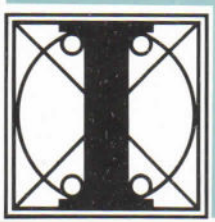
LSX Computer Line

Operating Systems

SHELL / C SHELL
X/OS Command Language

User Guide

X/OS



olivetti

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Direzione Documentazione
77, Via Jervis
10015 Ivrea (Italy)

Information from
Olivetti Documentation

LSX Computer Line

Operating Systems

SHELL / C SHELL
X/OS Command Language

User Guide

olivetti

PREFACE

This manual is the *Shell / C Shell X/OS Command Language User Guide*. It acts as a guide to the two shells provided with the LSX X/OS operating system.

SUMMARY

This manual comprises two main chapters. The first covers the default Bourne Shell, called **sh**. The second chapter covers the alternative C Shell, called **csh**. Both chapters cover the two functions performed by an LSX X/OS shell, that of command interpreter, and that of programming language. A wide range of examples is given for both, and in the case of the first chapter, some exercises are set.

REFERENCES

Read first ...

X/OS Operating Guide - Code 4055390 Y

X/OS User Guide (in this binder)

For further information, read ...

X/OS Advanced Utilities User Guide - Code 4043620 D

X/OS Utilities Reference Manual - Code 4041460 V

DISTRIBUTION: As part of software kit (W)

FIRST EDITION: December 1987 - X/OS Rel 1.0

CONTENTS

3. CSH: alternative shell

This manual is the *Shell/ C Shell X/OS Command Language User Guide*, for the X/OS operating system. It takes the form of two large chapters describing the two shells supplied.

Both shells can be used both as command interpreters and as programming languages. A wide range of examples is given, and where necessary, for example, where other X/OS utilities or shell commands are used, cross-references are given to the appropriate documents.

The first chapter describes the default Bourne Shell, called **sh**. The first half covers use of this shell as an command interpreter. It begins by explaining the shell's special characters, during which a number of the X/OS utilities are described. This first half of the chapter ends with a series of exercises. The second half covers use of the shell as a programming language. The shell's handling of variables and parameters, control statements, and expressions is covered in detail, before another series of examples are set. Finally, the answers to the exercises are given.

The second chapter describes the alternative C Shell, called **csh**. An introductory section is provided on how to use a terminal. Keyboard characters with special meanings to the shell are then described in detail, then job control is covered. There is also a section on the special variables and commands built into the shell. The second half of the chapter explains use of the C Shell as a programming language, with sections on variables, expressions and control statements.

Throughout this manual, certain conventions of presentation are used, as follows:

> this symbol in bold face is found in the example screens. It represents the system prompt displayed by X/OS whenever it is ready to accept another command line. When following the examples, this character should not be typed.

Note that the system prompt actually used by an X/OS system varies according to the way the system is configured.

key Where it is required that a specific key be pressed, the key's legend will be printed in bold face, for example, the sample screens use the symbol **CR** to indicate that the carriage return or enter key should be pressed. Also commonly encountered will be **ESC** representing the escape key, **DEL** for delete, and **space bar**. A further convention is use of the **CTRL** notation to indicate that the control key should be pressed. For example **CTRL-d** indicates that the key marked **CTRL** and the **d** keys should be pressed together. This is called a *control sequence*.

SH : DEFAULT SHELL

INTRODUCTION

This chapter describes the X/OS system default shell, **sh**. It's full name is the Bourne Shell, but this chapter will simply call it the shell. The chapter shows you how to use the shell to manage your files, to manipulate file contents, and to group commands together to make programs the shell can execute for you.

The chapter has two major sections. The first section, *Shell Command Language*, covers in detail using the shell as a command interpreter. It tells you how to use shell commands and characters with special meanings to manage files, redirect standard input and output, and execute and terminate processes. The second section, *Shell Programming*, covers in detail using the shell as a programming language. It tells you how to create, execute, and debug programs made up of commands, variables, and programming constructs like loops and case statements. Finally, it tells you how to modify your login environment.

The chapter offers many examples. You should login to your X/OS system and recreate the examples as you read the text.

In addition to the examples, there are exercises at the end of both the *Shell Command Language* and *Shell Programming* sections. The exercises can help you better understand the topics discussed. The answers to the exercises are at the end of the chapter.

SHELL COMMAND LANGUAGE

This section introduces commands and, more importantly, some characters with special meanings that let you

1. find and manipulate a group of files by using pattern matching

2. run a command in the background or at a specified time

3. run a group of commands sequentially

4. redirect standard input and output from and to files and other commands

5. terminate processes

It first covers the characters having special meanings to the shell and then covers the commands and notation for carrying out the tasks listed above. For your convenience, the following table summarizes the characters with special meanings discussed in this chapter.

CHARACTER	FUNCTION
* ? []	metacharacters that provide a shortcut for specifying file names by pattern matching
&	places commands in background mode, leaving your terminal free for other tasks
;	separates multiple commands on one command line
\	turns off the meaning of special characters such as *, ?, [,], &, ;, <, >, and |.
''	single quotes turn off the delimiting meaning of a space and the special meaning of all special characters

CHARACTER	FUNCTION
""	double quotes turn off the delimiting meaning of a space and the special meaning of all special characters except $ and `.
>	redirects output of a command into a file (replaces existing contents)
<	redirects input for a command to come from a file
>>	redirects output of a command to be added to the end of an existing file
|	creates a pipe of the output of one command to the input of another command
``	grave accents allow the output of a command to be used directly as arguments on a command line
$	used with positional parameters and user-defined variables; also used as the default shell prompt symbol

THE SHELL'S METACHARACTERS

Metacharacters, a subset of the special characters, represent other characters. They are sometimes called *wild cards*, because they are like the joker in card games that can be used for any card. The metacharacters * *(asterisk)*, ? (question mark), and [ ] (square brackets) are discussed here.

These characters are used to match file names or parts of file names, thereby simplifying the task of specifying files or groups of files as command arguments. (The files whose names match the patterns formed from these metacharacters must already exist.) This is known as file name expansion. For example, you may want to refer to all file names containing the letter *a*, all file names consisting of five letters, and so on.

The Metacharacter That Matches All Characters: the Asterisk

The asterisk (*) matches any string of characters, including a null (empty) string. You can use the * to specify a full or partial file name. The * alone refers to all the file and directory names in the current directory. The following example shows the effect of * when used as an argument to the **echo** command. Remember that the symbol > represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>echo * CR
chaptl.txt   job2       jobl      preface.txt
contents     plan.doc   readme

>
```

The **echo** command displays its arguments on your screen. Notice that the system response to **echo** * is a listing of all the file names in your current directory. However, the file names are displayed horizontally rather than in vertical columns such as those produced by the **ls** command.

The following figure summarizes the syntax and capabilities of the **echo** command.

COMMAND	OPTIONS	ARGUMENTS
echo	none	any character(s)
Description:	**echo** writes arguments, which are separated by blanks and ended with **CR**, to the output.	
Remarks:	In shell programming, **echo** is used to issue instructions, to redirect words or data into a file, and to pipe data into a command. All these uses will be discussed later in this chapter.	

Note that * is a powerful character. For example, if you type **rm** *, you will erase all the files in your current directory. Be very careful how you use it!

For another example, say you have written several reports and have named them *report*, *report1*, *report1a*, *report1b.01*, *report25*, and *report316*. By typing *report1** you can refer to all files that are part of report1, collectively. To find out how many reports you have written, you can use the **ls** command to list all files that begin with the string *report*, as shown in the following example.

```
>ls report* CR
report
report1
report1a
report1b.01
report25
report316

>
```

The * matches any characters after the string *report*, including no letters at all. Notice that * matches the files in numerical and alphabetical order. A quick and easy way to print the contents of your report files in order on your screen is by typing the following command:

```
>pr report* CR
```

Now try another exercise. Choose a character that all the file names in your current directory have in common, such as a lower case *a*. Then request a listing of those files by referring to that character. For example, if you choose a lower case *a*, type the following command line:

```
>ls *a* CR
```

The system responds by printing the names of all the files in your current directory, and the contents of all directories that contain a lower case *a*.

The * can represent characters in any part of the file name. For example, if you know that several files have their first and last letters in common, you can request a list of them on that basis. For such a request, your

command line might look like this:

```
>ls F*E CR
```

The system response will be a list of file names that begin with *F*, end with *E*, and are in the following order:

F123E
FATE
FE
Fig3.4E

The order is determined by the ASCII sort sequence: (1) numbers; (2) upper case letters; (3) lower case letters.

The Metacharacter That Matches One Character: the Question Mark

The question mark (?) matches any single character of a file name. Let's say you have written several chapters in a book that has twelve chapters, and you want a list of those you have finished through Chapter 9. Use the **ls** command with the ? to list all chapters that begin with the string *chapter* and end with any single character, as shown below:

```
>ls chapter? CR
chapter1
chapter2
chapter5
chapter9

>
```

The system responds by printing a list of all file names that match.

Although ? matches any one character, you can use it more than once in a file name. To list the rest of the chapters in your book, type:

```
>ls chapter?? CR
```

Of course, if you want to list all the chapters in the current directory, use the *:

```
>ls chapter* CR
```

Using the * or ? to Correct Typing Errors

Suppose you use the **mv** command to move a file, and you make an error and enter a character in the file name that is not printed on your screen. The system incorporates this non-printing character into the name of your file and subsequently requires it as part of the file name. If you do not include this character when you enter the file name on a command line, you get an error message. You can use * or ? to match the file name with the non-printing character and rename it to the correct name.

Try the following example.

1. Make a very short file called *trial*.

2. Type:

```
mv trial trialCTRL-gl
```

(To type **CTRL**-g you must hold down the key marked **CTRL**, and press the **g** key.)

3. Type:

```
ls triall
```

The command line and system response is as follows:

```
>ls triall CR
triall not found

>
```

4. Type:

```
ls trial?l
```

The system will respond with the file name *triall* (including the non-printing character), verifying that this file exists. Use the ? again to correct the file name.

```
>mv trial?l triall CR

>ls triall CR
triall

>
```

The Metacharacters That Match One of a Set: Brackets

Use square brackets (**[ ]**) when you want the shell to match any one of several possible characters that may appear in one position in the file name. For example, if you include **[crf]** as part of a file name pattern, the shell will look for file names that have the letter *c*, the letter *r*, or the letter *f* in the specified position, as the following example shows.

```
>ls [crf]at CR
cat
rat
fat

>
```

This command displays all file names that begin with the letter *c*, *r*, or *f* and end with the letters *at*. Characters that can be grouped within brackets in this way are collectively called a *character class*.

Brackets can also be used to specify a range of characters, whether numbers or letters. For example, if you specify

```
chapter[1-5]
```

the shell will match any files named *chapter1* through *chapter5*. This is an easy way to handle only a few chapters at a time.

Try the **pr** command with an argument in brackets:

```
pr chapter[2-4]
```

This command will print the contents of *chapter2*, *chapter3*, and *chapter4*, in that order, on your terminal.

A character class may also specify a range of letters. If you specify **[A-Z]**, the shell will look only for upper case letters; if **[a-z]**, only lower case letters.

The uses of the metacharacters are summarized in the table below.

CHARACTER	FUNCTION
*	matches any string of characters, including an empty (null) string
?	matches any single character
[ ]	matches one of the sequence of characters specified within the square brackets
[-]	matches one of the range of characters specified

Special Characters

The shell language has other special characters that perform a variety of useful functions. Some of these additional special characters are discussed in this section; others are described in a later section, *Input and Output Redirection*.

Running a Command in Background: the Ampersand

Some shell commands take considerable time to execute. The ampersand (**&**) is used to execute commands in background mode, thus freeing your terminal for other tasks. The general format for running a command in

background mode is

command **&** **CR**

You should not run interactive shell commands, for example **read** (see *Using the* **read** *Command* in this chapter), in the background.

In the example below, the shell is performing a long search in background mode. Specifically, the **grep** command is searching for the string *delinquent* in the file *accounts*.) Notice the **&** is the last character of the command line:

```
>grep delinquent accounts & CR
21940

>
```

When you run a command in the background, the X/OS system outputs a process number, in this case 21940. You can use this number to stop the execution of a background command. (Stopping the execution of processes is discussed in the section called *Executing and Terminating Processes*). The prompt on the last line means the terminal is free and waiting for your commands; **grep** has started running in background.

Running a command in background affects only the availability of your terminal; it does not affect the output of the command. Whether or not a command is run in background, it prints its output on your terminal screen, unless you redirect it to a file. (See *Redirecting Output*, below, for details.)

If you want a command to continue running in background after you log off, you can submit it with the **nohup**

command. (This is discussed in *Using the* **nohup** *Command*, below.)

Executing Commands Sequentially: the Semicolon

You can type two or more commands on one line as long as each pair is separated by a semicolon (;), as follows:

command1;*command2*;*command3* **CR**

The X/OS system executes the commands in the order that they appear in the line and prints all output on the screen. This process is called sequential execution.

Try this exercise to see how the ; works. Type

```
>cd;pwd;ls CR
```

The shell executes these commands sequentially:

1. **cd** changes your location to your login directory
2. **pwd** prints the full path name of your current directory
3. **ls** lists the files in your current directory

If you do not want the system's responses to these commands to appear on your screen, refer to *Redirecting Output* for instructions.

Turning Off Special Meanings: the Backslash

The shell interprets the backslash (\) as an escape character that allows you to turn off any special meaning of the character immediately after it. To see how this works, try the following exercise. Create a two-line file called *trial* that contains the following text:

```
The all * game
was held in Summit.
```

Use the **grep** command to search for the asterisk in the file, as shown in the following example:

```
>grep \* trial CR
The all * game

>
```

The **grep** command finds the * in the text and displays the line in which it appears. Without the \, the * would be a metacharacter to the shell and would match all file names in the current directory.

Turning Off Special Meanings: Quotes

Another way to escape the meaning of a special character is to use quotation marks. Single quotes ('...') turn off the special meaning of any character. Double quotes ("...") turn off the special meaning of all characters except **$** and ` (grave accent), which retain their special meanings within double quotes. An advantage of using quotes is that numerous special characters can be enclosed in the quotes; this can be more concise than using the backslash.

For example, if your file named *trial* also contained the line

```
He really wondered why? Why???
```

you could use the **grep** command to match the line with the three question marks as follows:

```
>grep '???' trial CR
He really wondered why? Why???

>
```

If you had instead entered the command

```
grep ??? trial CR
```

the three question marks would have been used as shell metacharacters and matched all file names of length three.

Using Quotes to Turn Off the Meaning of a Space

A common use of quotes as escape characters is for turning off the special meaning of the blank space. The shell interprets a space on a command line as a delimiter between the arguments of a command. Both single and double quotes allow you to escape that meaning.

For example, to locate two or more words that appear together in text, make the words a single argument (to the **grep** command) by enclosing them in quotes. To find the two words *The all* in your file *trial*, enter the following command line:

```
>grep 'The all' trial CR
The all * game

>
```

Grep finds the string *The all* and prints the line that contains it. What would happen if you did not put quotes around that string?

The ability to escape the special meaning of a space is especially helpful when you are using the **banner** command. This command prints a message across a terminal screen in large, poster size letters.

To execute **banner**, specify a message consisting of one or more arguments (in this case usually words), separated on the command line by spaces. The **banner** will use these spaces to delimit the arguments and print each argument on a separate line.

To print more than one argument on the same line, enclose the words, together, in double quotes. For example, to send a birthday greeting to another user, type:

```
banner happy birthday to you
```

The command prints your message as a four-line banner. To print the same message as a three-line banner, type:

```
banner happy birthday "to you"
```

Notice that the words *to* and *you* now appear on the same line. The space between them has lost its meaning as a delimiter.

The table below summarizes the syntax and capabilities of the **banner** command.

COMMAND	OPTIONS	ARGUMENTS
banner	none	*characters*

Description:	**banner** displays up to ten characters in large letters.
Remarks:	Later in this chapter, you will learn how to redirect the **banner** command into a file to be used as a poster.

Input and Output Redirection

In the X/OS system, some commands expect to receive their input from the keyboard (standard input) and most commands display their output at the terminal (standard output). However, the X/OS system lets you reassign the standard input and output to other files and programs. This is known as redirection. With redirection, you can tell the shell to

1. take its input from a file rather than the keyboard
2. send its output to file rather than the terminal
3. use a program as the source of data for another program

You use a set of operators, the less than sign (**<**), the greater than sign (**>**), two greater than signs (**>>**), and

the pipe (|) to redirect input and output.

Redirecting Input: the < Sign

To redirect input, specify a file name after a less than sign (<) on a command line:

```
command < file
```

For example, assume that you want use the **mail** command (see the *mail*(1) entry in *Utilities Reference Manual* for details) to send a message to another user with the login name *colleague*, and that you already have the message in a file named *report*. You can avoid retyping the message by specifying the file name as the source of input:

```
mail colleague < report
```

Redirecting Output to a File: the > Sign

To redirect output, specify a file name after the greater than sign (>) on a command line:

```
command > file
```

Note that if you redirect output to a file that already exists, the output of your command will overwrite the contents of the existing file.

Before redirecting the output of a command to a particular file, make sure that a file by that name does not already exist, unless you do not mind losing it. Because the shell does not allow you to have two files of

the same name in a directory, it will overwrite the contents of the existing file with the output of your command if you redirect the output to a file with the existing file's name. The shell does not warn you about overwriting the original file.

To make sure there is no file with the name you plan to use, run the **ls** command, specifying your proposed file name as an argument. If a file with that name exists, **ls** will list it; if not, you will receive a message that the file was not found in the current directory. For example, checking for the existence of the files *temp* and *junk* would give you the following output.

