pattern* is not present, then **all** lines (records) match and each line is processed by the specified actions.

Multiple actions may be specified.

The print Statement

One useful **action** is to **print** the data!

```
awk '/pattern/ { print }' ifile > ofile
```

- awk tests each **record** of the input for the specified *pattern*
- When a match is found the **print** statement sends the entire **record** to standard output

Figure 11-6. The print Statement AU232.3

Notes:

This is the default action.

awk Fields and Records

- Referencing fields in a record

  $0 = the entire record

  $1 = the first field in the record

  $2 = the second field in the record

  ...

- To print the first two fields in records beginning with "Ll"

```
$ awk '/^Ll/ {print "Name:", $2, $1 }' \
> phone.list
Name: Lynn Llewellyn,
Name: Lee Llewellyn,
$ _
```

Figure 11-7. awk Fields and Records AU232.3

Notes:

awk sees all input as a *record* which is made up of *fields*. By default, a record is delimited by a newline ("\n"). An awk field is delimited by whitespace by default. You will see later that these defaults may be changed.

Note that the RE metacharacters "^" and "$" refer to the beginning or end of a **field** respectively.

print Examples

- Special character sequences are available for use in print strings or regular expressions

  \n newline
  \t tab
  \r carriage return

```
$ awk '/^Ll/ { print "Name:\t", $1
>       print "Number:\t", $3, "\n" }' phone.list
Name:    Llewellyn,
Number:  316-6221

Name:    Llewellyn,
Number: 817-8823

$ _
```

Figure 11-8. print Examples AU232.3

Notes:

print can take an expression following an I/O redirection to specify a pathname. The print command always ends with an end of record character. Again, this is usually newline. There is another output command, *printf* that you will see later (it allows better formatting).

Comparison Operators and Examples

To compare regular expressions or strings with values:

```
==   equal to              !=    not equal to
<    less than             <=    less than or equal to
>    greater than          >=    greater than or equal to
~    matched by RE         !~    not matched by RE
||   logical "or"          &&    logical "and"
```

Examples

`$1 ~ /x/` field one matches regular expression x

`$1 !~ "No"` field one doesn't match string "No"

You can use comparison operators in the ***pattern*** to select records

```
$ awk '$1 == "Terrell," { print $2, "Smythe" }' phone.list
Terry Smythe
$ _
```

Figure 11-9. Comparison Operators and Examples AU232.3

Notes:

This example finds records with the first field (Lastname) starting with T or the phone number starting with 4 or 6.

```
$ awk '$1 ~ /^[T]/ || $3 ~/^[46] / {
print }' phone.list

Terrell, Terry      617-7989
Patterson, Pat      614-6122
Robinson, Robin     411-3745
$ _
```

Arithmetic Operators

You can use the following operators to perform arithmetic:

`+`		addition
`-`		subtraction
`*`		multiplication
`/`		division
`%`		remainder
`^`		exponential (x^y, raise x to the power y)
`++x`	`x++`	pre and post increment
`--x`	`x--`	pre and post decrement
`=`		assignment (x = 4)

```
x op= y              x = x op y
                     for:  +=, -=, *=, /=, %=
```

Example

```
count = count + 2
num *= 8
```

Figure 11-10. Arithmetic Operators AU232.3

Notes:

count = count +2

Sets *count* to 2 the first time, because *count* will be automatically initialized to zero.

num *= 8

Sets *num* to 8 times its value. The first time this will make *num* zero.

User Variables and Expressions

You can define your own variables:

- Names must:
  - start with a letter or underscore
  - be followed by letters, underscores or digits
- Awk does not require variables to be defined before use

Variables are initialized as empty (numerically zero)

The empty string is null ("")

Reference by name only

Figure 11-11. User Variables and Expressions AU232.3

Notes:

It is possible to pass parameters into an *awk script*.

```
awk -v var=val -f commands_file data_file
      - or -
awk -f commands_file variable1=val1 var2=2 FS= data_file
```

You can use these methods to assign values to built-in variables or to define your own variables.

Caution: From AIX Version 4, parameters passed to an *awk* script using the second method shown above are not accessible within any *BEGIN* section — we will meet *BEGIN* section actions later on.

BEGIN and END Processing

You have seen the **pattern** and **action** awk syntax
You can also have actions at the beginning and end of input
You use the special patterns BEGIN and END

```
awk 'BEGIN { begin_action }
     pattern { action }
     pattern { action }
     END { end_action }'   file...
```

Where

BEGIN means execute the *begin_action* before any input read

END means execute *end_action* once all input has been read

Figure 11-12. BEGIN and END Processing AU232.3

Notes:

These special patterns can be very handy for explicit variable initialisation or explicit EOF processing.

BEGIN without END Example

You can use **BEGIN** to print a header to the output...

```
$ awk  'BEGIN { print "Words"}
>             { wcount = wcount + NF
>               print wcount }'  phone.list
Words
    3
    6
    9
    ...
    24
    27
$ _
```

- Here we have a BEGIN with no END
- The statements within the second set of braces were performed on every line of "phone.list" as no ***pattern*** was specified

Figure 11-13. BEGIN Without END Example AU232.3

Notes:

To determine the value of *NF* (total number of fields in the current record), an input line has to be read.

END without BEGIN Example

You can use **END** to print a trailer after the output

```
$ awk  '{ wcount = wcount + NF }
> END { print "Words: ", wcount }' phone.list
Words: 27
$ _
```

- The statement within the first set of braces refers to the main **action**
- The main **action** is performed on every line of the file "phone.list", so the final value of wcount holds the total number of fields (or words) in the file
- At the end of the input **END** actions are processed
- This prints the heading "Words:" with the total word count

Figure 11-14. END Without BEGIN Example AU232.3

Notes:

Built-In Variables

Awk provides a number of useful built-in variables:

FILENAME	the name of the current **file**
NF	total number of **fields** in the current record
NR	number of **records** encountered
FS	**input field separator** (the default is space or tab)
RS	**input record separator** (default is newline)
OFS	**output field separator** (default is space)
ORS	**output record separator** (default is newline)

Figure 11-15. Built-In Variables AU232.3

Notes:

If *NR* is placed inside an *END* action, it is the number of the last record processed.

FS can be set using a regular expression to define several possible field separators. A single space is taken as any number of spaces and tabs. "`[ ]`" would be taken as a single space, "`\t`" a tab and "`\t+`" as several tabs.

If *RS* is set to the null string `""`, *awk* will assume multi-line records — i.e. a single record may be more than a single line.