```
>ls temp CR
temp

>ls junk CR
junk:  no such file or directory

>
```

This means you can name your new output file *junk*, but you cannot name it *temp* unless you no longer want the contents of the existing *temp* file.

Appending Output to an Existing File: the >> Symbol

To keep from destroying an existing file, you can also use the double redirection symbol (**>>**), as follows:

```
command >> file
```

This appends the output of a command to the end of *file*. If *file* does not exist, it is created when you use the **>>** symbol this way.

The following example shows how to append the output of the **cat** command to an existing file. First, the **cat** command is first executed on both files without output redirection to show their respective contents. Then the contents of *trial2* are added after the last line of *trial1* by executing the **cat** command on *trial2* and redirecting the output to *trial1*.

```
>cat trial1 CR
This is the first line of trial1.
Hello.
This is the last line of trial1.

>cat trial2 CR
This is the beginning of trial2.
Hello.
This is the end of trial2.

>cat trial2 >> trial1 CR

>cat trial1 CR
This is the first line of trial1.
Hello.
This is the last line of trial1.
This is the beginning of trial2.
Hello.
This is the end of trial2.

>
```

Useful Applications of Output Redirection

Redirecting output is useful when you do not want it to appear on your screen immediately or when you want to save it. Output redirection is also especially useful when you run commands that perform clerical chores on text files. Two such commands are **spell** and **sort**.

The spell Command

The **spell** program compares every word in a file against its internal vocabulary list and prints a list of all potential misspellings on the screen. If **spell** does not have a listing for a word (such as a person's name), it will report that as a misspelling, too.

Running **spell** on a lengthy text file can take a long time and may produce a list of misspellings that is too long to fit on your screen. **spell** prints all its output at once; if it does not fit on the screen, the command scrolls it continuously off the top until it has all been displayed. A long list of misspellings will roll off your screen quickly and may be difficult to read.

You can avoid this problem by redirecting the output of **spell** to a file.

```
>spell memo > misspell CR
```

COMMAND	OPTIONS	ARGUMENTS
spell	available	*file*

Description: **spell** collects words from a specified file or files and looks them up in a spelling list. Words that are not on the spelling list are displayed on your terminal.

Options: **spell** has several options, including one for checking British spellings.

Remarks: The list of misspelled words can be redirected to a file.

See the *spell*(1) entry in the *Utilities Reference Manual* for all the available options.

The sort Command

The **sort** command arranges the lines of a specified file in alphabetical order. Because users generally want to keep a file that has been alphabetized, output redirection greatly enhances the value of this command.

Be careful to choose a new name for the file that will receive the output of the **sort** command (the alphabetized list). When **sort** is executed, the shell first empties the file that will accept the redirected output. Then it performs the sort and places the output in the blank file. If you type

```
sort list > list
```

the shell will empty *list* and then sort nothing into *list*.

Combining Background Mode and Output Redirection

Running a command in background does not affect the command's output; unless it is redirected, output is always printed on the terminal screen. If you are using your terminal to perform other tasks while a command runs in background, you will be interrupted when the command displays its output on your screen. However, if you redirect that output to a file, you can work undisturbed.

For example, in the *Special Characters* section above, you learned how to execute the **grep** command in background with **&**. Now suppose you want to find occurrences of the

word *test* in a file named *schedule*. Run the **grep** command in background and redirect its output to a file called *testfile*:

```
>grep test schedule > testfile & CR
```

You can then use your terminal for other work and examine *testfile* when you have finished it.

Redirecting Output to a Command: the Pipe

The | character is called a pipe. Pipes are powerful tools that allow you to take the output of one command and use it as input for another command without creating temporary files. A multiple command line created in this way is called a pipeline.

The general format for a pipeline is:

```
command1 | command2 | command3 ...
```

The output of *command1* is used as the input of *command2*. The output of *command2* is then used as the input for *command3*.

To understand the efficiency and power of a pipeline, consider the contrast between two methods that achieve the same results.

- To use the input/output redirection method, run one command and redirect its output to a temporary file. Then run a second command that takes the contents of the temporary file as its input. Finally, remove the temporary file after the second command has finished running.

- To use the pipeline method, run one command and pipe its output directly into a second command.

For example, say you want to mail a happy birthday message in a banner to the owner of the login *david*. Doing this without a pipeline is a three-step procedure. You must

1. Use the **banner** command and redirect its output to a temporary file:

```
banner happy birthday > message.tmp
```

2. Enter the **mail** command using *message.tmp* as its input:

```
mail david < message.tmp
```

3. Remove the temporary file:

```
rm message.tmp
```

However, by using a pipeline you can do this in one step:

```
banner happy birthday | mail david
```

A Pipeline Using the cut and date Commands

The **cut** and **date** commands provide a good example of how pipelines can increase the versatility of individual commands. The **cut** command allows you to extract part of each line in a file. It looks for characters in a specified part of the line and prints them. To specify a position in a line, use the **-c** option and identify the part of the file you want by the numbers of the spaces it occupies on the line, counting from the left-hand margin.

For example, say you want to display only the dates from a file called *birthdays*. The file contains the following list:

```
Anne    12/26
Klaus   7/4
Mary    10/18
Peter   11/9
Andy    4/23
Sam     8/12
```

The birthdays appear between the ninth and thirteenth spaces on each line. The command line to display them, and the output produced, in shown in the next example:

```
>cut -c9-13 birthdays CR
12/26
7/4
10/18
11/9
4/23
8/12
```

The syntax and capabilities of the **cut** command are shown in the table:

COMMAND	OPTIONS	ARGUMENTS
cut	**-***clist*	*file*
	-*flist* [**-d**]	

Description: **cut** extracts columns from a table or fields from each line in a file

Options: **-c** lists the number of character positions from the left. A range of numbers such as characters 1-9 can be specified by **-c1-9**

-f lists the field number from the left separated by a delimiter described by **-d**

-d gives the field delimiter for **-f**. The default is a space. If the delimiter is a colon, it would be specified by **-d:**

Remarks: If you find the **cut** command useful, you may also want to use the **paste** command and the **split** command.

The **cut** command is usually executed on a file. However, piping makes it possible to run this command on the output of other commands, too. This is useful if you want only part of the information generated by another command. For example, you may want to have the time printed. The **date** command prints the day of the week, date, and time, as follows:

```
>date CR
Sat Dec 27 13:12:32 EST 1986

>
```

Notice that the time is given between the twelfth and nineteenth spaces of the line. You can display the time (without the date) by piping the output of **date** into **cut**, specifying spaces 12-19 with the **-c** option. Your command line and its output will look like this:

```
>date | cut -c12-19 CR
13:14:56

>
```

the following table summarizes the syntax and capabilities of the **date** command.

COMMAND	OPTIONS	ARGUMENTS
date	*+%m%d%y* *+%H%M%S*	available

Description: **date** displays the current date and time on your terminal.

Options: **+%** followed by *m* (for month), *d* (for day), *y* (for year), *H* (for hour), *M* (for minute), and *S* (for second) will echo these back to your terminal. You can add explanations such as:

```
date '+%H:%M is the time'
```

Remarks: If you are working on a small computer system of which you are both a system administator and a user, you may be allowed to set the date and time using optional arguments to the date command. The *date*(1) entry in the *Utilities Reference Manual* gives the details. When working in a multi-user environment, these arguments are available only to the system administrator.

Substituting Output for an Argument

The output of any command may be captured and used as arguments on a command line. This is done by enclosing the command in grave accents (``) and placing it on the command line in the position where the output should be treated as arguments. This is known as command substitution.

For example, you can substitute the output of the **date** and **cut** pipeline command used previously for the argument in a **banner** printout by typing the following command line:

```
banner `date | cut -c12-19`
```

Notice the results: the system prints a banner with the current time.

The section called *Shell Programming* shows you how you can also use the output of a command line as the value of a variable.

Executing and Terminating Processes

This section discusses the following topics:

1. how to schedule commands to run at a later time by using the **batch** or **at** command
2. how to obtain the status of active processes
3. how to terminate active processes
4. how to keep background processes running after you have logged off

Running Commands at a Later Time: batch and at

The **batch** and **at** commands allow you to specify a command or sequence of commands to be run at a later time. With the **batch** command, the system determines when the commands run; with the **at** command, you determine when the commands run. Both commands expect input from standard input (the terminal); the list of commands entered as input from the terminal must be ended by pressing **CTRL-d**.

The **batch** command is useful if you are running a process or shell program that uses a large amount of system time. The **batch** command submits a batch job (containing the commands to be executed) to the system. The job is put in a queue, and runs when the system load falls to an acceptable level. This frees the system to respond rapidly to other input and is a courtesy to other users.

The general format for **batch** is:

```
batch
command
.
.
.
command
CTRL-d
```

If there is only one command to be run with **batch**, you can enter it as follows:

```
batch command_line
CTRL-d
```

The next example uses **batch** to execute the **grep** command at a convenient time. Here **grep** searches all files in the current directory and redirects the output to the file

dol.file.

```
>batch grep dollar * > dol-file CR
CTRL-d
job 155223141.b at Sun Dec 7 11:14:54 1986

>
```

After you submit a job with **batch**, the system responds with a job number, date, and time. This job number is not the same as the process number that the system generates when you run a command in the background.

The syntax and capabilities of the **batch** command are summarized in the following table.

COMMAND	OPTIONS	ARGUMENTS
batch	none	*command_lines*

Description:	**batch** submits a batch job which is placed in a queue and executed when the load on the system falls to an acceptable level.
Remarks:	The list of commands must end with a **CTRL-d**.

The **at** command allows you to specify an exact time to execute the commands. The general format for the **at** command is

at *time*
command
.
.
.
command
CTRL-d

The *time* argument consists of the time of day and, if the date is not today, the date.

The following example shows how to use the **at** command to mail a happy birthday banner to login *emily* on her birthday:

```
>at 8:15am Feb 27 CR
banner happy birthday | mail emily CR
CTRL-d
job 453400603.a at Thurs Feb 27 08:15:00 1986

>
```

Notice that the **at** command, like the **batch** command, responds with the job number, date, and time.

If you decide you do not want to execute the commands currently waiting in a **batch** or **at** job queue, you can erase those jobs by using the **-r** option of the **at** command with the job number. The general format is

at -r *jobnumber*

Try erasing the previous **at** job for the happy birthday banner. Type in:

```
at -r 453400603.a
```

If you have forgotten the job number, the **at -l** command will give you a list of the current jobs in the **batch** or **at** queue, as the following screen shows:

```
>at -l CR
user = mylogin 168302040.a at Sat Nov 29 13:00:00 1986
user = mylogin 453400603.a at Fri Feb 27 08:15:00 1987

>
```

Notice that the system displays the job number and the time the job will run.

Using the **at** command, **mail** yourself the file *memo* at noon, to tell you it is lunch time. (You must redirect the file into **mail** unless you use the *here document*, described in the section called *Shell Programming.*) Then try the **at** command with the **-l** option:

```
>at 12:00pm CR
mail mylogin < memo CR
CTRL-d
job 263131754.a at Jun 30 12:00:00 1986

>at -l CR
user = mylogin 263131754.a at Jun 30 12:00:00 1986

>
```

The syntax and capabilities of the **at** command are summarized in the following table.

COMMAND	OPTIONS	ARGUMENTS
at	**-r**	*time* (*date*)
	-l	*jobnumber*

Description: **at** executes commands at the time specified. You can use between one and four digits, plus a.m. or p.m. to show the time. To specify the date, give a month name followed by the number of the day. You need not enter a date if you want your job to run the same day. See the *at*(1) entry in the *Utilities Reference Manual* for further details

Options: **-r** with the job number removes previously scheduled jobs

-l with no arguments reports the job number and status of all jobs scheduled by at and batch

Remarks: Examples of how to specify times and dates using at are:

```
at 08:15am Feb 27
at 5:14pm Sept 24
```

Obtaining the Status of Running Processes

The **ps** command gives you the status of all the processes you are currently running. For example, you can use the **ps** command to show the status of all processes that you run in the background using **&** (described in the earlier section called *Special Characters*).

The next section, called *Terminating Active Processes*, discusses how you can use the PID (process identification) number to stop a command from executing. A PID is a number from 1 to 30,000 that the X/OS system assigns to each active process.

In the following example, **grep** is run in the background, and then the **ps** command is issued. The system responds with the process identification (PID) and the terminal identification (TTY) number. It also gives the cumulative execution time for each process (TIME), and the name of the command that is being executed (COMMAND).

```
>grep word * > temp & CR
28223

>ps CR
PID        TTY      TIME   COMMAND
28124      tty10    0:00   sh
28223      tty10    0:04   grep
28224      tty10    0:04   ps

>
```

Notice that the system reports a PID number for the **grep** command, as well as for the other processes that are running: the **ps** command itself, and the **sh** (shell) command that runs while you are logged in. The shell program **sh** interprets the shell commands.

COMMAND	OPTIONS	ARGUMENTS
ps	several	none

Description: **ps** displays information about active processes processes.

Options: Several, described in the *ps*(1) entry of the *Utilities Reference manual*. If none are specified, **ps** displays the status of all the currently active processes associated with the current terminal.

Remarks: Gives you the PID (Process ID). This is needed to **kill** a process (stop it from executing). See the *kill*(1) entry in the *Utilities Reference Manual*.

Terminating Active Processes

The **kill** command is used to terminate active shell processes. The general format for the **kill** command is

kill *PID*

You can use the **kill** command to terminate processes that are running in background. Note that you cannot terminate background processes by pressing the **BREAK** or **DEL** key.

The following example shows how you can terminate the **grep** command that you started executing in background in the previous example.

```
>kill 28223 CR
28223 Terminated

>
```

Notice the system responds with a message and a > prompt, showing that the process has been killed. If the system cannot find the PID number you specify, it responds with an error message:

```
kill:28223:No such process
```

The syntax and capabilities of the **kill** command are summarized in the following table.

COMMAND	OPTIONS	ARGUMENTS
kill	available	job number or PID
Description:	**kill** terminates the process specified by the PID number.	

See the *kill*(1) manual page in the *Utilities Reference Manual* for all available options and an explanation of their capabilities.

Using the nohup Command

All processes are killed when you log off. If you want a background process to continue running after you log off, you must use the **nohup** command to submit that background command.

To execute the **nohup** command, follow this format:

nohup *command* **&**

Notice that you place the **nohup** command before the command you intend to run as a background process.

For example, say you want the **grep** command to search all the files in the current directory for the string *word* and redirect the output to a file called *word.list*, and you wish to log off immediately afterward. Type the command line as follows:

```
>nohup grep word * > word.list & CR
```

You can terminate the **nohup** command by using the **kill** command. The syntax and capabilities of the **nohup** command are summarized in the following table.

COMMAND	OPTIONS	ARGUMENTS
nohup	none	command line

Description: **nohup** executes a command line, even if you hang up or quit the system.

Now that you have mastered these basic shell commands and notations, use them in your shell programs! The exercises that follow will help you practice using shell command language. The answers to the exercises are at the end of the chapter.

COMMAND LANGUAGE EXERCISES

1. What happens if you use an * (asterisk) at the beginning of a file name? Try to list some of the files in a directory using the * with the last letter of one of your file names. What happens?

2. Try the following two commands; enter them as follows:

```
cat [0-9]*
```

```
echo *
```

3. Is it acceptable to use a ? at the beginning or in the middle of a file name generation? Try it.

4. Do you have any files that begin with a number? Can you list them without listing the other files in your directory? Can you list only those files that begin with a lower case letter between a and m? (Hint: use a range of numbers or letters in [ ]).

5. Is it acceptable to place a command in background mode on a line that is executing several other commands sequentially? Try it. What happens? (Hint: use ; and &.) Can the command in background mode be

placed in any position on the command line? Try placing it in various positions. Experiment with each new character that you learn to see the full power of the character.

6. Redirect the output of **pwd** and **ls** into a file named *temp* by using the following command line:

```
cd ; pwd > temp ; ls >> temp ; ed temp
```

Remember, if you want to redirect both commands to the same file, you have to use the >> (append) sign for the second redirection. If you do not, you will wipe out the information from the **pwd** command.

7. Instead of cutting the time out of the **date** response, try redirecting only the date, without the time, into **banner**. What is the only part you need to change in the time command line?

```
banner `date | cut -c12-19`
```

SHELL PROGRAMMING

You can use the shell to create programs - new commands. Such programs are also called *shell scripts*. This section tells you how to create and execute shell programs using commands, variables, positional parameters, return codes, and basic programming control structures.

The examples of shell programs in this section are shown two ways. First, the **cat** command is used in a screen to display the contents of a file containing a shell program:

```
>cat testfile CR
first command
.
.
.
last command

>
```

Second, the results of executing the shell program appear after a command line:

```
>testfile CR
program_output

>
```

You should be familiar with an editor before you try to create shell programs. Refer to the tutorials on **ed** and **vi**

SHELL PROGRAMS

Creating a Simple Shell Program

We will begin by creating a simple shell program that will do the following tasks, in order.

1. print the current directory
2. list the contents of that directory
3. display this message on your terminal: *This is the end of the shell program.*

Create a file called **dl** (short for directory list) using your editor of choice, and enter the following:

```
pwd
ls
echo This is the end of the shell program.
```

Now write and quit the file. You have just created a shell program! You can **cat** the file to display its contents, as the following screen shows:

```
>cat dl CR
pwd
ls
echo This is the end of the shell program.

>
```

Executing a Shell Program

One way to execute a shell program is to use the **sh** command. Type:

```
sh dl
```

The **dl** command is executed by **sh**, and the path name of the current directory is printed first, then the list of files in the current directory, and finally, the comment *This is the end of the shell program.* The **sh** command provides a good way to test your shell program to make sure it works.

If **dl** is a useful command, you can use the **chmod** command to make it an executable file; then you can type **dl** by

itself to execute the command it contains. The following example shows how to use the **chmod** command to make a file executable and then run the **ls -l** command to verify the changes you have made in the permissions.

```
>chmod u+x dl CR

>ls -l CR
total 2
-rw-------  1 login login  3661  Nov 2 10:28   mbox
-rwx------  1 login login    48  Nov 15 10:50  dl

>
```

Notice that **chmod** turns on permission to execute (**+x**) for the user (*u*). Now **dl** is an executable program. Try to execute it. Type:

```
>dl CR
```

You get the same results as before, when you entered **sh dl** to execute it. For further details, see the *chmod*(1) entry in the *Utilities Reference manual*.

Creating a bin Directory for Executable Files

To make your shell programs accessible from all your directories, you can make a *bin* directory from your login directory and move the shell files to your *bin*.

You must also set your shell variable **PATH** to include your **bin** directory:

```
PATH=$PATH:$HOME/bin
```

See the sections called *Variables* and *Using Shell Variables* in this chapter for more information about **PATH**.

The following example will remind you which commands are necessary. In this example, **dl** is in the login directory. Type these command lines:

```
cd

mkdir bin

mv dl bin/dl
```

Move to the *bin* directory and type the **ls -l** command. Does **dl** still have execute permission?

Now move to a directory other than the login directory, and type the following command:

```
>dl CR
```

What happened?

It is possible to give the *bin* directory another name; if you do so, you need to change your shell variable **PATH** again.

Warnings about Naming Shell Programs

You can give your shell program any appropriate file name. However, you should not give your program the same name as a system command. If you do, the system may execute your command instead of the system command. For example, if you had named your **dl** program **mv**, each time you tried to move a file, the system would have executed

your directory list program instead of the system's **mv** program.

Another problem can occur if you name the **dl** file **ls**, and then try to execute the file. You would create an infinite loop, since your program executes the **ls** command. After some time, the system would give you the following error message:

```
ls: fork failed - too many processes
```

What happened? You typed in your new command, **ls**. The shell read and executed the **pwd** command. Then it read the **ls** command in your program and tried to execute your **ls** command. This formed an infinite loop.

X/OS system designers wisely set a limit on how many times an infinite loop can execute. One way to keep this from happening is to give the path name for the system's **ls** command, */bin/ls*, when you write your own shell program.

The following **ls** shell program would work:

```
>cat ls CR
pwd
/bin/ls
echo This is the end of the shell program
```

If you name your command **ls**, then you can only execute the system **ls** command by using its full path name, */bin/ls*.

VARIABLES

Variables are the basic data objects shell programs manipulate, other than files. Here we discuss three types of variables and how you can use them:

- positional parameters
- special parameters
- named variables

Positional Parameters

A positional parameter is a variable within a shell program whose value is set from an argument specified on the command line invoking the program. Positional parameters are numbered and are referred to with a preceding **$**: **$1**, **$2**, **$3**, and so on.

A shell program may reference up to nine positional parameters. If a shell program is invoked from a command line that appears like this:

```
shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9
```

then positional parameter **$1** within the program will be assigned the value **pp1**, positional parameter **$2** within the program will be assigned the value **pp2**, and so on, when the shell program is invoked.

Create a file called **pp** (short for positional parameters) to practice positional parameter substitution. Then enter the **echo** commands shown in the following screen. Enter the command lines so that running the **cat** command on your completed file will produce the following output:

```
>cat pp CR
echo The first positional parameter is: $1
echo The second positional parameter is: $2
echo The third positional parameter is: $3
echo The fourth positional parameter is: $4

>
```

If you execute this shell program with the arguments *one*, *two*, *three*, and *four*, you will obtain the following results (first you must make the shell program **pp** executable using the **chmod** command):

```
>chmod u+x pp CR

>pp one two three four CR
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four

>
```

The following screen shows the shell program **bbday**, which mails a greeting to the login entered in the command line:

```
>cat bbday CR
banner happy birthday | mail $1

>
```

Try sending yourself a birthday greeting. If your login name is *sue*, your command line will be:

```
>bbday sue CR
```

The **who** command lists all users currently logged in on the system. How can you make a simple shell program called **whoson**, that will tell you if the owner of a particular login is currently working on the system?

Type the following command line into a file called *whoson*:

```
who | grep $1
```

The **who** command lists all current system users, and **grep** searches the output of the **who** command for a line containing the string contained as a value in the positional parameter **$1**.

Now try using your login as the argument for the new program **whoson**. For example, say your login is *sue*. When you issue the **whoson** command, the shell program substitutes *sue* for the parameter **$1** in your program and executes as if it were:

```
who | grep sue
```

The output is shown on the following screen:

```
>whoson sue CR
sue     tty26     Jan 24 13:35

>
```

If the owner of the specified login is not currently working on the system, **grep** fails and the **whoson** program

prints no output.

The shell allows a command line to contains 128 arguments. However, a shell program is restricted to referencing nine positional parameters, **$1** through **$9**, at a given time. This restriction can be worked around using the **shift**, described in the *sh*(1) entry in *Utilities Reference Manual.* The special parameter **$***, described in the next section, can also be used to access the values of all command line arguments.

Special Parameters

The **$#** parameter, when referenced within a shell program, contains the number of arguments with which the shell program was invoked. Its value can be used anywhere within the shell program.

Enter the command line shown in the following screen in an executable shell program called *get.num.* Then run the **cat** command on the file:

```
>cat get.num CR
echo The number of arguments is: $#

>
```

The program simply displays the number of arguments with which it is invoked. For example:

```
>get.num test out this program CR
The number of arguments is: 4

>
```

The **$*** special parameter, when referenced within a shell program, contains a string with all the arguments with which the shell program was invoked, starting with the first. You are not restricted to nine parameters as with the positional parameters **$1** through **$9**.

You can write a simple shell program to demonstrate **$***. Create a shell program called *show.param* that will **echo** all the parameters. Use the command line shown in the following completed file:

```
>cat show.param CR
echo The parameters for this command are: $*

>
```

show.param will echo all the arguments you give to the command. Make *show.param* executable and try it out, using the parameters **Hello. How are you?**:

```
>show.param Hello. How are you? CR
The parameters for this command are: Hello. How are you?

>
```

Notice that *show.param* echoes **Hello. How are you?**. Now try *show.param* using more than nine arguments:

```
>show.param one two 3 4 5 six 7 8 9 10 11 CR
The parameters for this command are: one two 3 4 5 six 7 8 9 10 11

>
```

Once again, *show.param* echoes all the arguments you give. The **$*** parameter can be useful if you use file name

expansion to specify arguments to the shell command.

Use the file name expansion feature with your *show.param* command file. For example, say you have several files in your directory named for chapters of a book: *chap1*, *chap2*, and so on, through *chap7*. *show.param* will print a list of all those files.

```
>show.param chap? CR
The parameters for this command are: chap1 chap2 chap3
chap4 chap5 chap6 chap7

>
```

Named Variables

Another form of variable that you can use within a shell program is a named variable. You assign values to named variables yourself. The format for assigning a value to a named variable is

```
named_variable=value
```

Notice that there are no spaces on either side of the = sign.

In the following example, *$var1* is a named variable, and *myname* is the value or character string assigned to that variable:

```
var1=myname
```

A **$** is used in front of a variable name in a shell program to reference the value of that variable. Using

the example above, the reference *$var1* tells the shell to substitute the value *myname* (assigned to *var1*), for any occurrence of the character string **!var1**.

The first character of a variable name must be a letter or an underscore. The rest of the name can be composed of letters, underscores, and digits. As in shell program file names, it is not advisable to use a shell command name as a variable name. Also, the shell has reserved some variable names you should not use for your variables. A brief explanation of these reserved shell variable names follows:

CDPATH defines the search path for the **cd** command.

HOME is the default variable for the **cd** command (home directory).

IFS defines the internal field separators (normally the space, the tab, and the carriage return).

LOGNAME is your login name.

MAIL names the file that contains your electronic mail.

PATH determines the search path used by the shell to find commands.

PS1 defines the primary prompt (default is **$**).

PS2 defines the secondary prompt (default is **>**).

TERM identifies your terminal type. It is important to set this variable if you are editing with **vi**.

TERMINFO identifies the directory to be searched for information about your terminal, for example, its screen size.

TZ defines the time zone (default is **EST5EDT**).

Many of these variables are explained in the section called *Modifying Your Login Environment*, below. You can also read more about them in the *sh*(1) entry in the *Utilities Reference Manual*.

You can see the value of these variables in your shell in two ways. First, you can type

echo $*variable_name*

The system outputs the value of *variable_name*. Second, you can use the **env** command to print out the value of all defined variables in the shell. To do this, type **env** on a line by itself; the system outputs a list of the variable names and values.

ASSIGNING A VALUE TO A VARIABLE

If you edit with **vi**, you know you can set the **TERM** variable by entering the following command line:

TERM=*terminal_name*

This is the simplest way to assign a value to a variable.

There are several other ways to do this:

1. Use the **read** command to assign input to the variable.
2. Redirect the output of a command into a variable by using command substitution with grave accents (`...`).

3. Assign a positional parameter to the variable.

The following sections discuss each of these methods in detail.

Using the read Command

The **read** command used within a shell program allows you to prompt the user of the program for the values of variables. The general format for the **read** command is:

read *variable*

The values assigned by **read** to *variable* will be substituted for **$***variable* wherever it is used in the program. If a program executes the **echo** command just before the **read** command, the program can display directions such as *Type in ....* The **read** command will wait until you type a character string, followed by a **CR** key, and then make that string the value of the variable.

The following example shows how to write a simple shell program called *num.please* to keep track of your telephone numbers. This program uses the following commands for the purposes specified:

echo to prompt you for a person's last name

read to assign the input value to the variable *name*

grep to search the file *list* for this variable

Your finished program should look like the one displayed here:

```
>cat num.please CR
echo Type in the last name:
read name
grep $name list

>
```

Create a file called *list* that contains several last names and phone numbers. Then try running *num.please*.

The next example is a program called **mknum**, which creates a list. **Mknum** includes the following commands for the purposes shown.

echo prompts for a person's name

read assigns the person's name to the variable *name*

echo asks for the person's number

read assigns the telephone number to the variable *num*

echo adds the values of the variables *name* and *num* to the file *list*

If you want the output of the **echo** command to be added to the end of *list*, you must use **>>** to redirect it. If you use **>**, *list* will contain only the last phone number you added.

Running the **cat** command on *mknum* displays the program's contents. When your program looks like this, you will be ready to make it executable (with the **chmod** command):

```
>cat mknum CR
echo Type in the name
read name
echo Type in the number
read num
echo $name $num >> list

>chmod u+x mknum CR

>
```

Try out the new programs for your phone list. In the next example, **mknum** creates a new listing for Mr. Niceguy. Then *num.please* gives you Mr. Niceguy's phone number:

```
>mknum CR
Type in the name
Mr. Niceguy CR
Type in the number
668-0007 CR


>num.please CR
Type in the last name
Niceguy CR
Mr. Niceguy 668-0007

>
```

Notice that the variable *name* accepts both **Mr.** and **Niceguy** as the value.

Substituting Command Output for the Value of a Variable

You can substitute a command's output for the value of a variable by using *command substitution*. This has the following format:

```
variable=`command`
```

The output from *command* becomes the value of *variable*.

In one of the previous examples on piping, the **date** command was piped into the **cut** command to get the correct time. That command line was the following:

```
date | cut -c12-19
```

You can put this in a simple shell program called **t** that will give you the time.

```
>cat t CR
time=`date | cut -c12-19`
echo The time is: $time

>
```

Remember there are no spaces on either side of the equal sign. Make the file executable, and you will have a program that gives you the time:

```
>chmod u+x t CR

>t CR
The time is: 10:36
```

```
>
```

Assigning Values with Positional Parameters

You can assign a positional parameter to a named parameter by using the following format:

```
var1=$1
```

The next example is a simple program called *simp.p* that assigns a positional parameter to a variable. The following screen shows the commands in *simp.p*:

```
>cat simp.p CR
var1=$1
echo $var1

>
```

Of course, you can also assign the output of a command that uses positional parameters to a variable, as follows:

```
person=`who | grep $1`
```

In the next example, the program *log.time* keeps track of your **whoson** program results. The output of **whoson** is assigned to the variable *person*, and added to the file *login.file* with the **echo** command. The last **echo** displays the value of *$person*, which is the same as the output from the **whoson** command:

```
>cat log.time CR
person=`who | grep $1`
echo $person >> login.file
echo $person

>
```

The system response to *log.time* is shown in the following screen:

```
>log.time maryann CR
maryann     tty61          Apr 11 10:26

>
```

SHELL PROGRAMMING CONSTRUCTS

The shell programming language has several constructs that give added flexibility to your programs:

- Comments let you document a program's function.
- The *here document* allows you to include within the shell program itself lines to be redirected to be the input to some command in the shell program.
- The **exit** command lets you terminate a program at a point other than the end of the program and use return codes.
- The looping constructs, **for** and **while**, allow a program to iterate through groups of commands in a loop.
- The conditional control commands, **if** and **case**, execute a group of commands only if a particular set of conditions is met.

- The **break** command allows a program to exit unconditionally from a loop.

Comments

You can place comments in a shell program in two ways. All text on a line following a # (or sterling) sign is ignored by the shell. The # sign can be at the beginning of a line, in which case the comment uses the entire line, or it can occur after a command, in which case the command is executed but the remainder of the line is ignored. The end of a line always ends a comment. The general format for a comment line is

```
#comment
```

For example, a program that contains the following lines will ignore them when it is executed:

```
# This program sends a generic birthday greeting.
# This program needs a login as
# the positional parameter.
```

Comments are useful for documenting a program's function and should be included in any program you write.

The here Document

A *here document* allows you to place into a shell program lines that are redirected to be the input of a command in that program. It is a way to provide input to a command in a shell program without needing to use a separate file. The notation consists of the redirection symbol << and a delimiter that specifies the beginning and end of the lines of input. The delimiter can be one character

or a string of characters; the ! is often used.

The general format of a *here document* is as follows:

```
command << delimiter
... input lines ...
delimiter
```

In the next example, the program **gbday** uses a *here document* to send a generic birthday greeting by redirecting lines of input into the **mail** command:

```
>cat gbday CR
mail $1 <<!
Best wishes to you on your birthday.
!

 >
```

When you use this command, you must specify the recipient's login as the argument to the command. The input included with the use of the here document is:

```
Best wishes to you on your birthday
```

For example, to send this greeting to the owner of login **mary**, type:

```
gbday mary
```

Login *mary* will receive your greeting the next time she reads her mail messages:

```
>mail CR
From mylogin Wed May 14 14:31 CDT 1986
Best wishes to you on your birthday

>
```

Using ed in a Shell Program

The here document offers a convenient and useful way to use **ed** in a shell script. For example, suppose you want to make a shell program that will enter the **ed** editor, make a global substitution to a file, write the file, and then quit **ed**. The following screen shows the contents of a program called *ch.text* which does these tasks.