Built-In Variables Examples - 1

```
$ cat employee.list
Name, company, city, phone
Pete Davis, IBM, Augusta, 770-835-3788
Bill Moran, IBM, Gaithersburg, 301-240-8068
Tommy Todd, IBM, Atlanta, 770-835-3523
$ _
```

```
$ awk 'BEGIN { FS = "," ; OFS = ":" }
>        { print $1, $4 }' employee.list
Name: phone
Pete Davis: 770-835-3788
Bill Moran: 301-240-8068
Tommy Todd: 770-835-3523
$ _
```

Figure 11-16. Built-In Variables Examples - 1 AU232.3

Notes:

Built-In Variable Examples - 2

```
$ cat authors
R.S. Davis                  FIELD 1
Augusta, GA 30809           FIELD 2
770-835-3788                FIELD 3
                            RECORD SEPARATOR
F.W. Moran
Gaithersburg, MD 20879
301-240-8068

C.T. Todd
Atlanta, GA 30339
770-835-3523

$ awk 'BEGIN { FS="\n" ; RS="\n\n" ; OFS="\n" ; ORS="\n\n"}
> { print $1, $3
> } ' authors
```

Figure 11-17. Built-In Variables Examples - 2 AU232.3

Notes:

And the answer is:

```
R.S. Davis
770-835-3788

F.W. Moran
301-240-8068

C.T. Todd
770-835-3523
```

if - else if - else Statement

```
awk '{
        if (first logical test) {
               action if test true
        }
        else if (second logical test)   {
               action if first test false and
               second test true
        }
        else  {
               action if both tests false
        }
}' file
```

Figure 11-18. if - else if - else Statement AU232.3

Notes:

You can see that *awk* is a proper programming language. It has variables, input/output facilities and program logic constructs.

The *else if* and *else* parts of the *if* statement are optional. Comparison operators (">", "<", "==", etc.) must be used in the logical tests of the *if* statement to test for a value. Don't use the assignment operator "=", which assigns a value to a variable, if you are testing for equality use "==".

```
$ awk '{
     { if ( $2 == "Terry" )
            print $2 ", " $1 "--" $3
     }
}' phone.list
```

which gives

```
Terry, Terrell,--617-7989
```

The while Loop

```
awk   ' {
      while   (condition)   {
                    action
      }
      } ' file
```

Example

```
awk   ' {
    i = 1
    while     (i <= 4)      {
                    print   $i
                    ++i
    }
    } ' file
```

Figure 11-19. The while Loop AU232.3

Notes:

The for Loop

```
awk    '{
         for (initialise; test; increment) {
                action
         }
        }' file
```

Examples...

- **to read and print each field of the current input line**

```
for (i=1; i<=NF; i++)
          print $i
```

- **to print from the last field to the first of the current line**

```
for (i=NF; i>=1; i--)
          print $i
```

Figure 11-20. The for Loop AU232.3

Notes:

The *for* syntax can be re-written as an identical while loop:

```
awk '{
         initialise; while (test) {
                  action; increment
              }
    }'  file
```

The continue and next Statements

The **continue** statement stops the current innermost loop iteration and starts the next one:

```
awk  '{
        y = 42
        for (x=1; x<=NF; x++)  {
              if (y!=$x)
              {
                     continue
              }
              print x, $x
        }
    }'  file
```

The **next** statement causes the next **record** to be read in, and the program to start from the first ***pattern* { action }** block again:

```
awk 'BEGIN { action }
     pattern {
               action
               action
               next
               action
               }
     END { action }' file
```

Figure 11-21. The continue and next Statements AU232.3

Notes:

In *awk* there is also a *break* statement. This functions similar to a break in Shell and leaves the processing of the current loop.

The exit Statement

The **exit** statement jumps to any ***END*** processing – or out of the program if already in the ***END*** section. An exit code can be passed back to the Shell:

```
$ awk  '{
>           y = 42
>           for (x=1; x<=NF; ++x) {
>                if (y==$x) {
>                     print x, $x
>                     exit
>                }
>           }
>        }
>        END  { exit 3 }' file
$ print $?
3
$ _
```

Figure 11-22. The exit Statement AU232.3

Notes:

Arrays

- Awk allows array variables
- An array is a variable with an index
- An index is an expression in brackets
  - for example, array[10]

- Awk arrays are "associative"
  - index can be a string or number
  - no implicit order
  - to access all elements, use the **in** operator

    `for ( var in array_name )`

Be aware that all array indices are internally strings

Figure 11-23. Arrays AU232.3

Notes:

To define an array element you just use it — as with any awk variable no definition or initialization is needed. You can iterate through an array by numeric index as in

```
for ( i=1; i < 6; i++ )
     arr[ i ] = i
```

If you have a record with two text fields as fields 1 and 2, such as a database with a word followed by a definition phrase, you can use the associative array concepts as in

```
arr[ $1 ] = $2
```

If you want to delete an array, it is not sufficient to null the value. Use the *delete* command

```
delete arr[ i ]
```

printf for Formatted Printing

- One use of awk is as a report generator
- Better printing formats required
  - use **printf**
- **printf syntax:** `printf ( fmt [, args] )`

- Parentheses are optional
- *fmt* is usually a string constant with format specifications
- Specifiers are like the C language printf
- Format specification: %<char>

%s	string
%d	decimal integer
%f,%e	floating point (fixed or exponent notation)
%o	unsigned octal
%%	literal percent

Figure 11-24. printf for Formatted Printing AU232.3

Notes:

printf allows better formatting of output than *print*. For those who are familiar with the language C or C++, the format specifiers are very similar. For awk, remember that print will terminate each occurrence with the ORS but printf does not — hence the "\n" usually found at the end of format string.

Do not forget to make sure that you supply enough arguments to satisfy the number of format specifiers. It is a common error to make at first.

printf Formats

- Format specification strings can use modifiers
  `%-width.precision`
  - If width used, contents are right justified
  - use - (minus/hyphen) after % to left justify
  - precision controls
    1. number of digits to right of decimal point for numeric values
    2. maximum number of characters to print for string values

- To print Hello within #'s right justified in 10 character field
  `printf ("#%10s#\n", "Hello")`
- To print a number left justified with minimum 3 characters
  `printf ("%.3d\n", $1)`

Figure 11-25. printf Formats AU232.3

Notes:

You get more control of the output the more you specify but maybe at the cost of more complexity.

Functions in Awk

- There are four types of functions
- Three types are built-in to awk
  - general
  - arithmetic
  - string
- The fourth type is a user defined function

- General functions include
  - close
  - system
  - getline

Figure 11-26. Functions in Awk AU232.3

Notes:

The general functions allow the explicit *close()* of a file so that it can be reopened or used later in the awk script. It also has the benefit of avoiding running out of file descriptors etc. *system()* takes a string argument which is the external command to use. *getline* reads the input stream for the next record.

Built-In Arithmetic Functions

Functions available include

atan2(y,x)	arctangent of y/x in range -π to +π
cos(x)	cosine of x (x in radians)
sin(x)	sine of x
exp(x)	*e* to the power x
log(x)	natural log of x
sqrt(x)	square root of x
int(x)	truncated value of x
rand()	pseudo-random number r, 0≤r≤1

Figure 11-27. Built-In Arithmetic Functions AU232.3

Notes:

The list of arithmetic functions includes all the usual facilities. One not shown but available is *srand* that will set the random number seed. See the online documentation for details.