```
>cat ch.text CR
echo Type in the file name.
read file1
echo Type in the exact text to be changed.
read old_text
echo Type in the exact new text to replace the above.
read new_text
ed - $file1 <<!
g/$old_text/s//$new_text/g
w
q
!

>
```

Notice the - (minus) option to the **ed** command. This option prevents the character count from being displayed on the screen. Notice, also, the format of the **ed** command for global substitution:

g/*old_text***/s//***new_text***/g**

The program uses three variables: *file1*, *old_text*, and *new_text*. When the program is run, it uses the **read** command to obtain the values of these variables. The variables provide the following information:

file the name of the file to be edited

old_text the exact text to be changed

new_text the new text

Once the variables are entered in the program, the here document redirects the global substitution, the write command, and the quit command into the **ed** command. Try the new *ch.text* command. The following screen shows sample responses to the program prompts:

```
>ch.text CR
Type in the filename.
memo CR
Type in the exact text to be changed.
Dear John: CR
Type in the exact new text to replace the above.
To whom it may concern: CR

>cat memo CR
To whom it may concern:

>
```

Notice that by running the **cat** command on the changed file, you could examine the results of the global substitution.

For further details of the **ed** utility, see the *ed*(1) entry in the *Utilities Reference Manual*, or the **ed** tutorial in the *User Guide*. The stream editor **sed** can

also be used in shell programming. You can find more information on the **sed** editor in the *sed*(1) entry in the *User Guide*.

Return Codes

Most shell commands issue return codes that indicate whether the command executed properly. By convention, if the value returned is 0 (zero) than the command executed properly; any other value indicates that it did not. The return code is not printed automatically, but is available as the value of the shell special parameter **$?**.

Checking Return Codes

After executing a command interactively, you can see its return code by typing

```
echo $?
```

Consider the following example:

```
>cat hi CR
This is file hi.

>echo $? CR
0

>cat hello CR
cat: cannot open hello

>echo $? CR
2

>
```

In the first case, the file *i* exists in your directory and has read permission for you. The **cat** command behaves as expected and outputs the contents of the file. It exits with a return code of 0, which you can see using the parameter **$?**. In the second case, the file either does not exist or does not have read permission for you. The **cat** command prints a diagnostic message and exits with a return code of 2.

Using Return Codes With the exit Command

A shell program normally terminates when the last command in the file is executed. However, you can use the **exit** command to terminate a program at some other point. Perhaps more importantly, you can also use the **exit** command to issue return codes for a shell program. For more information about **exit**, see the *exit*(2) entry in the *System Interfaces and Libraries Reference Manual*.

Looping

In the previous examples in this chapter, the commands in shell programs have been executed in sequence. The **for** and **while** looping constructs allow a program to execute a command or sequence of commands several times.

The for Loop

The **for** loop executes a sequence of commands once for each member of a list. It has the following format:

```
for variable
   in value_list
do
   command
   command
   .
```

```
.
.
command
done
```

For each iteration of the loop, the next member of the list is assigned to the variable given in the **for** clause. References to that variable may be made anywhere in the commands within the **do** clause.

It is easier to read a shell program if the looping constructs are visually clear. Since the shell ignores spaces at the beginning of lines, each section of commands can be indented as it was in the above format. Also, if you indent each command section, you can easily check to make sure each **do** has a corresponding **done** at the end of the loop.

The variable can be any name you choose. For example, if you call it *var*, then the values given in the list after the keyword **in** will be assigned in turn to *var*; references within the command list to *$var* will make the value available. If the **in** clause is omitted, the values for *var* will be the complete set of arguments given to the command and available in the special parameter **$*.** The command list between the keywords **do** and **done** will be executed once for each value.

When the commands have been executed for the last value in the list, the program will execute the next line below **done**. If there is no line, the program will end.

The easiest way to understand a shell programming construct is to try an example. Create a program that will move files to another directory Include the following commands for the purposes shown:

echo to prompt the user for a path name to the new directory.

read	to assign the path name to the variable *path*
for *variable*	to call the variable *file*; it can be referenced as *$file* in the command sequence.
in *value_list*	to supply a list of values. If the **in** clause is omitted, the list of values is assumed to be **$*** (all the arguments entered on the command line).
do *command_sequence*	to provide a command sequence. The construct for this program will be:

```
do
    mv $file $path/$file
done
```

The following screen shows the text for the shell program *mv.file*:

```
>cat mv.file CR
echo Please type in the directory path
read path
for file
   in memo1 memo2 memo3
do
   mv $file $path/$file
done

>
```

In this program the values for the variable *file* are already in the program. To change the files each time the program is invoked, assign the values using positional

parameters or the **read** command. When positional parameters are used, the **in** keyword is not needed, as the next screen shows:

```
>cat mv.file CR
echo type in the directory path
read path
for file
do
   mv $file $path/$file
done

>
```

You can move several files at once with this command by specifying a list of file names as arguments to the command. (This can be done most easily using the file name expansion mechanism described earlier).

The while Loop

Another loop construct, the **while** loop, uses two groups of commands. It will continue executing the sequence of commands in the second group, the **do ... done** list, as long as the final command in the first group, the **while** list, returns a status of true (meaning the command can be executed).

The general format of the **while** loop is as follows:

```
while
     command
     command
     .
     .
     .
     command
do
```

```
        command
        command
        .
        .
        .
        command
    done
```

For example, a program called *enter.name* uses a **while** loop to enter a list of names into a file. The program consists of the following command lines:

```
>cat enter.name CR
while
   read x
do
   echo $x>>xfile
done

>
```

With some added refinements, the program becomes:

```
>cat enter.name CR
echo Please type in each person's name and then hit carriage return
echo Please end the list of names with a CTRL-d
while read x
do
   echo $x>>xfile
done
echo xfile contains the following names:
cat xfile

>
```

Notice that after the loop is completed, the program executes the commands below the **done**.

You used special characters in the first two **echo** command lines, so you must use quotes to turn off the special meaning. The next screen shows the results of *enter.name*:

```
>enter.name CR
Please type in each person's name and then hit carriage return
Please end the list of names with a CTRL-d
Mary Lou CR
Janice CR
CTRL-d
xfile contains the following names:
Mary Lou
Janice

>
```

Notice that after the loop completes, the program prints all the names contained in *xfile*.

The Shell's Garbage Can: /dev/null

The file system has a file called */dev/null* where you can have the shell deposit any unwanted output.

Try out */dev/null* by destroying the results of the **who** command. First, type in the **who** command. The response tells you who is on the system. Now, try the **who** command, but redirect the output into */dev/null*:

```
>who > /dev/null
```

Notice that the system responded with a prompt. The output from the **who** command was placed in */dev/null* and was effectively discarded.

Conditional Constructs

if ... then

The **if** command tells the shell program to execute the **then** sequence of commands only if the final command in the **if** command list is successful. The **if** construct ends with the keyword **fi**.

The general format for the **if** construct is as follows:

```
if
        command
        command
        .
        .
        .
        command
then
        command
        command
        .
        .
        .
        command
fi
```

For example, a shell program called *search* demonstrates the use of the **if ... then** construct. *Search* uses the **grep** command to search for a word in a file. If **grep** is successful, the program will **echo** *that the word is found in the file. Copy the search* program (shown below), and try it yourself:

```
>cat search CR
echo Type in the word and the file name.
read word file
if grep $word $file
   then echo $word is in $file
fi

>
```

Notice that the **read** command assigns values to two variables. The first characters you type, up until a space, are assigned to *word*. The rest of the characters, including embedded spaces, are assigned to *file*.

A problem with this program is the unwanted display of output from the **grep** command. If you want to dispose of the system response to the **grep** command in your program, use the file */dev/null*, changing the **if** command line to the following:

```
if grep $word $file > /dev/null
```

Now execute your *search* program. It should respond only with the message specified after the **echo** command.

if ... then ... else

The **if ... then** construction can also issue an alternate set of commands with **else**, when the **if** command sequence is false. It has the following general format:

```
if
     command
     command
     .
```

```
        .
        .
        command
    then
        command
        command
        .
        .
        .
        command
    else
        command
        command
        .
        .
        .
        command
    fi
```

You can now improve your **search** command so it will tell you when it cannot find a word, as well as when it can. The following screen shows how your improved program will look:

```
>cat search CR
echo Type in the word and the file name.
read word file
if
   grep $word $file >/dev/null
then
   echo $word is in $file
else
   echo $word is NOT in $file
fi

>
```

The test Command for Loops

The **test** command, which checks to see if certain conditions are true, is a useful command for conditional constructs. If the condition is true, the loop will continue. If the condition is false, the loop will end and the next command will be executed. Some of the useful options for the **test** command are:

test *-rfile*	true if the file exists and is readable
test *-wfile*	true if the file exists and has write permission
test *-xfile*	true if the file exists and is executable
test *-sfile*	true if the file exists and has at least one character
test *var1* **-eq** *var2*	true if *var1* equals *var2*
test *var1* **-ne** *var2*	true if *var1* does not equal *var2*

You may want to create a shell program to move all the executable files in the current directory to your *bin* directory. You can use the **test -x** command to select the executable files. Review the example of the **for** construct that occurs in the *mv.file* program, shown in the following screen:

```
>cat mv.file CR
echo type in the directory path
read path
for file
do
  mv $file $path/$file
done
```

```
>
```

Create a program called *mv.ex* that includes an **if test -x** statement in the **do ... done** loop to move executable files only.

Your program will be as follows:

```
>cat mv.ex CR
echo type in the directory path
read path
for file
  do
    if test -x $file
       then
         mv $file $path/$file
    fi
  done

>
```

The directory path will be the path from the current directory to the *bin* directory. However, if you use the value for the shell variable **HOME**, you will not need to type in the path each time. **$HOME** gives the path to the login directory. **$HOME**/*bin* gives the path to your *bin*.

In the following example, *mv.ex* does not prompt you to type in the directory name, and therefore, does not read the *path* variable:

```
>cat mv.ex CR
for file
  do
    if test -x $file
       then
         mv $file $HOME/bin/$file
```

```
    fi
  done

>
```

Test the command, using all the files in the current directory, specified with the * metacharacter as the command argument. The command lines shown in the following example execute the command from the current directory and then changes to *bin* and lists the files in that directory. All executable files should be there.

```
>mv.ex * CR

>cd; cd bin; ls CR
list of executable files

>
```

case ... esac

The **case...esac** construction has a multiple choice format that allows you to choose one of several patterns and then execute a list of commands for that pattern. The pattern statements must begin with the keyword **in**, and a closing parenthesis,), must be placed after the last character of each pattern. The command sequence for each pattern is ended with ;;. The **case** construction must be ended with **esac** (the letters of the word case reversed).

The general format for the **case** construction is shown below.

```
case word
in
    pattern1)
```

```
        command
        command
        .
        .
        .
        command
    ;;
    pattern2)
        command
        command
        .
        .
        .
        command
    ;;
    patternx)
        command
        command
        .
        .
        .
        command
    ;;
    *)
        command
        command
        .
        .
        .
        command
    ;;
esac
```

The **case** construction tries to match the *word* following the word **case** with the *pattern* in the first pattern section. If there is a match, the program executes the command lines after the first pattern and up to the corresponding ;;.

If the first pattern is not matched, the program proceeds to the second pattern. Once a pattern is matched, the program does not try to match any more of the patterns, but goes to the command following **esac**.

The * used as a pattern matches any *word*, and so allows you to give a set of commands to be executed if no other pattern matches. To do this, it must be placed as the last possible pattern in the **case** construct, so that the other patterns are checked first. This provides a useful way to detect erroneous or unexpected input.

The patterns that can be specified in the *pattern* part of each section may use the metacharacters *, ?, and **[]** as described earlier in this chapter for the shell's file name expansion capability. This provides useful flexibility.

The *set.term* program contains a good example of the **case...esac** construction. This program sets the shell variable **TERM** according to the type of terminal you are using. It uses the following command line:

```
TERM=terminal_name
```

(For an explanation of the commands used, see the **vi** tutorial in the *User Guide*. In the following example, the terminal is a Teletype 4420, Teletype 5410, or Teletype 5420.)

set.term first checks to see whether the value of **TERM** is 4420. If it is, the program makes **T4** the value of **TERM**, and terminates. If it the value of **TERM** is not 4420, the program checks for other values: 5410 and 5420. It executes the commands under the first pattern that it finds, and then goes to the first command after the **esac** command.

The pattern *, meaning everything else, is included at the end of the terminal patterns. It will warn that you do not have a pattern for the terminal specified and will allow you to exit the **case** construct:

```
>cat set.term CR
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
        in
            4420)
                    TERM=T4
            ;;
            5410)
                    TERM=T5
            ;;
            5420)
                    TERM=T7
            ;;
            *)
            echo not a correct terminal type
            ;;
esac
export TERM
echo end of program

>
```

Notice the use of the **export** command. You use **export** to make a variable available within your environment and to other shell procedures. What would happen if you placed the * pattern first? The *set.term* program would never assign a value to **TERM**, since it would always match the first pattern *, which means everything.

Unconditional Control Statements: the break and continue Commands

The **break** command unconditionally stops the execution of any loop in which it is encountered, and goes to the next command after the **done**, **fi**, or **esac** statement. If there are no commands after that statement, the program ends.

In the example for *set.term*, you could have used the **break** command instead of **echo** to leave the program, as the next example shows:

```
>cat set.term CR
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
        in
            4420)
                TERM=T4
            ;;
            5410)
                TERM=T5
            ;;
            5420)
                TERM=T7
            ;;
            *)
                break
            ;;
esac
export TERM
echo end of program

>
```

The **continue** command causes the program to go immediately to the next iteration of a **do** or **for** loop without

executing the remaining commands in the loop.

DEBUGGING PROGRAMS

At times you may need to debug a program to find and correct errors. There are two options to the **sh** command (listed below) that can help you debug a program:

sh -v *shellprogname* prints the shell input lines as they are read by the system

sh -x *shellprogname* prints commands and their arguments as they are executed

To try out these two options, create a shell program that has an error in it. For example, create a file called *bug* that contains the following list of commands:

```
>cat bug CR
today=`date`
echo enter person
read person
mail $1
$person
When you log off come into my office please.
$today
MLH

>
```

Notice that *today* equals the output of the **date** command, which must be enclosed in grave accents for command substitution to occur.

The mail message sent to Tom (**$person**) at login **tommy** (**$1**) should read as the following screen shows:

```
>mail CR
From mlh  Thu  Apr 10  11:36  CST  1984
Tom
When you log off come into my office please.
Thu  Apr 10  11:36:32  CST  1986
MLH
?

>
```

If you try to execute *bug*, you will have to press the **BREAK** or **DEL** key to end the program.

To debug this program, try executing *bug* using **sh -v**. This will print the lines of the file as they are read by the system, as shown below:

```
>sh -v bug tommy CR
today=`date`
echo enter person
enter person
read person
Tom
mail $1
```

Notice that the output stops on the **mail** command, since there is a problem with **mail**. You must use the here document to redirect input into **mail**.

Before you fix the *bug* program, try executing it with **sh -x**, which prints the commands and their arguments as they are read by the system:

```
>sh -x bug tommy CR
+date
today=Thu  Apr 10  11:07:23  CST  1986
+ echo enter person
```

```
enter person
+ read person
Tom
+ mail tommy

>
```

Once again, the program stops at the **mail** command. Notice that the substitutions for the variables have been made and are displayed.

The corrected *bug* program is as follows:

```
>cat bug CR
today=`date`
echo enter person
read person
mail $1 <<!
$person
When you log off come into my office please.
$today
MLH
!

>
```

The **tee** command is a helpful command for debugging pipelines. While simply passing its standard input to its standard output, it also saves a copy of its input into the file whose name is given as an argument.

The general format of the **tee** command is:

command1 **|** **tee** *saverfile* **|***command2*

where *saverfile* is the file that saves the output of *commandl* for you to study.

For example, say you want to check on the output of the **grep** command in the following command line:

```
who | grep $1 | cut -c1-9
```

You can use **tee** to copy the output of **grep** into a file called **check**, without disturbing the rest of the pipeline.