Built-In String Functions

Functions available include

length(s)	length of string s or of $0 if s not supplied
index(s,t)	position of substring t in s or zero if not present
match(s,r)	position in s of where RE r begins or zero
sub(r,s,t), gsub(r,s,t)	substitutes for r in t, returns 1 for OK uses $0 if t not supplied (gsub does all matches)
split(s,a,sep)	parses s into array a elements using field separator sep (use RS if not supplied)

Set by match()

RSTART	start of the match (same as the return value)
RLENGTH	length of the matching sub-string

Figure 11-28. Built-In String Functions AU232.3

Notes:

Checkpoint

1. With **awk**, what happens if I don't supply a pattern?
2. With **awk**, what happens if I don't supply the action?
3. **awk** causes the **-f** option to read instructions from a default line.
4. **awk** must have both the **BEGIN** and **END** statements.

Figure 11-29. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.
2.
3.
4.

Summary

- Regular expressions in *awk*
- Basic *awk* programming
- BEGIN and END processing
- Flow control – if, while, and for
- Leaving loops – continue, next and exit
- Awk arrays
- Better printing
- Awk functions

Figure 11-30. Summary AU232.3

Notes:

Unit 12. Putting It All Together

What This Unit Is About

This unit examines some real AIX shell scripts and looks at examples for script headers, script structure and syntax.

What You Should Be Able to Do

You should be able to recognize:

- Programs that use the Korn Shell
- Shell program headers
- Shell program structure

How You Will Check Your Progress

Accountability:

- Checkpoint questions
- Hands on group exercise

Objectives

In this unit we will see:

- Shell script uses in AIX
- Program headers
- Program structure
- Selected syntax examples

Figure 12-1. Objectives AU232.3

Notes:

Korn Shell Scripts in AIX 4.3

/usr/sbin			
automount	bosboot	cfgmir	cfgvg
chC2admin	chlv	chlvcopy	chpv
chvg	chwebconfig	clvm_cfg	cplv
dhcpaction	dhcpaction8	dhcpremove	dhcpremove8
dtappintegrate	exportvg	extendlv	extendvg
fbcheck	importvg	index_config.sh	index_unconfig.sh
IsC2admin	lsjfs	migratepv	mirrorvg
mkC2admin	mkinsttape	mklv	mklvcopy
mktcpip	mkvg	piofontin	piomisc_base
rc.bootx	redefinevg	reducevg	reorgvg
rmC2admin	rmlv	rmlvcopy	shutdown
slipcall	snap	splitlvcopy	synclvodm
syncvg	tapechk	unmirrorvg	updatelv
updatevg	varyoffvg	which_fileset	
/usr/bin			
bf	bfrpt	chdoclang	chlang
chtz	defaultbrowser	ibm3812	mkpmhlv
mksysb	mkszfile	ndx	oslevel
pmd	restvg	smit	spellin
subj	vgrind		
/etc			
rc	rc.C2	rc.bsdnet	rc.dacinet
rc.dt	rc.net	rc.net.serial	rc.powerfail
slip.logout			

Figure 12-2. Korn Shell Scripts in AIX 4.3 AU232.3

Notes:

The programs listed above were taken from an AIX Version 4.3 system — subsequent AIX Versions may use different programs or vary the function of those listed.

Remember, you can use the *file* command to determine the type of a file. Try this:

```
more $(file * | grep Korn | cut -f 1 | sed 's/://')
```

This can be useful in finding examples of certain commands or syntax.

Shell Script Uses in AIX 4.3

Shell Scripts also make up part of the AIX operating system:

Start-up and shutdown...

- `rc.*` multi-user start-up programs
- `bosboot` configures and creates a device boot image
- `mktcpip` sets required values for starting TCP/IP
- `shutdown` used to shutdown the system before power-off, or to enter maintenance mode

Documentation...

- `snap` documentation for your system

Figure 12-3. Shell Script Uses in AIX 4.3 AU232.3

Notes:

There are many other programs in AIX, but these are among the most interesting in terms of function and syntax.

We will use these files as well as some others in this unit.

Program Headers

```
#!/bin/ksh
#@(#)54       1.45 src/tcpip/usr/sbin/mktcpip/mktcpip, tcpip, tcpip43D, 9808A_43D 2/20/98
17:59:51
#
#COMPONENT_NAME: (TCPIP)
#
#FUNCTIONS: mktcpip.sh
#
#ORIGINS: 27
#
                      COPYRIGHTS HAVE BEEN DELETED TO SAVE SPACE
#
##[End of PROLOG]

#FILE NAME: mktcpip
#
#FILE DESCRIPTION: High-level shell command for performing minimal
# configuration required to get a maching up and running TCP/IP.
#
#Basic functions performed are:
# 1) the hostname is set both in the config database and in running machine
# 2) the IP address of the interface is set in the config database
# 3) /etc/hosts entries made for hostname and IP address
# 4) the IP address of teh nameserver and domain name are set
# 5) the subnet mask is set
# 6) destination and gateway routes are set
# 7) TCP/IP deamons started
#    or
# 8) Retrieve the above information for SMIT display
# 9) the cable type (bnc, dix or tp) is set in database
#
# See Usage message for explanation of parms
#
#RETURN VALUE DESCRIPTION
#        0        Successful
#        non-zero Unsuccessful
#
#
#EXTERNAL PROCEDURES CALLED: chdev, hostname, hostsent, lsdev
#                              mkdev, netstat, namerslv, /etc/rc.tcpi, route
```

Figure 12-4. Program Headers AU232.3

Notes:

This is the header of the *mktcpip* program — minus the copyrights. It clearly states what the program does. It also contains modification information, expectations and environment details. No author is noted, but it was probably a team effort!

Program Headers (Cont.)

```
#!/bin/ksh
#/usr/sbin/mktcpip
...
PATH=/bin:/usr/bin:/usr/sbin:/etc:/usr/ucb export PATH

NAME=$0

#Parse command flags arguments
set -- `getopt h:a:i:n:d:m:g:t:r:sc:D:S: $*`
if [ $? != 0 ]; then     #test for syntax error
  usage              #issue msg and don't return
fi

if [ $# -lt 3 ]; then    #test for too few parms

HOSTNAME= IPADDRESS= INTERFACE= NAMESERVER= DOMAIN= SUBNETMASK=
DESTINATION= GATEWAY= STARTTCP= SHOW= TYPE= DESTADDR= SUBCHANNEL=
RING=

while [ "$1" != "--" ]
do
   case $1 in
      -h) unset HOSTNAME
         HOSTNAME=$2 shift 2;;
...
```

Figure 12-5. Program Headers (Cont.) AU232.3

Notes:

This is after the header of the *mktcpip* program.