```
who | grep $1 | tee check | cut -c1-9
```

The file *check* contains a copy of the **grep** output, as shown in the following screen:

```
>who | grep mlhmo | tee check | cut -c1-9 CR
mlhmo

>cat check CR
mlhmo   tty61   Apr 10   11:30

>
```

MODIFYING YOUR LOGIN ENVIRONMENT

The X/OS system lets you modify your login environment in several ways. One modification that users commonly want to make is to change the default values of the erase (#) and line kill (@) characters.

When you log in, the shell first examines a file in your login directory named *.profile* (pronounced *dot profile*). This file contains commands that control your shell environment.

Because the *.profile* is a file, it can be edited and changed to suit your needs. On some systems you can edit this file yourself, while on others, the system administrator does this for you. To see whether you have a *.profile* in your home directory, type:

```
ls -al $HOME
```

If you can edit the file yourself, you may want to be cautious the first few times. Before making any changes to your *.profile*, make a copy of it in another file called *safe.profile*. The command line for this is:

```
cp .profile safe.profile
```

You can add commands to your *.profile* just as you add commands to any other shell program. You can also set some terminal options with the **stty** command, and set some shell variables.

Adding Commands to Your .profile

Practice adding commands to your *.profile*. Edit the file and add the following **echo** command to the last line of the file:

```
echo Good Morning! I am ready to work for you.
```

Write and quit the editor.

Whenever you make changes to your *.profile* and you want to initiate them in the current work session, you may cause the commands in *.profile* to be executed directly using the . (dot) shell command. The shell will reinitialize your environment by reading executing the commands in your *.profile*. Try this now. Type:

```
. .profile
```

The system should respond with the following:

```
Good Morning! I am ready to work for you.
>
```

SETTING TERMINAL OPTIONS

The **stty** command can make your shell environment more convenient. There are three options you can use with **stty: -tabs, erase CTRL-h**, and **echoe**. These are used as follows:

stty -tabs — This option preserves tabs when you are printing. It expands the tab setting to eight spaces, which is

the default. The number of spaces for each tab can be changed. (See *stty*(1) in the *Utilities Reference Manual* for details.)

stty erase CTRL-h — This option allows you to use the erase key on your keyboard to erase a letter, instead of the default character **#**. Usually the **BACKSPACE** key is the erase key.

stty echoe — If you have a terminal with a screen, this option erases characters from the screen as you erase them with the **BACKSPACE** key.

If you want to use these options for the **stty** command, you can create those command lines in your *.profile* just as you would create them in a shell program. If you use the **tail** command, which displays the last few lines of a file, you can see the results of adding those four command lines to your *.profile*:

```
>tail -4 .profile CR
echo Good Morning! I am ready to work for you
stty -tabs
stty erase
stty echoe

>
```

The capabilities of the **tail** command are laid out below.

COMMAND	OPTIONS	ARGUMENTS
tail	*-n*	*filename*

Description: **tail** displays the last lines of a file.

Options: Use *-n* the specify the number (*n*) of lines to display. The default is 10. You can specify a number of blocks (*-nb*) or characters (*-nc*) instead of lines.

USING SHELL VARIABLES

Several of the variables reserved by the shell are used in your *.profile*. You can display the current value for any shell variable by entering the following command:

echo $*variable_name*

Four of the most basic of these variables are discussed next.

The HOME Variable

This variable gives the path name of your login directory. Use the **cd** command to go to your login directory and type:

```
pwd
```

What was the system response? Now type:

```
echo $HOME
```

Was the system response the same as the response to **pwd**?

$HOME is the default argument for the **cd** command. If you do not specify a directory, **cd** will move you to **$HOME**.

The PATH Variable

This variable gives the search path for finding and executing commands. To see the current values for your **PATH** variable type:

```
echo $PATH
```

The system will respond with your current **PATH** value, for example:

```
>echo $PATH CR
:/mylogin/bin:/bin:/usr/bin:/usr/lib

>
```

The colon (:) is a delimiter between path names in the string assigned to the **$PATH** variable. When nothing is specified before a :, then the current directory is understood. Notice how, in the last example, the system looks for commands in the current directory first, then in */mylogin/bin/*, then in */bin*, then in */usr/bin*, and

finally in */usr/lib*.

If you are working on a project with several other people, you may want to set up a *bin* directory for special shell programs used only by your project members. The path might be named */projectl/bin*. Edit your *.profile*, and add the entry *:/projectl/bin* to the end of your **PATH**, as in the next example.

```
PATH=":/mylogin/bin:/bin:/usr/lib:/projectl/bin"
```

The TERM Variable

This variable tells the shell what kind of terminal you are using. To put assign a value to it, you must execute the following three commands in this order:

```
TERM=terminal_name
export TERM
```

These two lines are necessary to tell the computer what type of terminal you are using.

If you do not want to specify the **TERM** variable each time you log in, add these two command lines to your *.profile*; they will be executed automatically whenever you log in.

If you log in on more than one type of terminal, it would also be useful to have your *set.term* command file in your *.profile*.

The PS1 Variable

This variable sets the primary shell prompt string (the default is the **$** sign). You can change your prompt by changing the **PS1** variable in your *.profile*.

Try the following example. Note that to use a multi-word prompt, you must enclose the phrase in quotes. Type the following variable assignment in your *.profile*.

```
PS1="Your command is my wish"
```

Now execute your *.profile* (with the . command) and watch for your new prompt sign.

```
>. .profile CR
Your command is my wish
```

The mundane **$** sign is gone forever, or at least until you delete the **PS1** variable from your *.profile*.

SHELL PROGRAMMING EXERCISES

1. Create a shell program called *TIME* from the following command line:

   ```
   banner `date | cut -c12-19`
   ```

2. Write a shell program that will give only the date in a banner display. Be careful not to give your program the same name as an X/OS system command.

3. Write a shell program that will send a note to several people on your system.

4. Redirect the **date** command without the time into a file.

5. Echo the phrase *Dear colleague* in the same file that contains the date command, without erasing the date.

6. Using the above exercises, write a shell program that will send a memo to the same people on your system mentioned in Exercise 3. Include in your memo:

   a) The current date and the words *Dear colleague* at the top of the memo

   b) The body of the memo (stored in an existing file)

   c) The closing statement

7. How can you **read** variables into the *mv.file* program?

8. Use a **for** loop to move a list of files in the current directory to another directory. How can you move all your files to another directory?

9. How can you change the program **search**, so that it searches through several files?

   Hint:

   for file
   in $*

10. Set the **stty** options for your environment.

11. Change your prompt to the word *Hello*.

12. Check the settings of the variables **$HOME**, **$TERM**, and **$PATH** in your environment.

ANSWERS TO EXERCISES

ANSWERS TO COMMAND LANGUAGE EXERCISES

The answers to the first set of exercises are as follows:

1. The * at the beginning of a file name refers to all files that end in that file name, including that file name.

```
>ls *t fBCRfR
cat
123t
new.t
t

>
```

2. The command **cat [0-9]*** will produce the following output:

```
1memo
100data
9
05name
```

   The command **echo *** will produce a list of all the files in the current directory.

3. You can place ? in any position in a file name.

4. The command **ls [0-9]*** will list only those files that start with a number.

   The command **ls [a-m]*** will list only those files that start with the letters *a* through *m*.

5. If you placed the sequential command line in the background mode, the immediate system response was the PID number for the job.

   No, the **&** (ampersand) must be placed at the end of the command line.

6. The command line would be:

```
cd; pwd > junk; ls >> junk; ed trial
```

7. Change the **-c** option of the command line to read:

```
banner `date | cut -c1-10`
```

ANSWERS TO SHELL PROGRAMMING EXERCISES

The answers to the second set of exercises are as follows:

1.

```
>cat time CR
banner `date | cut -c12-19`

>chmod u+x time CR

>time CR
(banner display of the time 10:26)

>
```

2.

```
>cat mydate CR
banner `date | cut -c1-10`

>
```

3.

```
>cat tofriends CR
echo Type in the name of the file containing the note.
read note
mail janice marylou bryan < $note

>
```

Or, if you used parameters for the logins, instead of the logins themselves, your program may have looked like this:

```
>cat tofriends CR
echo Type in the name of the file containing the note.
read note
mail $* < $note

>
```

4.

```
date | cut -c1-10 > file1
```

5.

```
echo Dear colleague >> file1
```

6.

```
>cat send.memo CR
date | cut -c1-10 > memo1
echo Dear colleague >> memo1
cat memo >> memo1
echo A memo from M. L. Kelly >> memo1
mail janice marylou bryan < memo1

>
```

7.

```
>cat mv.file CR
echo type in the directory path
read path
echo type in file names, end with CTRL-d
  while
  read file
    do
      mv $file $path/$file
    done
echo all done

>
```

8.

```
>cat mv.file CR
echo Please type in directory path
read path
for file in $*
  do
    mv $file $path/$file
  done

>
```

The command line for moving all files in the current directory is:

```
>mv.file * CR

>
```

9. See hint given with exercise 9.

```
>cat search CR
for file
  in $*
  do
    if grep $word $file >/dev/null
    then echo $word is in $file
    else echo $word is NOT in $file
    fi
  done

>
```

10. Add the following lines to your *.profile.*

```
stty -tabs
stty erase CTRL-h
stty echoe
```

11. Add the following command lines to your *.profile*

```
PS1=Hello
export PS1
```

12. To check the values of these variables in your home environment:

```
echo $HOME
echo $TERM
echo $PATH
```

INTRODUCTION

This document is an introduction to **csh**, more commonly called the C Shell. X/OS provides two shells, which have the same function, which is that of a software system to interpret command lines entered by the user. Once it has determined whether a command is valid, the shell calls up whatever resources are needed for the computer to carry it out. Each shell has its own formal language, that is, a set of commands that the user must use in order to be understood by the computer system.

It is the intention of this document to explain how this language is used, and a bit about how it works. A more formal definition of the C Shell is available in the *csh*(1) entry in the *Utilities Reference Manual*.

C SHELL COMMAND LANGUAGE

The X/OS operating system can be visualised as consisting of three concentric layers, the *kernel*, the *utilities* and the *shell*.

The inner layer is the kernel. This provides the deepest level of a computer's services, for example, memory management and input/output (I/O) control. The user rarely has any direct contact with this level.

The second layer consists of the utilities, which are the systems provided by X/OS for specific tasks such as file and directory handling, programming, peripheral control and text editing.

The top layer is the shell. When the user types in a command, the shell performs a series of checks to ensure that it is valid, then interprets it according to the computer resources required to carry it out. The various utility and kernel functions are then called. While the shell has a number of its own built-in functions, most of its activities consist of calling up other, external,

programs.

For the user, the shell is the immediately visible layer. For it to function, it must have its own command structure. This document explains how the user accesses the computer's resources via the *interface* provided by these C Shell commands. The commands and procedures provided by **csh** are explained here, and as often as possible, examples are given.

USING THE TERMINAL

The C Shell is most commonly used directly by the user via a terminal, although it also contains its own programming language. This section will explain the various features of the shell as it is used directly. Programming the shell will be covered in a later section.

Entering Commands

When a computer terminal is ready to accept the user's instructions, it displays one or more special characters on the screen. This symbol is called the *system prompt*. While the shell is occupied with a command, the system prompt disappears. As soon as the shell is ready for another command, the system prompt reappears. A flashing *cursor* indicates where the command will appear on the screen while it is being entered.

A *command* consists of one or more elements. It begins with the *command name*, which may then be contextualised or modified by one or more *arguments*. Depending on the command being used, arguments may be *mandatory*, or they may be *optional*. An example of a shell command is **cal**, which prints a calendar. It has the following form:

```
cal [month] [year]
```

The first element (**cal**) is the command name. The two following words are the arguments to **cal**. Note that they are enclosed in square brackets. This is the convention used throughout this manual for optional elements. If **cal** is used without arguments, it prints a calendar for the current month. If the month is specified, a calendar is printed for that particular month. Specifying a year causes a calendar of that particular year to be printed.

This short example implies the following: when the **cal** command is typed, the C Shell interprets it according to its own internal rules. It recognises the command name, and looks for one or both of the arguments. If it finds arguments, it checks that they make sense. If they are meaningless, an error message is displayed. If they are correctly entered, the shell calls the **cal** utility, which in turn prints the appropriate calendar.

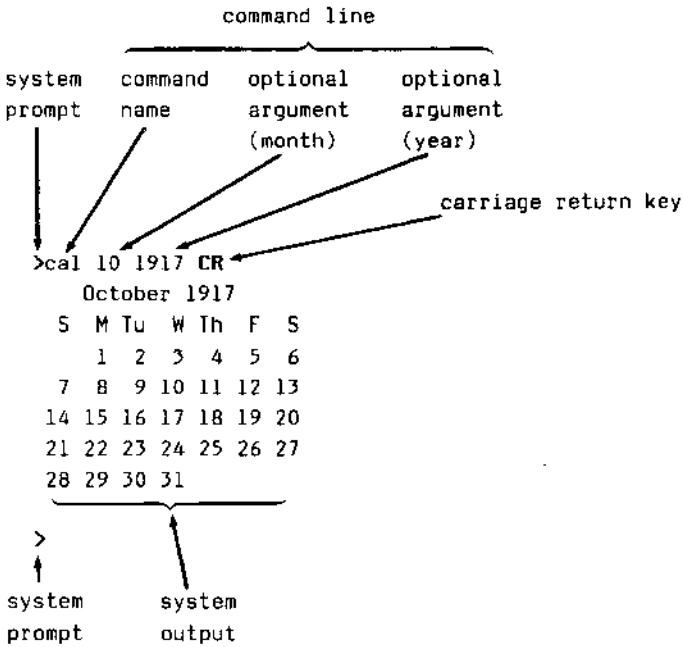
Once a command has been typed onto the screen, it must be entered, using the carriage return key. Once this has been pressed, the shell begins to process what has been typed.

The interaction between the user and the computer system therefore consists of two components. The first is the *command line* entered by the user. A command line is one or more commands entered in a single operation. Examples of multiple-command command lines will be given later. The second component is the *system response*. This consists of either an error message, or, where the command line can be correctly *executed*, some form of action on the part of the computer. Many commands will produce a message of some sort on the screen, informing the user of what has been done. Some commands are quiet, and the only indication that they have been completed is the reappearence of the system prompt.

Here is an example of a typical interaction. It begins with the system prompt, here represented by a **>** character in bold face type. Because this was displayed, the user knew that the shell was ready for a command. As soon as

the command was typed, the user entered it by pressing the carriage return key, shown by the symbol **CR**. The system response consisted of a calendar for October 1917. As soon as the command has been executed, the system prompt reappears. See the diagram, below.

Another common argument to a command takes the form of filename. Many commands handle files in some way, either creating them, changing them, or deleting them. Specifying a filename tells the command which file is to be processed. The *User Guide* gives a brief introduction to files and directories, but first, it should be explained what happens to commands that do not involve files.



Command Flags

A commonly used argument to an X/OS command takes the form of a *option*. Many commands provide more than one version, and to select a particular version, it is necessary to specify a flag on the command line. An example of a commonly used command with a large number of flags is **ls**, which, in its simplest form, lists the files in the current directory. In full, it takes the form

```
ls [-RadCxmlnogrtucpFbqisf] [names]
```

where one or more of the 21 flags can be used to specify the type of list required. The following example begins by listing the files in the current directory (**ls**). The second example (**ls -l**) lists the same files in *long* format, giving detailed information about each. The third (**ls -ra**) combines two flags, and lists the files in reverse order (the **-r** flag) including the dot directories which are usually not listed (the **-a**).

```
>ls CR
chapt1.doc
chapt2.doc
chapt3.doc
appendixA
appendixB
memos

>ls -l CR
-rwxr--r--  1 spike    489  Jul 21 14:20     chapt1.doc
-rwxr--r--  1 spike   1245  Jul 21 17:09     chapt2.doc
-rwxr--r--  1 spike   3329  Jul 23 12:32     chapt3.doc
-rwxr--r--  1 spike    998  Jul 23 11:44     appendixA
-rwxr--r--  1 spike   1025  Jul 25 13:09     appendixB
-rwxr--r--  1 spike    985  Jul 21  9:23     memos
```

```
>ls -ra CR
.
..
memos
chapt1.doc
chapt2.doc
chapt3.doc
appendixA
appendixB

>
```

Details of the flags and other arguments provided by each of the X/OS commands can be found in the various *Reference Manuals*.

Combining Commands

It is possible to combine one or more commands on a single line by separating the individual commands using the semi-colon (;). For example, to obtain a calendar of September 1939 followed by August 1945, the following command sequence can be used:

```
>cal 9 1939;cal 8 1945 CR

    September 1939
 S  M Tu  W Th  F  S
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

```
    August 1945
 S  M Tu  W Th  F  S
             1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

>
```

The two commands were executed in sequence before the system prompt reappeared.

Diagnostic Output

Whenever a command line is entered, X/OS checks to see whether it is valid. That is, the command must be recognisable, and any arguments and options must be consistent with the syntax of the commands.

All commands accepted by the system return an execution code. This may be checked by the user at any time in order to ascertain whether the command executed correctly or not.

If this is found not to be the case, some form of diagnostic response is displayed. The full range of error messages returned by X/OS can be found in the *LSX X/OS Message Book*. See also the lists provided in the manuals supplied with the various software kits.

Standard Input and Standard Output

When a command is executed, X/OS usually produces some form of output. This may show the outcome of a command, for example, a list of processed data, or it may be a status message. In some cases, this will be automatically sent to the *standard output*. This means that the output will appear on the terminal, usually the screen.

Similarly, a command may, by default, operate on the *standard input*. This means that data entered from the normal input device will be used. This is usually the keyboard.

It is commonly the case, however, that a command is to use an *input file*, that is, data that already exists. Also, the command may be required to send its output into an *output file* instead of displaying it on the screen. This facility is usually built into a command. In some cases, it may be necessary to use the shell's *re-direction operators*.

The next two sections begin by explaining files and directories, then go on to explain how to use the shell's *metacharacters*, including the re-direction operators.

Files and Directories

The X/OS files and directories system is described in detail in the *User Guide*. This is supported by several of the tutorial chapters in that volume, which cover utilities which handle files and directories. For examples of these, see the *User Guide* tutorials on **cd**, **ls**, **mkdir**, **pwd**, **rm** and **rmdir**.

As a brief recap of the information available in the *User Guide*, the following few paragraphs supply a couple of examples of files and directories in use. Note that the **cat** utility has its own tutorial in the *User Guide*.

The following examples use the **cat** command. In its simplest form, **cat** displays the contents of a file. For the purpose of the examples, the current directory is */sue/job2/part1*. The first example checks the contents of a file called *ch1.txt*, which contains five lines of rather obscure text.

```
>cat ch1.txt CR
What is electronegativity?
A measure of an atom's ability to attract an
electron. Cannot be expressed in a uniquely
quantitive way. The bond energies method of
scaling electronegativity was devised by Pauling.

>
```

The second example uses **cat** to display the contents of a file contained by another directory, in this case the file called *chptl* held in */spike/job2*. The current directory is still */sue/job2/part1*, so to display this file, it is necessary to give the full pathname.

```
>cat /spike/job2/chptl CR
What is elementarism?
A shool of psychology that analyses complex
behaviour patterns in terms of their component
parts and the inter-relationships between them.

>
```

In the next section, the shell's metacharacters are explained. The first ones relate to moving around the directory system.

THE C SHELL'S METACHARACTERS

Before going on to the directory indicators and the redirection operators, it should be explained what is meant by a *metacharacter*. These are characters that appear on the keyboard like normal characters, but which have special meanings. Whenever the shell recognises these characters, it uses them as a sort of shorthand. The first ones to be covered are the directory indicators.

The C Shell's Directory Indicators

It has already been explained that users can move between directories. The command for this is **cd** which stands for *change directory*. It has the following form:

```
cd [directory]
```

The optional *directory* argument tells **cd** which directory is to be made the current directory. It may take the form of a pathname, as in the next example, which makes */sue/job2/part2* the current directory.

```
>cd /sue/job2/part2 CR

>
```

Having worked on the files in this directory, Sue may want to add a file to the next directory up, that is, */sue/job2*. There are two ways of reaching this directory. The first example illustrates the obvious one:

```
>cd /sue/job2 CR
```

The pathname of the new working directory was simply typed in in full. The next example shows a quicker way that uses the first of the directory metacharacters.

```
>cd .. CR

>
```

The two dots (..) are used to indicate the *parent* of the current directory. By using them with **cd** in this way, the next directory up becomes the current directory. Note that to move up two levels, the .. notation can be used twice, separated by the normal slash (/) separator. From */sue/job2/part2*, the next example makes */sue* the current directory.

```
>cd ../.. CR

>
```

Note that metacharacters, directory names and filenames may be combined. In the next example, the working directory is */sue/job2/part2*. To display the contents of the file */sue/job1/chapt1*, the following command can be used:

```
>cat ../../job1/chapt1 CR
What is commodity fetishism?
The idea that commodities are transformed by the market
from simple objects into the objectification of complex
social relationships. An analysis of commodity treatment
therefore requires "recourse to to the mist-enveloped
regions of the religious world".
```

The two .. notations moved up two directory levels, and the rest of the pathname moved back down to the appropriate file.

The second metacharacter is the single dot. This stands for the current directory. In the examples, the current directory is */spike/job2/intro*. The **cp** command is used to copy the file called *intro.txt* from directory */sue/job2/part2* into the current directory. The first command shows the obvious method:

```
>cp /sue/job2/part2/intro.txt /spike/job2/intro CR
>
```

This command will work perfectly, but there is a quicker way of copying the file. The second example uses the current directory metacharacter.

```
>cp /sue/job2/part2/intro.txt . CR
>
```

This command tells **cp** to copy *intro.txt* from */sue/job2/part2* to the *current* directory, that is, */spike/job2/intro*.

The third metacharacter is the one that refers to the top of the directory tree. This position is called *root*, and it is referred to using the slash character (**/**). It is used in the same way as any other directory name, for example

```
>cd / CR
```

This command makes *root* the current directory.

Note that a pathname that begins with *root*, and specifies all the intermediate locations, for example */sue/job2/part2*, is called an *absolute* pathname. A pathname that does not begin with root is called a *relative* pathname because it specifies a location relative to the current directory, not to *root*.

Another location indicator provided by the C Shell is the tilde (~). This is used to refer to the *home* directory of a user. In the rather simple directory tree used for these examples, the home directory of user Sue is */sue*, but some systems may run to more than a hundred users, and may have tens of thousands of files. In such cases, there may be several directory levels between *root* and the users home directory. For example, there may be a series of directories called */usr1*, */usr2*, and so on, each of which has twenty or thirty users.

In the following example, user John, part of the *usr2* group, wants to copy a file called *intro.txt* from his current directory (say, */usr2/john/project1/part1*), to his home directory, */usr2/john*. The example shows how the ~ character is used to do this:

```
>cp ~/project/part1/intro.txt ~ CR

>
```

When the command is executed, the shell expands the ~ character to give the full pathname of John's home directory. In effect, the following command was given:

```
>cp /usr2/john/project/part1/intro.txt /usr2/john CR
```

The C Shell's Re-direction Operators

The C Shell provides four directional operators to channel data in a required direction. They are as follows:

< input from an existing file

> output to a new file

>> output to the end of an existing file

>& diagnostic output to a file

To channel the output from a command into a file, when it usually displays its results on the standard output, the > symbol is used. The following example uses the **cal** command (see above) to create a file called *sept1752*. Remember that when **cal** was used earlier, it printed a calendar on the standard output, that is, the screen. Note that this month seemingly lost eleven days when the calendar system was changed. The contents of the file called *sept1752* are displayed using **cat**.

```
>cal 9 1752 > sept1752 CR

>cat sept1752 CR
    September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

>
```

The second re-direction operator allows a command to use data already stored in a file. An example of a command that can use this system to access an input file is **write**. This command sets up communications between two

users. It has the following syntax:

```
write user [line]
```

It functions by setting up a link, and allowing users to talk to each other. Each user in turn types in a few lines of text, then sends them to the other user's terminal. However, it is possible to write a text file and use that as the input to the **write** command. In the following example, the user has written a short text file called *electrode.doc*. The **cat** command displays the contents of this file, then the *write* command is used to send it to another user called *basil*.

```
>cat electrode.doc CR
What is an electrode?
Either an emitter (cathode) or receiver (anode)
of electrons, for example an electron gun or a
thermionic valve, respectively.

>write basil < electrode.doc CR

>
```

This is what Basil suddenly sees:

```
Message from spike (tty12) [Tue Oct 27 16:44:32] ...

What is an electrode?
Either an emitter (cathode) or receiver (anode)
of electrons, for example an electron gun or a
thermionic valve, respectively.

EOT
```

The first line was added by **write** to identify who sent the message and the user's terminal number, and the **EOT** on the last line stands for *End of Text*.

The **>>** metanotation works just like the **>** character, except that it tells the command to attach the output to the *end* of an existing file rather than overwriting anything that may already be there. In this example, a file called *magnet.txt* contains a few lines of text. To add a line to the beginning of the file, the **>>** notation can be used.

The **cat** command is used to display the existing contents of *magnet.txt* and *magnet2.txt*, then *magnet.txt* is appended to *magnet2.txt* using **>>**. The revised contents of *magnet2.txt* are displayed using **cat**.

```
>cat magnet.txt CR
The magnetism found in iron, nickel and cobalt,
where it is possible to align all the elementary
atomic magnets in the same direction.

>cat magnet2.txt CR
What is ferromagnetism?

>cat magnet.txt >> magnet2.txt CR

>cat magnet2.txt CR
What is ferromagnetism?
The magnetism found in iron, nickel and cobalt,
where it is possible to align all the elementary
atomic magnets in the same direction.
```

The fourth metanotation is **>&**, which is used to direct the diagnostic output produced by a command, as well as the ordinary output, into a file. Diagnostic output has been already covered above.

There are some cases, for example where a program is designed to run over a long period, where a permanent record of the diagnostic output is needed. Such programs include system tests, which are often run over-night. The diagnostic output as well as the normal output is directed to a particular file using this **>&** notation, for example:

```
>testprog >& diagnosis.out CR

>cat diagnosis.out CR
diagnostic output
diagnostic output
.
.
.
diagnostic output

>
```

In this case, a program called **testprog** was run, and all output was directed into a file called *diagnosis.out*. At the termination of **testprog**, the contents of *diagnosis.out* were checked.

A further metanotation is **<<**, which reads data from the standard input until a specified word is encountered. The input takes the form of lines separated by new line characters. Each time a line is input, the shell compares it with the end of input string:

```
>sort <<end CR
Jonson, Ben CR
Keats, John CR
Blake, William CR
Milton, John CR
end CR
```

```
Blake, William
Jonson, Ben
Keats, John
Milton, John

>
```

The output from **sort** appeared immediately after the word **end** was entered. Note that the end of input marked was not itself sorted. This notation can be combined with others:

```
>sort <<end > sortfile CR
Jonson, Ben CR
Keats, John CR
Blake, William CR
Milton, John CR
end CR

>cat sortfile CR
Blake, William
Jonson, Ben
Keats, John
Milton, John

>
```

The Pipe Metacharacter

This third metacharacter acts as a link between two commands. It is expressed using the pipe character (|), and it takes the output from one command and uses it immediately as the input for another command. In fact, the pipe metacharacter re-directs the standard input of a command so that it comes from another command. For example, the **banner** command can be used to write a message in very large characters.

The example used here creates a banner message saying *Good Morning*, and sends it to another user using the **write** command. Without the pipe metacharacter, the following steps would have to be taken:

1. the **banner** command would have to be used, and the message re-directed into a file.

2. the **write** command would be used to send the message, using the file as input.

With the pipe, this sequence can be reduced to the following:

```
>banner Good Morning | write basil CR

>
```

Basil would then receive a screenful of text.

More prosaically, the pipe can be used to sort a list of files into alphabetical order. This example begins by using the **ls** command to list the files and sub-directories held in the current directory. It goes on to re-run **ls**, this time piping the output away from the screen, and into the **head** command. This displays only the first four filenames in the list.

```
>ls CR
OOREADME
chaptl.txt
contents
jobl
job2
plan.doc
preface.txt



>ls | head -4 CR
OOREADME
chaptl.txt
contents
jobl

>
```

Filename Substitution Metacharacters

It has already been mentioned that many commands use filenames as arguments. A command like **sort**, for example, will rarely use more than a couple of filenames as arguments. However, there are commands that may easily use several dozen filenames in the same command line, for example, the commands that control the printing of text.

There are two ways of specifying multiple filenames. The first is to simply type in the filenames in full. The second is to use the filename substitution notation. There are three metacharacters, as follows:

? the question mark is used to match any single character.

* the asterisk is used to match any zero or more characters.

[ ] the square brackets are used to match any range of characters.

These characters are often called *wildcards.*

In the first example, the current working directory is called */sue/job1*. This directory contains ten files. The command sequence begins with the **ls** command to list the files in that directory. The output shows a number of files called *chapt1.txt* to *chapt10.txt*. The next command is **cp**, which makes a copy of one or more files. The first version enters each filename by hand, in order to send copies of all ten files from the current directory to the directory called */sue/job2/part2*.

```
>ls CR
chapt1.txt
chapt2.txt
chapt3.txt
.
.
chapt10.txt

>cp chapt1.txt chapt2.txt chapt3.txt chapt4.txt chapt5.txt chapt6.
txt chapt7.txt chapt8.txt chapt9.txt chapt10.txt /sue/job2/part2 CR

>
```

This is obviously a slow business. The following versions use the three types of metanotation to enter the copy command more quickly. The first uses the question mark. This tells **cp** to find all filenames beginning with *chapt* and ending in *.txt*, with a *single* character in place of the question mark, and to copy them to the new directory.

```
>cp chapt?.txt /sue/job2/part2 CR
```

This command actually only copies nine of the ten files. This is because the file called *chaptl0.txt* matches all the criteria for selection *except* for the one about the *single* character. This filename has two, and is therefore not selected by **cp**.

The next command line uses the asterisk to successfully copy all ten files. Because all the files have the same filename extension, the asterisk can be used to tell **cp** to select those files beginning with any sequence of characters, of any length.

```
>cp *.txt /sue/job2/part2 CR

>
```

The next version also copies all ten files, by using the square brackets to specify a *range* of characters, from 1 to 9, and then adds the tenth filename by hand.

```
>cp chapt[1-9].txt chaptl0.txt /sue/job2/part2 CR

>
```

The easiest version was the one that used the asterisk, but this is not always appropriate, for example when not all of the ten files are to be copied. The next command line copies only three files, by using the square brackets in a slightly different way.

```
>cp chapt[289].txt /sue/job2/part2 CR

>
```

It is possible to combine these notations in a single command line. The following example shows how. Note that the **-C** option to **ls** causes the output to appear in columns.

```
>ls -C CR
chapt1.abc     chapt2.abc     chapt3.abc     appendixA     appendixB
appendixS      index          readme1        readme2       readme12
header1.txt    header2.txt    header3.doc    header4.doc

>cp *.abc appendix[AS] index readme? header[1-3].* /sue/job2/par
t2 CR

>
```

The various elements in this command line copy the following files:

**.abc*	all files with the extension *.abc*, that is *chapt1.abc* to *chapt3.abc*.
appendix[AS]	only the files called *appendixA* and *appendixS*.
index	the single file called *index*.
readme?	the files called *readme1* and *readme2*.
*header[1-3].**	the files with a filename beginning with the word *header* and ending with a number in the range 1 to 3, and having *any* extension.

Note that there are certain files, beginning with the period or full stop character, that cannot be accessed using these metacharacters.

Releasing Metacharacters

The three types of metacharacter (re-direction indicators, file expansion characters and directory indicators) are useful and commonly used. However, they pose a problem when the characters used in metanotation need to be used literally. The next example uses the **echo** command, which displays on the screen any string of characters entered as an argument.

```
>echo hello CR
hello

>
```

Metacharacters cannot be echoed in this way. For example, attempting to print an asterisk using the **echo** command has the following effect:

```
>echo * CR
chapt1.abc    chapt2.abc    chapt3.abc    appendixA    appendixB
appendixS     index         readme1       readme2      readme12
header1.txt   header2.txt   header3.doc   header4.doc

>
```

In this case, **echo** prints the names of all the files contained in the current directory. In order to tell **echo** to print an asterisk, the asterisk must be *released* from its special metacharacter meaning. This is done using an *escape sequence*, as follows:

```
>echo '*' CR
*
```

In this case, the asterisk must be enclosed in single quotes (''). This escape sequence is used to release all the metacharacters from their special meanings, with the exception of the shriek (!). The meaning of this metacharacter will be explained later, in the section called *The History Mechanism*.

A slight problem has been raised by the use of the single quotes escape sequence. The following example attempts to print a single quote mark, using **echo**:

```
>echo ' CR
Unmatched '.

>
```

The shell assumes that the single quote is the beginning of an escape sequence, and that the user has failed to type in the characters to be echoed. The system response, *Unmatched '.*, is an *error message* indicating that the command line is incorrect.

In order to correct the command, it is necessary to release the single quote. Enclosing it in single quotes of its own means that there are three single quotes, and because one is apparently still unmatched, the same error message is produced. Instead, the backslash (\) is used.

```
>echo \' CR
'

>
```

Summary of the Metacharacters

The following is a list of some of the metacharacters supplied by the shell.