After the header, the program checks the arguments.

Program Structure

```
/usr/sbin/snap
#---------------------------MAIN----------------------------
trap intr_action 2
# Save off current umask and set it to 077.
UMASKSAVE=`umask`
umask 077

set -- `getopt AaDd:flgGklcnNo:prv:sStXib $*`
if [ "$?" != 0 ]
then
   usage
   exit 1
fi
userid=`id -ru`
if [ "$userid" != 0 ]
then
   echo "Must be root user [0] to use this utility"
   exit 2
fi

while [ "$1" != -- ]
do
   case $1 in
      -A)    doasync=y        #Gather async (tty) information
         action=y
         shift;;
      -a)    doall=y          #Gather all information
         dopred=y
         dosec=y
         action=y
         shift;;
      -d)    destdir=$2       #Directory to put information
         valid_dir $destdir
         shift;shift;;
...
```

Figure 12-6. Program Structure AU232.3

Notes:

In the *snap* program, it initializes variables and the environment using the *getopt* command.

Normally function definitions would appear next, but in this case they are not shown, then the main program follows.

An example (ideal) structure might be:

- Header information
- Check options and arguments
- Initialize variables and environment
- Function declarations
- Main script

Selected Syntax Examples - 1

Rather than wade through very long programs, here we have some selected interesting bits of syntax

rc.net: using exec & re-direction...

```
# Close file descriptor 1 and 2 because the parent may be
# waiting for the file desc. 1 and 2 to be closed.  The reason
# is that this shell script may spawn a child which inherits
# all the file descriptors from the parent and the child
# process may still be running after this process is
# terminated.  The file desc. 1 and 2 are not closed and leave
# the parent hanging waiting for those desc. to be finished.

LOGFILE=/tmp/rc.net.out     # LOGFILE is where all stdout goes.
>$LOGFILE                   # truncated LOGFILE.

exec 1<&-                   # close descriptor 1
exec 2<&-                   # close descriptor 2

exec 1</dev/null            # open descriptor 1
exec 2</dev/null            # open descriptor 2
```

Figure 12-7. Selected Syntax Examples - 1 AU232.3

Notes:

Selected Syntax Examples - 2

```
#!/bin/ksh
#/usr/sbin/snap
...
  TMPDIR=${TMPDIR:-$HOME/tmp}
  [[ ! -d $TMPDIR ]] && TMPDIR=/tmp
  TMPDIR=$TMPDIR/${0##/}.$$

  mkdir $TMPDIR || {
    print -u2 "${0##*/}: Could not create temporary files"
    exit 1
  }
  trap "/bin/rm -rf $TMPDIR 2>/dev/null" EXIT INT TERM QUIT HUP

  tdumpf=$TMPDIR/tmpfile.$$
  ...
```

Figure 12-8. Selected Syntax Examples - 2 AU232.3

Notes:

This small portion of code uses:

- Variable replacement/assignment
- Korn Shell test syntax
- Conditional execution
- Substring manipulation
- Flow Control
- Traps

Selected Syntax Examples - 3

/usr/sbin/snap

```
...
#
#Now proceed to call the associated functions for real
#This is pass 2 on state functions
passno-2
for i in $state
do
   state_func${i}
done

#Set the umask back to the original value
umask $UMASKSAVE
...
```

shutdown **sed** & **awk** example...

```
# NAME: tabmnt
# FUNCTION: collect the mount information and force every field
# to be separated by a tab, so that awk can look at the
# different fields.
tabmnt()
{

mount 2>/dev/null | awk '{ line[i] = "-"$0; i++; }
                    END { while ( i >= 4 ) {
                         i--; print line[i]; }
                        }' - >/tmp/mount.a

tab /tmp/mount.a
# remove extra tabs and blanks

   sed "/ /s//       /g" /tmp/mount.a \
   | sed "/                    /s//       /g" \
   | sed "/                    /s//       /g" \
   | sed "/          /s//        /g" >/tmp/mount.t
}

rm -f /tmp/mount.a 2>/dev/null
```

Figure 12-9. Selected Syntax Examples - 3 AU232.3

Notes:

This part of the main program does all the real work.

The *tab* command changes all spaces into tabs ("-*e*" to operate on leading spaces only), writing back to the input file (or standard output if standard input was used):

```
tab -e file
```


Selected Syntax Examples - 4

```
#!/usr/bin/ksh
# /usr/sbin/cfgmir
...
  #keep getting parent device until parent device is a bus
  #device or sio device
  print $PARENT_MON | egrep "bus|sio" > /dev/null 2>&1
  done ="$?"
...
  #wait (with timeout) the end of portmir
  for i in 1 2 3 4 5 6
  do
    if ps -ef | grep portmir | grep -v grep >/dev/null
    then
      sleep 1
    else
      break
    fi
  done
...
```

Figure 12-10. Selected Syntax Examples - 4 AU232.3

Notes:

The first example uses *egrep* to search for either *bus* or *sio*.

The second does not display the *grep portmir* command in the output.

Checkpoint

1. Does AIX use Korn shell scripts? How can you find them?
2. Now expand the above command to show you the name of the program and ONLY the first line of that program.
3. How does the file command know what type of file it is?

Figure 12-11. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.

2.

3.

Summary

- Shell Script uses in AIX
- Program headers
- Program structure
- Selected syntax examples

Figure 12-12. Summary AU232.3

Notes:

Unit 13. Good Practices and Review

What This Unit Is About

This unit discusses general design, overall layout, ease of maintenance, and general performance of shell scripts. It also provides a brief course summary.

What You Should Be Able to Do

After completing the unit, you should be able to:

- Understand why "plan and design" comes before "write and test"
- Use comments to your advantage
- Debug your code
- Understand some performance issues

How You Will Check Your Progress

Accountability:

- Checkpoint questions

Objectives

To write any serious script we need to:
- plan the activity
- produce "good code"

In this unit:

- Planning and design
- Documentation
- Debugging
- Performance issues
- Guidelines for scripting
- Course summary

Figure 13-1. Objectives AU232.3

Notes:

Planning and Design

As well as your favorite design methodology (Flow Charts, Data-Flow, SSADM, etc.) consider:

- functionality – clearly defined specification
- modular design – use of functions, separate programs
- environment – variables, directories
- file naming convention – for temporary files, results
- testing – individual units, integration tests, boundary conditions
- debugging code – do not forget the next maintainer

Figure 13-2. Planning and Design AU232.3

Notes:

Without a specification, how do you know when you have finished? The specification should include a description of the required output(s) and return codes, files that are to be used or created, and any environment variables that are to be used.

Modular coding often means that you can re-use bits in other programs — sharing common functions. It is also a lot easier to read, understand and maintain.

It might seem trivial, but a file naming convention will help you later on when you try to interface different programs. This may be something that the specification has set-out for you to follow.