.	the current directory.
..	the parent of the current directory (that is, one directory up).
/	the root directory.
~	the user's home directory.
>	re-directs output from a command to a file.
<	re-directs input to a command from a file.
>>	re-directs output from a command to the end of an existing file.
>&	re-directs diagnostic output as well as normal output to a file.
<<	re-directs input upto a specified word into a command.
|	the pipe: directs the output from one command into another command, as input.
?	matches any single character.
*	matches any zero or more characters.
[*x-y*]	matches any character in the range *x* to *y*.
[*zyz*]	matches any character from the group *x*, *y* and *z*.
'	releases the special meaning of the above metacharacters.

\ releases the special meaning of the ' character.

Directing Output to Existing Files

When using some of the above metacharacters, output can be directed into a file. In the case of >>, the new material is appended to any existing material. When the > character is used, one of two things can happen. The first possibility is that the file does not already exist. In this case, it is automatically created. The second possibility is that the file already exists. In this case, existing material is overwritten. This may cause serious loss of valuable data, so it is necessary to be careful to make sure that unique filenames are used.

The C Shell does provide a protection against this happening. For details of **noclobber** and other shell variables, see the section called *Shell Variables*, below.

JOB CONTROL

When a command line is typed and the carriage return key is pressed, the shell creates a single *job*. This is the case even where two or more individual commands are linked together using the semi-colon or pipe notation. The simplest jobs are, obviously, those with a single command element. Each job created by the shell is given an identifier number, and it is by using this number that jobs can be controlled by the user.

Foreground Jobs

All the examples used in this document so far have been *foreground* jobs. This means that the shell accepts the command line, attempts to execute it, and displays the system prompt only when it has finished with that job. While the command is being executed, no further work can be done. This is the standard method of interacting with the shell. However, this is not the only method.

Background Jobs

By adding the ampersand character (**&**) to the end of a command line, it is possible to create a background job. The & is another example of the shell's metacharacters. When the carriage return key is pressed, the shell does not wait for the command line to be executed, but immediately prints the system prompt, ready for another command. This allows several commands to be run at the same time. This is particularly useful where system check programs or printing commands are executed.

The following example uses **pack** to compress a series of files, thereby saving disk space. Because this is an automatic process not requiring user response, time may be saved by executing this command in the background, and continuing to run other commands as foreground jobs. Note that the **pack** command line uses the asterisk to process all files in the current directory with the extension *.txt*.

```
>pack *.txt & CR
[1] 2054

>command CR

>command CR

[1] - Done          pack *.txt
```

As soon as the command line was entered with the ampersand, the shell responded with a *job number* in square brackets, and a *process ID number*, in this case, 2054. The job number indicates the number of the background job owned by the user. The shell then immediately displayed the system prompt, and went on to execute a number of normal foreground jobs. As soon as the **pack** command line was completed, the shell signalled the end of the job just before printing the next system prompt.

Note that commands that print output on the screen should have their output re-directed into a file when running in the background. If this is not done, the output will appear as normal, thereby becoming confused with new command lines being entered. Note also that although this will make the screen look confused, the output will not interfere with any commands being typed in at the time. The following command will immediately generate output on the screen although it is a background job.

```
>cal 1980; cal 1982; cal 1983 & CR
[2] 2055

>
```

The next command will execute in the same way, except that its output is directed into a file called *calfile*. Note that to ensure that calendars for all three years are directed into the file, it is necessary to group the commands using parentheses.

```
>(cal 1980; cal 1982; cal 1983) > calfile & CR
[3] 2056
```

Note that when a command line includes a number of commands connected by pipes, the shell returns a single job number, but a number of process ID numbers, for example

```
>ls -l | sort > listfile & CR
[4] 2057  2058

>
```

In this case, this command line generates Job 4, comprising processes 2057 (the **ls -l** command) and 2058 (the **sort** operation).

Where problems arise, and a background job cannot be terminated correctly, the shell will display a message saying that the job has been killed. This will also occur just before the appearence of the next system prompt.

Note that a background job cannot read input from the terminal. Unless the job and its data source are self-contained, the shell will suspend the job. The input can be supplied by running the command in the foreground. If necessary, the job can be transferred back to the background until a further data request is made.

The next section explains how jobs may be controlled by the user.

Terminating Commands

Occasionally, it may be necessary to stop a command before it has finished executing. This may be because it is not doing what was originally intended, or because it has already yielded the required information. The shell provides a number of methods of handling foreground jobs in this way. However, in order to terminate a background job, it is necessary to first bring the command into the foreground using **fg**, and then to issue the appropriate signal.

The Interrupt Signal

The *interrupt* signal is obtained by pressing the keys marked **CTRL** and **c** together. Some commands like **cat** and **ls** are not designed to handle interrupts in any specific way, and accordingly, they terminate. The shell detects the interrupt and displays another system prompt. On this basis, hitting **CTRL-c** again would have the effect of terminating the shell program, and logging out the user; however, **csh** is designed to ignore interrupts. It responds by merely displaying another system prompt.

The Quit Signal

A further way of stopping a job is to use the *quit* signal. This is most useful when running new programs that may still contain bugs. This signal has the effect of stopping a job ungracefully but surely. It is obtained by typing **CTRL-**. The following example shows the effect of using the Quit signal to terminate a program called *testprog*.

```
>testprog CR
CTRL-\
Quit (Core dumped)
```

This signal tells the system to terminate the program, and create a *core dump*, which contains data useful in interpreting the behaviour of the program up to the time the signal was received.

Job Control Commands

X/OS supplies a range of commands for controlling jobs. The commands **jobs**, **fg** and **stop** are actually built-in shell functions, while **kill** is an independent X/OS utility. Because it can be used when the shell's own job termination commands fail, it is mentioned here.

The jobs Command

The **jobs** command will give a list of the currently active background jobs. A typical listing is as follows:

```
>jobs CR
[1]  - Running        mail sue
[2]    Running        pack *.txt
[3]  + Running        mail basil

>
```

Background jobs are flagged with the message *Running*. Any job listed using the **jobs** command can be brought into the foreground with the **fg** command. Note that the + and - signs are used to indicate the priority of the jobs. Used without arguments, **fg** will reactivate the *current* job (marked +) first, and the job marked - second. In the above example, job 2 would be affected last.

The fg Command

Any background job can be brought into the foreground with the **fg** command. Once a list has been displayed using **jobs**, the job numbers displayed can be used as arguments to **fg**. Using the jobs list shown above, the following example would activate job 2. Typing **fg** without arguments would have activated job 3 instead.

```
>fg %2 CR
pack *.txt
```

The **%** metacharacter is used for all the job control commands. It can be followed by a job number, a hyphen to indicate the previous job, a unique prefix of one of the command lines, or a question mark (**?**) followed by a string found in only one of the jobs. The following examples would all reactivate job 2:

```
>fg %2 CR
pack *.txt

>fg %pack CR
pack *.txt

>fg %?pa CR
pack *.txt

>
```

The kill Command

The **kill** command is used to terminate a background job immediately. Like the other job control systems, it may be given arguments consisting of either one of the normal job identifiers, or a process ID. These may be obtained

using the **ps** command. Note that **kill** is the name of both an in-built C Shell function and a general X/OS command.

OUTPUT CONTROL

The shell supplies a number of facilities for controlling what happens to the output from a command. The first is the set of re-direction metacharacters. These have already been described, above. The second system consists of the screen control signals. The third consists of a variety of output formatting commands. These latter two facilities are described below.

The Screen Control Signals

Many commands are capable of producing more than a single screenful of output at a time. If this is to be read, the shell must be told to print only a single screenful, then pause. For example, running **cat** on a long text file will cause most of the text to zoom off the top of the screen before it can be read. The first way of dealing with this is to type **CTRL-s**. This stops the text as soon as the signal is received. To start the output going again, **CTRL-q** is typed. These two can be used as often as needed.

The pg Command

An easy way of controlling output is to use the **pg** command. In its simplest form, it will accept the name of one or more files as arguments, and perform a **cat**-type operation. This is actually a general X/OS comand, but it is very useful, so it is mentioned here. The *User Guide* supplies a full **pg** tutorial. The following example displays the contents of a file called *textfile.txt*, 20 lines at a time. To see the next screen of text, the **CR** key is pressed.

```
>pg -20 textfile.txt CR
What is archaeomagnetism?
The study of thermo-resident magnetism in fired clay artifacts
and other objects originally subjected to high temperatures,
.
.
.
Dating is based on matching archaeomagnetic phenomena against
known changes in the Earth's magnetic field.
:
```

THE C SHELL VARIABLES

Predefined and Environment Variables

The shell maintains a series of variables which are assigned values using the **set** command, which typically takes the form

set *name* = *value*

although in some cases, it is not necessary to assign a value. **Set** used without arguments displays the current values of the shell variables, which are stored in a range of files held in the user's home directory. A typical list of the variables obtained using **set** in this way is as follows:

```
>set CR
argv     ()
cwd      /usr/spike
home     /usr/spike
path     (. /bin /usr/local/bin /usr/spike/bin)
prompt   >
```

```
shell    /bin/csh
status   0
term     vt100
user     spike

>
```

All of these except the last two are set by the shell itself, and with the exception of **cwd** and **status**, all are set at initialisation. This means that if the user changes any of the values of these variables, the new values will be used by the shell only after the user has logged off, and started a new session. These is, however, a way of forcing the shell to read the new values during the current session. This is described below.

From the list, it can be seen that Spike's home directory is */usr/spike*, (the **home** variable), and that he is using the C Shell as opposed to the Bourne Shell. The **cwd** variable points to the current working directory. This changes as the user moves around the directory system. The **path** variable contains a list of the directories that will be searched by the shell in order to find the executable files containing the code for commands entered by the user. The search path begins with the current directory, indicated by its normal metacharacter (.). The **prompt** variable defines the sequence of characters used as the system prompt, in this case, the greater-than symbol (**>**). The **term** variable indicates the type of terminal in use. The codes used to define terminal types are listed in the *term*(5) entry of the *System Interfaces and Libraries Reference Manual*. Use of **argv** and **status** are explained below, in the section entitled *Programming the Shell*.

The current values of these (and other) variables are stored in a number of files held in the user's home directory. These begin with a dot, for example *.login*, and accordingly, can be listed using the **-a** option of the **ls** command.

```
>ls -a ~ CR
.
..
.cshrc
.history
.login
.logout

>
```

Remember that the tilde character (~) points to the user's home directory.

The rest of this section describes the contents and roles of these files, during which, some of the variables are described. At the end of the section, a complete list of the C Shell's variables is given.

These dot files will be present irrespective of the shell in use. The exception to this is the file called *.cshrc*. This file is present only if the user has access to the C Shell.

.cshrc

If the user has access to the C Shell, this file is read by **csh** immediately on log in. It is used to set the C Shell variables, which may or may not be different to those set for the Bourne Shell. Note that X/OS supplies both shells, so some variables will be duplicated. The *.cshrc* file contains the C Shell's **path** variable, and any **alias** names that have been assigned. The **alias** system is described below. Also set are the **history** and **savehist** variables. These two will not be found elsewhere, because the **history** mechanism is peculiar to the C Shell. **History** is also described below. Because the C Shell's prompt may be different from the Bourne Shell's, the **prompt** variable can also be set here.

```
>cat .cshrc CR
alias h history
alias dir 'ls -l | pg -20'
set history=40
set savehist=20
set path=(. /bin /usr/local/bin /usr/spike/bin)
set prompt="[>\!]"
umask 0022

>
```

According to the above list, the user has assigned two aliases: this allows him to type **dir** rather than the longer **ls -l | pg -20**, and **h** instead of **history**. The **history** mechanism is set to memorise the last 40 commands entered, and after logging out, the last 20 of these will be stored in the *.history* file, for use in the next session. The **path** variable tells the shell where to look for executable command files whenever a command is entered, and the prompt is set to a *greater than* sign, followed by the current command number. The **umask** entry sets the file creation mode, and has its own tutorial in the *User Guide*.

Any of these variables that may be set elsewhere are over-ridden by the values set in *.cshrc*, when the C Shell is in use.

.login

This file is read by the shell whenever the user logs into the X/OS system. If *.cshrc* is present, *.login* is read immediately after the C Shell variables have been read. It contains commands that are to be executed each time the user starts a session.

The following example displays the contents of the *.login* file, with some variables set.

```
>cat .login CR
set noclobber
set ignoreeof
setenv SHELL /bin/csh
setenv TERM vt100
set home=/usr2/spike
cd /usr2/spike/project3
.
.
.

>
```

The **ignoreeof** variable is here used to tell **csh** not to log off the user whenever **CTRL-d** is pressed. In this way, the user must explicitly use the **logout** command. Note that **ignoreeof** does not take a specific value. Another such flag type variable is **noclobber** which tells the shell not to accidentally overwrite existing files when the **>** re-direction indicator is used. If **noclobber** is set, a warning message is printed instead of automatically *clobbering* the existing contents of the file. Note that it is possible to force the over-writing of a file when **noclobber** is set by using the metanotation **>!** instead of **>**.

The **setenv** entries set the environment variables **SHELL** and **TERM**. For further details of **setenv**, see the *csh*(1) entry in the *Utilities Reference Manual*.

The user's home directory is set to */usr2/spike*, and the **cd** command automatically makes *project3* the current working directory at the beginning of each session.

Note that this file can also be used to set any terminal options that may be needed. A full list of the optional settings is given in the *stty*(1) entry of the *Utilities Reference Manual*.

.logout

This file contains commands that are to be carried out after the shell has logged out the user, for example:

```
>cat .logout CR
clear

>
```

The **clear** command tells X/OS to clear the screen before displaying the new login prompt.

.history

This file contains a list of previously executed commands. The number of commands memorised depends on the setting of the **savehist** variable stored in the *.cshrc* file.

```
>cat .cshrc CR
.
.
.
set history=40
set savehist=20

>cat .history CR
cd ~
dir
ls -l text.*
rm text.bak
cat para1 para2 para3 para4 > chap1
rm chap2
h
cd ../manual3
dir
```

```
cat ~/.logout | pg -20
.
.
.
```

Changing Variable Values

The values of the shell variables within these special files can be altered. An example is to change the value of the **prompt** variable in the *.cshrc* file:

```
set prompt="[OH NO, NOT AGAIN\!]"
```

This command line will set the system prompt to read *OH NO, NOT AGAIN*, followed by the current command number. This will appear instead of the greater-than symbol that has been used throughout this document, for example:

```
[OH NO, NOT AGAIN28]ls -aC ~ CR
.          .cshrc      .history    .login
..         .logout     .profile

>
```

Another example is to add a new command to the system, for example, a shell script (see below). In order for the shell to find the new command file, is necessary to ensure that it is contained in one of the directories named by the **path** variable. This would be done by amending the value of **path**. Note that after the command has been added, and **path** up-dated, the system may not find it immediately because the search mechanism relies on data collated at login. Accordingly, it may be necessary to log out, then log back in. A quicker way is to issue the command **rehash**. This forces the shell to

re-read the **path** variable.

The shell can also be forced into reading the other variables stored in one or more of the special files. This is done with the **source** command, which accpets as an argument the name of the file containing the particular variable to be read, for example:

```
>source .cshrc CR

>
```

This command line would force the shell to re-read all the variables relating to the C Shell.

THE HISTORY MECHANISM

The shell maintains a *history list* of the last commands entered by the user. The size of the list, that is, the number of past commands that can be remembered, is set using the **history** variable which is stored in the *.cshrc* file.

The history list can be used to repeat past commands, to edit out typing mistakes, or to replace parameters. Accessing the history list is done by typing the **history** command. The output consists of a command number followed by the command that was typed in. Note that even commands that failed to execute are included (for example, see command 4, below.)

```
>history CR
 1  ls -aC
 2  ps -a > status.doc
 3  cat status.doc
 4  hstory
 5  history
```

Among the metanotations available to access these entries are the following:

!n accesses the command line with number *n*.

!! accesses the last entry in the list.

!x accesses the last command beginning with the string *x*. This string must consist of as many letters as are needed to uniquely point to the required command.

The **history** system also provides a range of metacharacters for editing existing list entries. These are explained in full in the *csh*(1) entry in the *Utilities Reference Manual*.

In the following examples, a history list of five command lines is used. Note that the fourth was entered incorrectly.

```
>!! CR
history

 1  ls -aC
 2  ps -a > status.doc
 3  cat textfile.txt
 4  hstory
 5  history

>!3 | pg -20 CR
cat textfile.txt
What is archaeomagnetism?
The study of thermo-resident magnetism in fired clay artifacts
and other objects originally subjected to high temperatures,
.
.
.
```

```
Dating is based on matching archaeomagnetic phenomena against
known changes in the Earth's magnetic field.
:

>!p CR
ps -a > status.doc

>!4:s/hs/his/ CR
history

 1  ls -aC
 2  ps -a > status.doc
 3  cat textfile.txt
 4  hstory
 .
 .
 .
```

Note that the contents of file *textfile.txt* were displayed using the history notation in conjunction with a command entered in the usual way. Because the shell expands the history notation before a command line is executed, it can be combined with other shell commands as if it were typed in full.