If you don't plan to test your code from the start, you will find it much more time-consuming later on. Testing should be with sample data, or whatever is typical of the final environment, and with extreme cases — boundary testing. If you have a program that deals with numbers, test the smallest and the largest values that you can have, plus and minus one.

By including debugging code — activated by setting some flag variable for example — you can make it much easier to track down the source of a bug later on.

Use of Comments

A good programmer uses comments in a program to:

- Explain the purpose and function of the code at key points
- Describe the use of variables
- Explain complicated syntax
- Give yourself the credit (or the blame) for your work
- Mark corrections or additions

Remember to update the comments with the code

Figure 13-3. Use of Comments AU232.3

Notes:

Key points for your script might be function definitions and the start of the script. With variables perhaps you should describe the expected values. If you have a complicated — or clever — piece of script or syntax and you do not describe it in comments then you may well forget what, why and how you did it.

When giving yourself the credit do not forget the versions and dates, even if you are using one of the source code control tools. When you do the change, mark it at the top of the script (in your version history perhaps) and where the code changed.

Commenting Out

Lines can be commented out using the # comment character:

```
# command arg1 arg2
```

- no Shell interpretation is performed to the right of #
- legal anywhere, except as the only statement in a flow-control construction (if, while, until)

The "null" command can be used where commenting out would not work:

```
:  command arg1 arg2
```

- arguments are ignored, but processed as usual
- always returns 0 (true)

Figure 13-4. Commenting Out AU232.3

Notes:

Commenting out works in the Bourne Shell too.

Watch out for the second syntax using the null (:) command. When you supply variables or arguments they are evaluated and can cause unwanted side effects.

Script Layout

Some things must be done in a certain order
other things can be arranged for "good code":

- Shell control line (first in script) `#!/usr/bin/ksh`
- Header comments
- Validation of options
- Testing of arguments
- Initialization of variables
- Function definitions
- Main code

Figure 13-5. Script Layout AU232.3

Notes:

Debugging Code

Korn Shell options can help with syntax checking:

- to check the syntax of a Shell Script without running it

  `set -o noexec` or `set -n`

- for the Shell to print its input as it reads it

  `set -o verbose` or `set -v`

- an execution trace displays each command before it is run and after command line processing

  `set -o xtrace` or `set -x`

- for functions, use

  `typeset -ft function ...`

Figure 13-6. Debugging Code AU232.3

Notes:

The *PS4* variable is expanded and displayed with each *xtrace* line — set it to *$LINENO* to get Script line numbers.

Notice that you can debug a single function by appropriate use of *typeset*.

DEBUG Traps

After each simple command the Korn Shell issues the fake signals

- **DEBUG**
- ERR
- EXIT

The order is DEBUG, ERR, then any other traps, and lastly EXIT

To display the environment after each command set this trap

```
trap  "set"  DEBUG
```

When a command has a non-zero exit status, the Korn Shell sends the **ERR** signal

For example, to see what signals are causing error exits set this trap

```
trap  "kill -l $?"  ERR
```

Figure 13-7. DEBUG Traps AU232.3

Notes:

DEBUG is technically a fake signal — that is, it is not raised by the operating system but the Korn Shell itself.

Main program traps are inherited by functions, and in the Korn Shell, function traps are local to functions.

The *kill* command syntax used above was introduced with AIX Version 4. You might use "*print $?*" with earlier versions of AIX to see the return code for each error exit.

Maintaining Code

Documentation: design and comments

Clarity

- Code
- Documentation

Modularity

- Main script
- Use "good" functions or separate programs

Figure 13-8. Maintaining Code AU232.3

Notes:

Maintenance of code is at least as important as its creation. These are some issues that you may like to consider to ensure that your script can be maintained by others.

Good Functions

To write functions that are reliable and easy to maintain:

- avoid altering global variables inside a function
- define and export functions only when necessary
- do not change the working directory inside a function (why?)
- tidy up local temporary files

Figure 13-9. Good Functions AU232.3

Notes:

Remember that functions run in the same environment as the caller, so *$$* is the same for the function and its calling Shell.

Setting traps inside a function will not work with early versions of the Korn Shell, so think about portability before using traps in a function.

The answer to the question is: because any changes to the current directory remain in force once the function completes or returns.

Performance Issues for Shell Scripts

If performance is an issue

- Do not guess
- Measure!

Performance of a script means two areas:

- that of the Korn Shell
- that of the script

Remember that you should work in this order

- Get the functionality working
- Make it robust
- If you have to, make it more efficient/faster

Figure 13-10. Performance Issues for Shell Scripts AU232.3

Notes:

If you suspect performance is an issue, then get some measurements.

When tuning a script, it is more usual to make it robust before worrying about whether it needs to be faster.

Timing Commands

To report the elapsed, user and system time for a command or pipeline, use **time** in the KornShell:

- a Korn Shell reserved word (not a command)
- **time** output is to standard error
- input or output redirection applies to the command(s) under test only
- return value is that of the command(s) under test

```
$ time find / -name 'unix*' -print|sort
/unix                        find output
/usr/lib/unixtomh
real   0m25.51s              wall clock time
user   0m1.56s
sys    0m11.01s
$ _
```

Figure 13-11. Timing Commands AU232.3

Notes:

The operating system also has a *time* command (*/bin/time*). It only reports in tenths of a second, and cannot handle pipelines. There is also a *timex* operating system command that uses the *sar*, *vmstat*, or *iostat* utilities to monitor a single command.

Times for Shells

The **times** command displays how much time your current Shell and all its Sub-Shells have consumed:

```
$ times
0m0.99s 0m15.37s
0m8.61s 0m33.21s
```

- user and system timings given in hundredths of a second
- first line for the current Shell
- second line for the Sub-Shells

Figure 13-12. Helvetica- for Shells AU232.3

Notes:

The *times* command returns 0 (true) always.

Korn Shell Performance

To increase the startup speed of a new Shell:

- keep your history file **(.sh_history)** small
- minimize the size of any **$ENV** file
- use *autoload* with your functions
- use *FPATH* with your functions
- `set -o nolog` to prevent function definitions being logged in your history
- use "tracked aliases"
- try to use an **alias** in place of a simple function
- set *MAILCHECK* greater than the 600 second default

Figure 13-13. Korn Shell Performance AU232.3

Notes:

Keeping the history small reduces the Shell startup speed because it is read when the script starts. The file pointed to by the *ENV* variable is read for each Korn Shell invocation.

Setting *MAILCHECK* to 0 causes the Shell to check for new mail at every new prompt!

Korn Shell Script Performance

Tips for faster performance Shell Scripts:

- Shell built-in commands run faster than AIX ones
- Avoid command substitution where you can use `${ }` parameter expansions, *let* or pattern matching
- Note `$(< file)` is faster than `$(cat file)`
- Use multiple arguments rather than separate commands – e.g. `typeset -i a=3 b=4`
- Use `set -f` **or** `set -o noglob` if not using pathname metacharacters
- Use `{ }` grouping that is faster than `( )`
- Apply I/O re-directions to the whole of a loop syntax
- Set the *integer* attribute for suitable variables and don't use **$** for them with arithmetic expressions

Figure 13-14. Korn Shell Script Performance AU232.3

Notes:

Make sure that your *PATH* is correctly set — to prevent long search times for AIX commands! A tracked alias (see Unit 7) may also be helpful to reduce command search time. There is a table of Korn Shell builtin commands in Unit 7 also.

General programming techniques can also bring about performance benefits. Move loop invariants to before the loop — if you have a fixed command inside a loop you are repeating it many times without reason. Vary loop increments or the order of nesting — quite a bit of optimization relies on this kind of trick; for example, the obvious way to perform matrix multiplication is not the fastest!

Good Rules To Follow

1. Documentation
2. Make Backups
3. Try three times
4. Don't overlook the obvious
5. Try it, it might work
6. Never say never, always avoid always
7. There's usually another way to do it

Figure 13-15. Good Rules To Follow AU232.3

Notes:

1) Documentation:	Comment, comment, comment.
2) Make backups:	Every good user has a good backup... right?
3) Try three times:	Then get help, whether it be another person, a reference *manual*, or another set of eyes. Don’t frustrate yourself too much, you’ll go crazy!
4) Don't overlook the obvious:	The easiest soluti<on to implement is the easiest to overlook.
5) Try it, it might work:	Just be sure of Rule Number 2.
6) Never say never, always avoid always:	Either one will come back to haunt you.
7) There's usually another way to do it:	Every situation can, and will, be different. Use what works well for you.

Course Summary

Basic concepts
Shell variables and parameters
Exit status, return codes and traps
Progamming constructs – control flow
Shell commands and features
Arithmetic in Shell
Shell types and functions

Figure 13-16. Course Summary AU232.3

Notes:

Course Summary (Cont.)

More Shell variables

Regular expressions and text selection

Personal productivity – *sed*, *crontab/at*, *tar*

Using *awk*

Shell scripts in practice

Summary – good practice, debugging, performance

Figure 13-17. Course Summary (Cont.) AU232.3

Notes:

Checkpoint

1. What allows you to document your program for future reference?

2. Why is it a good idea to plan and design before you code?

3. Which statement is faster and why? $(< data.file) or $(cat data.file)

4. What set options can help in debugging a script?

Figure 13-18. Unit Checkpoint AU232.3

Notes:

Write down your answers here:

1.

2.

3.

4.

Summary

- Planning and design
- Documentation
- Debugging
- Performance issues
- Guidelines for scripting
- Course summary

Figure 13-19. Summary AU232.3

Notes:

HAPPY SCRIPTING!

Appendix A. vi Reference

Overview of Operations

Initially, when you enter a command you are in input mode. To edit, the user enters control mode by typing *ESC* and moves the cursor to the point needing correction and then inserts or deletes characters or words as needed.

Most control commands accept an optional repeat Count prior to the command.

When in *vi mode* on most systems, canonical processing is initially enabled and the command will be echoed again if the speed is 1200 baud or greater and it contains any control characters or less than one second has elapsed since the prompt was printed.

The *ESC* character terminates canonical processing for the remainder of the command and the user can then modify the command line.

This scheme has the advantages of canonical processing with the type-ahead echoing of *raw mode*.

If the option *viraw* is also set, the terminal will always have canonical processing disabled.

This mode is implicit for systems that do not support two alternate end of line delimiters, and might be helpful for certain terminals.

vi Input Edit Commands (by default the editor is in input mode)

ERASE	(User-defined erase character as defined by the stty command, usually Ctrl-h or #) Deletes previous character.
Ctrl-w	Deletes the previous blank separated word.
Ctrl-v	Escapes the next character.
Ctrl-v	Editing characters, the user's ERASE or KILL characters can be entered in a command line or in a search string if preceded by a Ctrl-v
Ctrl-V	The Ctrl-V removes the next character's editing features (if any).
\	Escapes the next ERASE or KILL character.

Motion Edit Commands

l	Moves the cursor forward (right) one character.
w	Moves the cursor forward one alphanumeric word.
W	Moves the cursor to the beginning of the next word that follows a blank.
e	Moves the cursor to end of the current word.
E	Moves the cursor to end of the current blank delimited word.
h	Moves the cursor backward (left) one character.
b	Moves the cursor backward one word.
B	Moves the cursor to the previous blank separated word.
|	Moves the cursor to the column specified by the Count parameter.
fc	Finds the next character c in the current line.

Fc	Finds the previous character c in the current line.
tc	Equivalent to f followed by h.
Tc	Equivalent to F followed by l.
;	Repeats Count times, the last single character find command.
0	Moves the cursor to start of line.
$	Moves the cursor to end of line.
^	Moves the cursor to start of line.

Text Modification Edit Commands

A	Appends text to the end of the line.
C	Deletes the current character through to the end of line and enters input mode.
d	Deletes the current character through to the end of line.
i	Enters the input mode and inserts text before the current character.
I	Inserts text before the beginning of the line.
P	Places the previous text modification before the cursor.
p	Places the previous text modification after the cursor.
R	Enters the input mode and types over the characters on the screen.
rc	Replaces the number of characters specified by the Count parameter, starting at the current cursor position, with the character(s) specified by c
x	Deletes the current character.
X	Deletes the preceding character.
.	Repeats the previous text modification command.
~	Inverts the case of the number of characters specified by the Count parameter, starting at the current cursor positions, and advances the cursor.

Search Edit Commands (these commands access your command history)

k	Fetches the previous command.
j	Moves forward through command list.
G	Fetches the command whose number is specified by the Count parameter that should precede it.
/String	Searches backward through history for a previous command containing the specified String. String is terminated by a RETURN or new-line character. If the specified string is preceded by a caret (^), the matched line must begin with String. If String is null, the previous string will be used.
?String	Same as / except that the search is in the forward direction.
n	Searches for the next match of the last pattern to / or ? commands.
N	Searches for the next match of the last pattern to / or ?, but in the opposite direction. Searches history for the String entered by the previous / command.

Other Edit Commands

y	Yanks the current character through the character to which Motion would move the cursor and puts them into the delete buffer. The text and cursor are unchanged.
Y	Yanks from the current position to the end of the line. Equivalent to y$.
u	Undo the last text modifying command.
U	Undo all the text modifying commands performed on the line.
e	Count in the input buffer. If Count is omitted, then the current line is used.

Features of "vi" with "set -o vi" only

\	Filename completion. Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended. If the match is unique, a / is appended if the file is a directory and a space is appended if the file is not a directory.
*	Appends an asterisk to the current word and attempts filename generation. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.
=	Lists the file names that match the current word as if an asterisk were appended to it.
_	(Underscore) Causes the Count word of the previous command to be appended and input mode entered. The last word is used if Count is omitted.
@Letter	Searches the alias list for an alias named Letter. If an alias of this name is defined, its value is placed into the input queue for processing.
#	Sends the line after inserting a # in front of the line. Useful for causing the current line to be inserted in the history without being executed.
Ctrl-c	Terminates the set -o vi edit
Ctrl-j	(New line) Executes the current line, regardless of the mode.
Ctrl-l	Line feeds and prints the current line. Has effect only in control mode.
Ctrl-m	(Return) Executes the current line, regardless of the mode.

Appendix B. Checkpoint Solutions

Unit 1 - Basic Shell Concepts

1. What type of file is /**dev/tty3**?

Correct Answer:

/dev/tty3 is a special device file, representing a terminal.

2. How could we find out a file type?

Correct Answer:

Use the "file" command to identify a file type.

3. How can we get .kshrc to run in an explicit Korn Shell?

Correct Answer:

export ENV="$HOME/.kshrc".

4. How can we specify the first character in a file name to be uppercase?

Correct Answer:

[[:upper:]]* or [A-Z]*.

5. How can we ignore error messages from a command?

Correct Answer:

command ... 2>/dev/null.

6. How do you make the normal output of a command appear as error output?

Correct Answer:

command ... 1>&2.

7. How can we group commands, in order to re-direct the standard output from all of them?

Correct Answer:

Use braces, or curly brackets, to surround the group and then do the redirection on the closing brace.

8. What will **kill 1** do?

Correct Answer:

Nothing. kill %1 will kill your job no.1, but kill 1 will attempt to kill process id 1, which is init, the parent of all other process. Even root cannot kill init.

9. If you have submitted a job to run in foreground, how could you move it to background?

Correct Answer:

First suspend the job with <Ctrl>-z, and then use the bg command to move it to the background.

10. How would you set up a command line recall facility?

Correct Answer:

set -o vi.

Unit 2 - Variables

1. How could we use positional parameter 3 in a shell script?

Correct Answer:

$3 or (better) ${3}.

2. Which variable contains the number of positional parameters?

Correct Answer:

$# or ${#}.

3. How can we change the value of a variable set in a different process?

Correct Answer:

This can't be done. A sub-process can only change a copy of an exported variable supplied by its parent process.

4. What is the variable ***IFS***?

Correct Answer:

Internal Field Separator used to read statements, and many other commands. It normally contains a space character, followed by a tab character, followed by a newline character.

5. How can we reset ***PS1*** to show the current directory?

Correct Answer:

export PS1='${PWD} $ '.

6. By setting a variable, how can we have a command recall facility?

Correct Answer:

set EDITOR or VISUAL to vi, emacs, or gmacs, and export it.

Unit 3 - Return Codes and Traps

1. How can you tell whether a command you have just entered was successful?

Correct Answer:

echo $? or print $?

2. How can you test if file *datafile* is non-empty?

Correct Answer:

test -s datafile or

[-s datafile] or

[[-s datafile]]

3. How can you check if you have been logged on for more than 20 minutes, and if so, print out a suitable message?

Correct Answer:

test "$SECONDS" -ge 1200 && echo Have a rest, $USER

4. How could you log off, using the kill command?

Correct Answer:

kill -9 $$ or kill $$

(The -9 is not usually necessary, unless a trap has been set.)

5. If you are a DBA is this a desirable command to terminate the <oracle_server>? **kill -KILL <oracle_server>**

Correct Answer:

Probably not — but at least you are the DBA and can clean up the situation.

6. What does this command do? **trap echo you did <Ctrl-c> 2**

Correct Answer:

Nothing! You get an error message indicating invalid syntax. It tries to identify the word 'you' as a signal. (It converts it to uppercase too). Single quotes need to be put around the echo and its arguments: trap 'echo "you did <cntrl-c>"' INT

7. How could you get <Ctrl-c> to log you off?

Correct Answer:

trap 'exit' 2.

Note: In this case, the quotes are not necessary, discipline yourself to use them anyway.

Unit 4 - Flow Control

1. What is wrong with this fragment of shell script?

```
if [ "$x" -eq 5 ]
then
        echo $x
elif [ "$x" -eq 3 ]
else
        echo "x is only 3"
        exit
fi
```

Correct Answer:

There must be a then statement after the elif.

2. What is the fundamental difference between a **while** and an **until** construct?

Correct Answer:

While statements assume "true", until statements assume "false"

3. How could we write an endless loop?

Correct Answer:

while true

4. What syntax would we use to perform a loop a finite number of times, resetting an identifier each time?

Correct Answer:

for identifier in word1 word2 word3 ...

5. Which construct is best suited to allow conditional processing, based on pattern matching?

Correct Answer:

case $identifier in

6. What would the following lines produce?

```
select word in To be or not to be
do
        :
done
```

Correct Answer:

as follows:

1) To
2) be
3) or
4) not
5) to
6) be
#?

7. Which construct is best used within the previous **do-done**? block?

Correct Answer:

case statement

8. How can we terminate one iteration of a loop and commence the next?

Correct Answer:

continue

9. How can we abruptly terminate all iterations of a loop but continue further processing in a shell script?

Correct Answer:

break

Unit 5 - Shell Commands

1. Without using redirection, how could we print information to file descriptor 2?

Correct Answer:

Use -u2 option to the print command.

2. What is wrong with the following command?
   `read speed?"mph" distance?"miles"`

Correct Answer:

read speed? "Enter MPH and DISTANCE" miles.

3. What **getopts** statement would allow you to process options **p**, and **a**, with option **t** expecting an associated value?

Correct Answer:

Specify a : after the t option getopts pat: varname

4. What command would print out the first and last positional parameters?

Correct Answer:

You must use the eval statement eval print $1 \${$#}

5. Which **set** option disables metacharacter pathname expansion?

Correct Answer:

set -o noglob or set -f

6. Which **set** options would be most useful in helping to debug a shell script?

Correct Answer:

You can do this by either using the full name options or the single letters.

set -o verbose or set -o xtrace or set -vx.

Unit 6 - Arithmetic

1. Multiply together variables **a** and **b**, using **expr**.

Correct Answer:

expr $a * $b

2. Use **expr** to multiply variable **a** by the sum of **b** and **c**.

Correct Answer:

expr $a * \($b + $c \)

3. Set variable **hex** to contain the hexadecimal value **7c**.

Correct Answer:

hex=16#7c

4. Write a **let** statement to test whether variable **a** is smaller than variable **b**.

Correct Answer:

((a<b)) or let "a < b"

5. Define a variable **num** as numeric only.

Correct Answer:

integer num

6. Increment a numeric variable **numvar**, by three.

Correct Answer:

Assuming the variable has been defined as an integer, we can use an implicit list:

numvar=numvar+3

Otherwise,

((numvar=numvar+3)) or let numvar=numvar+3

((numvar += 3)) or let numvar += 3

7. How would you calculate 6/7 to 6 decimal places?

Correct Answer:

echo "scale=6; 6/7"| bc

or

echo "scale=6 \n 6/7"| bc

answer is 0.857142

8. How would you calculate the square root of 178356025?

Correct Answer:

echo "sqrt(178356025)" | bc -l

answer is 13355

Unit 7 - Korn Shell Types, Commands and Shell Functions

1. How is an array defined?

Correct Answer:

For a new array, we can use: set -A arrayname (values) or set +A arrayname (values).

Or we can simply assign a value to any single element arrayname[17]=99.

2. How do we refer to array elements?

Correct Answer:

By using braces and square brackets:

${arrayname[99]} or we can simply assign a value to any single element.

3. How could we set a variable **users**, to contain the number of users logged onto the system?

Correct Answer:

users=$(who l wc -l) or users=`who l wc -l`

4. How would we write a function to check the readability of a file?

Correct Answer:

```
function caniread
{
if [ -r "$1" ]
then
   echo yes
   return 0
else
   echo no
   return 1
fi
```

}

5. How would we write a function to print the square root of a number, with 6 decimal places?

Correct Answer:

```
function sqrt
{
print "scale=6 \n sqrt($1)" I bc
}
```

6. How do we define local variables within a function?

Correct Answer:

With the integer or typeset commands.

7. How can we list which functions are defined?

Correct Answer:

typeset +f (-f option to list the function definitions)

8. Which command would allow you to load a library of functions?

Correct Answer:

The autoload or typeset -fu command

9. How could we create an alias to show how many minutes have elapsed since the current shell began?

Correct Answer:answer:

alias mins='echo $(expr $SECONDS / 60)'

Unit 8 - More on Shell Variables

1. What happens when the variable **TMOUT** is set and you enter the following? **TMOUT=${TMOUT:-60}**

Correct Answer:

Nothing, if TMOUT already has a value, otherwise TMOUT is given the value 60.

2. What would your prompt say if you were in your **bin** directory and you entered this: **PS1='${PWD#$HOME/} $'**.

Correct Answer:

Your prompt would read: bin $.

3. How could you find out the number of characters in the variable HOME?

Correct Answer:

Use the # operator; print ${#HOME}.

Unit 9 - Regular Expressions and Text Selection Utilities

1. What regular expression can you use to select surnames?

Correct Answer:

^[A-Z][a-z]*[^a-z]

2. What regular expression can you use to select text with repeated characters in the surname?

Correct Answer:

^.*\(.\)\1.*,

3. What command can you use to select lines in phone.list with four character first names?

Correct Answer:

grep ', [A-Z][a-z]\{3\}[^a-z]' phone.list

4. How could you count the number of processes whose PIDs are in the range 1000-9999?

Correct Answer:

ps -ef | grep '^[a-z]*[0-9]\{4\}'\

'[^0-9]' | wc -l

5. How would you convert spaces to a tab in phone.list?

Correct Answer:

Use the command

tr " " "\t" <phone.list >phone.list.nospaces

6. What would this next command accomplish? **cut -d: -f1,3,4 /etc/passwd**

Correct Answer:

This will display the username, userid, and groupid from /etc/passwd file

7. Using the **paste** command, output the /etc/passwd file so that each line of information is separated by a tab and so that the fifth, sixth and seventh fields are on a separate line from the others. (Hint: make each field a line.)

Correct Answer:

tr ":" "\n" </etc/passwd | paste -s -d"\t\t\t\n\t\t\n" -

Unit 10 - Utilities for Personal Productivity

1. Write a command line script that displays a **ps -ef** with your username as the owner of *init*.

Correct Answer:

ps -ef | grep init | sed 's/root/teamXX/'

2. How can I make phone.list appear double spaced?

Correct Answer:

sed ` a\
> ` $HOME/phone.list

3. How could you backup files in your HOME directory to tape?

Correct Answer:

Your could use (assuming tape is accessible)
tar -cv [HOME (without v is acceptable).

4. Suppose you needed to run a script but don't need the results until the next day, what command might you use?

Correct Answer:

Probably the "at" command.

5. As an administrator you decide that every weekend you will check the disk usage of users' directories in /home. Which method is preferable — using *at* or *cron/crontab*?

Correct Answer:

For the question as given, use a crontab line.

6. Construct a suitable crontab entry for the previous question.

Correct Answer:

What about
33 1 * * 6 du -s /home/*

Unit 11 - The AWK Program

1. With **awk**, what happens if I don't supply a pattern?

Correct Answer:

The action is applied to each and every line.

2. With **awk**, what happens if I don't supply the action?

Correct Answer:

The pattern is applied and matches will display to STDOUT.

3. **awk** causes the **-f** option to read instructions from a default line.

Correct Answer:

No, the -f tells awk to read instructions from a named file, for example,

awk -f check.sum phone.list.

4. **awk** must have both the **BEGIN** and **END** statements.

Correct Answer:

No, neither is necessary.

Unit 12 - Putting It All Together

1. Does AIX use Korn shell scripts? How can you find them?

Correct Answer:

grep ksh * in the proper directories.

2. Now expand the above command to show you the name of the program and ONLY the first line of that program.

Correct Answer:

head -1 $(file * | grep Korn | cut -f1 | sed 's/://').

3. How does the file command know what type of file it is?

Correct Answer:

Magic! /etc/magic!

Unit 13 - Good Practices and Review

1. What allows you to document your program for future reference?

Correct Answer:

Comments, #

2. Why is it a good idea to plan and design before you code?

Correct Answer:

It will help you to know when you are finished.

3. Which statement is faster and why? $(< data.file) or $(cat data.file)

Correct Answer:

$(< data.file) because < does not create a new process

4. What set options can help in debugging a script?

Correct Answer:

verbose, xtrace, and noexec

Bibliography

ISBN 0-13-460866-6	*UNIX Shells by Example*, Ellie Quiqley, Prentice Hall, 1997
ISBN 0-201-56324-X	*Korn Shell Programming Tutorial*, Barry Rosenberg, Addison-Wesley, 1991
SR28-5706-00	*The New Korn Shell Command and Programming Language*, M. I. Bolsky & D. G. Korn, Prentice Hall, 1995
SR28-4965-00	*UNIX Power Tools*, Peek, O'Reilly, and Loukides, O'Reilly & Associates, Inc., 1994
SR28-4856-01	*Essential System Administration*, Aeleen Frisch, O'Reilly & Associates, Inc., 1995
SR285268-00-	*Learning the Korn Shell*, Bill Rosenblatt, O'Reilly & Associates, Inc., 1993
ISBN 1-53592-001-5	*UNIX in a Nutshell*, Daniel Gilly, O'Reilly & Associates, Inc., 1994
ISBN 1-56592-225-5	*sed & awk*, Dale Dougherty & Arnold Robbins, O'Reilly &Associates, Inc., 2nd. Edition, 1997.