The last command used the editing system to correct the spelling mistake in command line 4. The initial **:s** is one of the shell's *modifiers* (described below), and stands for *substitute*. The text between the slashes are respectively the original text and its desired replacement. Note that this substitution does not alter the contents of the **history** list, but merely the command line to be executed.

ALIASES

The shell supports an *alias* system, whereby complicated commands that are used frequently can be given a simpler name. In the example, a couple of **alias** declaractions have been inserted in the *.cshrc* file, as follows:

```
>cat .cshrc CR
alias  dir  'ls -l | pg -20'
alias  h    history
.
.
.

>h CR

 4  history
 5  history
 6  history
 7  cat .cshrc
 8  ps -a > status.doc
 9  history
10  cat .cshrc
11  h

>
```

Each time an alias name is typed, the full command that it represents will be run. The *h* alias entered above had the effect of running the **history** command. Notice that the **ls -l | pg -20** command sequence was enclosed in quotes. This was done in order to group the elements together, but also has the effect of screening any special characters from being interpreted by the shell as being metacharcters.

Because these aliases have been set in the *.cshrc* file, they are read by the shell whenever the user logs into the system or issues the **source** command. Note that there

is no real limit to the number of aliases that can be used, but maintaining a large number in *.cshrc* will cause the shell to start up slowly.

The shell also supplies an **alias** command, which in its simplest form, returns all the current aliases. When entered with an argument, it prints the alias of that string.

```
>alias dir CR
ls -l | pg -20

>alias CR
dir  ls -l | pg -20
h    history

>
```

THE DIRECTORY STACK

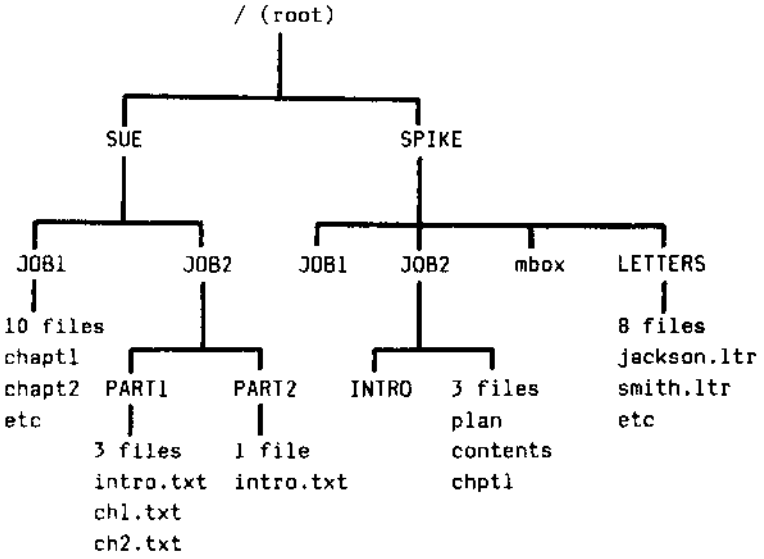
The directory system is created using the command **mkdir** (make directory), and users can move around the resulting system using **chdir**, optionally shortened to **cd** (change directory). Empty directories can be deleted with **rmdir** (remove directory), and the current location can be printed using **pwd** (print working directory). Some of these commands have been introduced already.

The shell keeps track of the directories being used. The **cwd** shell variable always stores the pathname of the current directory, and previously used directories can be remembered using the *directory stack* system. To enter directory names into the stack, the **pushd** command is used instead of **cd**. If used with a pathname as an argument, the named directory is pushed onto the top of the directory stack. Because the top entry in the stack is the current directory, the new directory becomes the current working directory.

Used without an argument, **pushd** swaps the top two entries, and what was the second entry becomes the working directory. A further use of **pushd** will swap them back again. Typing **pushd** *+n* will swap the *n*'th entry in the stack to the top.

To remove a directory from the stack, the **popd** command is used. Used without a directory name as argument, **popd** removes the top entry. Used in the form **popd** *+n*, the *n*'th entry in the stack is discarded.

The following sequence of commands uses this example directory system as its basis, with files named in lower case, and directories in upper case:



The user has just logged on, and needs to check on the location and contents of a number of files. The first command adds the directory */spike/letters* to the stack. Note that **pushd** responds with the current entries in the stack. The user's home directory is always the first

entry, so that the first **pushd** command pushes it down the stack into second place. Note also that the command line uses the tilde (~) metanotation to indicate the pathname to the home directory of Spike. Any directory levels between *root* and */spike/letters* will be added automatically. Note that the user's home directory is always present in the directory stack unless explicitly removed using **popd**.

```
>pushd ~spike/letters CR
~spike/letters ~

>pushd ~sue/job2/part1 CR
~sue/job2/part1 ~spike/letters ~

>ls -C CR
intro.txt   ch1.txt        ch2.txt

>dirs CR
~sue/job2/part1 ~spike/letters ~

>pushd CR
~spike/letters ~sue/job2/part1 ~

>ls -C CR
jackson.ltr   smith.ltr       browne.ltr     jones.ltr
sales.ltr     addresses       network.ltr    mrktng.ltr

>popd CR
~sue/job2/part1 ~

>
```

The second command line pushed another directory onto the stack. The following **ls** command listed the files contained by this second directory. The next command was **dirs** which stands for *directory stack*, and which prints out the current contents of the stack without changing the order. The next **pushd** command, used without

arguments, swapped the first two entries so that the following **ls** accessed the letters directory. Finally, **popd** was used to flush *~spike/letters* out of the stack.

This section ends with a quick reference to the pre-defined and environment variables available. Some of them have already been covered in some detail. More detail is often available in the *csh*(1) entry of the *Utilities Reference manual*.

argv used to handle positional parameters and shell arguments. See the section on shell programming for more details.

cdpath gives a list of alternative directories searched to find sub-directories, when using the shell's **chdir** command. Where the specified directory name is not found in the current directory, the directories specified by the **cdpath** variable are checked.

cwd the full pathname of the current directory.

histchars resets the characters used in the **history** metanotation. The first value given replaces the ! character.

history specifies the size of the history list.

home the home directory of the user.

ignoreeof tells the shell to ignore **CTRL-d** signals.

mail specifies a list of files to be checked for mail. The shell displays a message whenever new mail is received in one or more of the specified locations.

noclobber places restrictions on the shell's ability to over-write existing files when re-direction operators are used.

noglob places restrictions on the shell's ability to execute filename expansion.

nonomatch inhibits the shell's returning of error messages where a filename expansion does not match an existing filename.

notify tells the shell to inform the user about completed jobs immediately after termination of the job, rather than waiting for the next system prompt to be printed.

path specifies the directories to be searched for command files. If new commands are added to the system during a session, the **rehash** command will force the shell to re-read the **path** variable.

prompt specifies the character string to be used for the system prompt. The shriek character (!) indicates that the current history list number will be printed.

savehist specifies the number of entries from the history list to be saved in the *.history* file after logout.

shell specifies the file that stores the current shell.

status stores the status returned by the last command.

time specifies that a warning be displayed whenever a command is entered that uses more than a specified number of CPU seconds.

verbose causes the expanded version of a command to be printed following a history substitution.

Note that a more detailed version of this list can be found in the *csh*(1) entry in the *Utilities Reference Manual*.

THE SECURITY SYSTEM

The shell maintains a system that protects files from unwanted interference from other users, accidental or intentional. Throughout this document, there have been times when the **ls -l** command has been used to give a *long* listing of the files held in a directory. The output from this command for a file called *project* would be something like the following:

```
>ls -l CR
-rwxr--r--  1 spike  GRP2  56731  July 21 12:07   project

>
```

This output line states that *project* has a single link, is owned by Spike and group GRP2, consists of 56731 bytes, and was created or last modified at 12:07 on July 21. The first field of 10 characters defines the *access permissions* of file *project*.

The first hyphen indicates that this is a normal file, not a directory. Directories have a letter *d* at this position. The following nine positions define the read (*r*), write (*w*) and execute (*x*) permissions for various classes of user. There are three user classes. The first set of three positions defines the *user's* access permissions. The user is often called the *owner* of the file. In this case, the user that created the directory has the right to read, write and execute its contents. The next set of three positions defines the *group's* permissions. A user group is typically set up by the system administator so that users working on the same project can be supervised together.

The third set of three positions states the permissions of users other than the owner and members of the owner's group.

These permissions can be set using the **chmod** command, which uses the following codes to identify the range of users affected:

u the owner of the file (user)

g the user's group

o other users

a all users, that is **ugo**

To give the user's group permission to execute *project*, the user would type

```
>chmod g+x project CR

>ls -l CR
-rwxrx-r--  1 spike  GRP2  56731  July 21 17:07   project

>
```

Note that the **ls -l** command confirmed that the user's group now has permission to execute this file. **Chmod** can be used to set any combination of permissions. The following indicators are used:

+ adds a permission

- removes a permission

= assigns an absolute permission

In the first of the following two examples, read permission is removed from the user's group, while the second sets read, write and execute permission for all users.

```
>chmod g-r project CR

>chmod a=rwx project CR

>
```

An *absolute* method of using **chmod** is explained in the *chmod*(1) entry of the *Utilities Reference Manual*.

MORE BUILT-IN COMMANDS

The C Shell supplies a number of other useful built-in commands not yet covered. The first of these is actually one that has been mentioned before, but which has a further useful function.

The prompt Command

The **prompt** command has already been used to set the set of characters to be used as a system prompt. However, it can be used in conjunction with the **history** mechanism to keep a running total of the number of commands entered during a session.

The shriek character (**!**) can be used to number each command line as it is entered. The following example command line sets the system prompt to **[HELLO:** *nn***]**, where *nn* is the number of the current command line in the **history** list:

```
set prompt='[HELLO: \!]'
```

Note that the shriek must be escaped using the backslash (\), despite the quote marks. The system prompt will thereafter have the form

```
[HELLO: 28]
```

where the current command line is the 28th of the session.

The repeat Command

This useful shell command allows the repetition of any standard shell command. It accepts as an argument, the number of times the command is to be repeated. For example, to append ten copies of a file called *boring* to the end of a file called *dump*, the following command could be used

```
>repeat 10 cat boring >> dump CR
>
```

The time Command

The **time** command can be used to check on how much computer time is being used by the commands entered by the user. It accepts as an argument any standard shell command line. The following example illustrates **time** in use to check on a simple **ls** operation.

```
>time ls -l CR
-rwxr-xr-x   1 spike  GRP2   4876  Jul 21 14:20   chap1.txt
-rwxr-xr-x   1 spike  GRP2  23851  Jul 21 16:05   chap2.txt
0.1u 0.1s 0:01 15% 25+43k 3+1io 1pf+0w

>
```

After the output from the **ls** operation appears a single line giving the statistics generated by **time**. It indicates that **ls** used up 0.1 seconds of user time (**u**), 0.1 seconds of system time (**s**), and 1 second (**0:01**) of real time, and that while the **ls** program was active, it used 15% of the machines available CPU cycles. It also indicates that the command used an average of 25 kbytes of program space and 43 kbytes of data space. Three disk read operations and one disk write operations were performed, ad the operation took one page fault, and was not swapped.

Unsetting Aliases and Variable Definitions

Once set, aliases and variable definitions can be freed using the **unalias** and **unset** commands respectively. Environment variables can be freed using **unsetenv**.

PROGRAMMING THE C SHELL

Introduction

In addition to being a command interpreter, the shell also acts as a programming language. The user can create files containing sequences of the shell's programming commands. These files are called *shell scripts*, and are used to invoke shell operations under program control.

Invoking a Script and Using the argv Variable

A **csh** command script file can be activated by typing

```
>csh script args CR
```

where *script* is the name of the shell script file, and *args* is a sequence of arguments. The values of these arguments are placed in the variable *argv* which indexes the values in the form *argv*[*n*]. The variable *n* is the index. As the shell script is executed, **csh** accesses the arguments as they are required by reading *argv*.

The values to be entered into an argument are set using the notation

```
set name = (x y z) CR
```

where *name* identifies the variable, and *x*, *y* and *z* are its actual values.

A number of metanotations are available for checking the contents and status of variables. The first example sets the values of *name* to **a**, **b** and **c**. The **echo** command is used to print out the current values of *name*. The **$** character is always used to indicate that the following word is a variable name.

```
>set name = (a b c) CR

>echo $name CR
a b c

>echo $?name CR
1

>echo $name[2] CR
b

>echo $name[1-2] CR
a b

>echo $1 CR
a

>
```

The third command line in the sequence used the **?** character to enquire whether the values of *name* had been set. A **1** is returned if *name* has been set, a **0** if not.

The fourth command in the sequence used the index notation to access the value of a particular component of a multiple value variable. The second value in the set (**b**) is returned.

The fifth command returns the values of entries 1 and 2, while the last command acts as shorthand for **$name[1]**.

The hash character (**#**) is used to indicate the number of values available to *name*:

```
>echo $#name CR
3

>echo $name[$#name] CR
```

```
c

>
```

The second command line used a combination of notations. The index element inside square brackets is expanded to return the number 3 (the number of values set for *name*). This then acts as the index to the **echo $name** command. This notation is useful when the number of values set is not known, and the value of the last one is of interest.

Values can be removed using the **unset** command:

```
>unset name CR

>echo $?name CR
0

>echo $name CR
Undefined variable: name

>
```

Note that once *name* has been unset, **?** returns **0**, and that an attempt to display the contents of *name* produces an error message.

It is also possible to establish a variable, and then to read a previously unknown value into it from the keyboard. This is useful when writing an interactive script. It is done using the notation **$<**. For example, the following sequence will prompt the user to enter a name, then read the name into variable *id*.

```
echo 'Please enter your name\c'
set id = ($<)
```

Expressions

The full range of expressions available to the C programming language are available for use in **csh** shell scripts. These are described in more detail in the *csh*(1) reference of the *Utilities Reference Manual*. The == (equal to) and != (not equal to) expressions are used to compare strings, and the **&&** and **||** expressions respectively implement the Boolean AND and OR operations. Special expressions using the tilde character (~) are also available. These are =~ and !~, which are used to match strings in the same way as == and !=, except that the right hand string may include the pattern matching metacharacters (*, ? and []).

The shell also supplies a number of file enquiry expressions. These take the form *-? filename* where *filename* identifies the file, and may include a pathname or may point to a number of files using variables or indices. The ? is a single letter, for example:

r read access

w write access

x execute access

e existence

o ownership

z zero size

f plain file

d directory

For example, the expression **-e scriptl** will check whether the file *scriptl* exists, while **-r scriptl** checks whether the user has read access to the file.

It may also be useful to determine how a command has terminated. It can be the case that a shell script will need to take account of whether a command terminated successfully. The shell supplies two ways of checking this termination status of a command. The first involves using a primitive in the form { *command* }. If *command* terminates normally, the primitive returns **1**. If *command* fails, the primitive returns **0**.

Another way to check on command termination is to use the *$status* variable. A *$status* value is returned by every command ever executed by the shell, and this can be checked in the line following the command. Note that since *$status* is updated for every command, it is highly transient, and should be checked immediately after the command is executed.

For a full list of the expressions available to the shell, see the *csh*(1) entry of the *Utilities Reference Manual*.

Control Structures

The shell provides a range of statements that control the flow of execution of a shell script. These will be familiar in outline to any user who has already used a programming language. They are explained in full in the *csh*(1) entry of the *Utilities Reference Manual*.

The most commonly used are

if *expr command*

> If the expression (*expr*) is true, *command* is executed.

if *expr1* **then**
commands
else if *expr2* **then**
commands

else
commands
endif

If the expression (*expr1*) is true, the first set of commands are executed. However, if *expr* is false and *expr2* is true, the second set of commands are executed instead. If *expr1* and *expr2* are both false, the third set of commands are executed. The expressions should be exclusive. Note that any number of **else...if** pairs are allowed, but only one **endif** is needed.

while (*expr*)
commands
end

While *expr* is true, *commands* are carried out.

foreach *name (wordlist)*
commands
end

The variable *name* is successively set to each value found in *wordlist*, and *commands* are executed for each value.

goto *label*

This is an unconditional jump to another point in the program, *label*. Execution continues from the point identified by *label*.

switch (*string*)
:case
commands
breaksw
...
default
commands
breaksw

endsw

Each example of *case* is matched against *string*, and the *commands* following the matching *case* are carried out. If no match occurs, the *commands* following the default condition are carried out. Each set of commands should be followed by a **breaksw** statement. The construction should end with a **endsw** statement.

These control statements are often subject to a range of conditions, for example, many of those printed in boldface above, should appear at the start of a line. Most forms of loop (the **foreach** ... **end** statement for example) can be continued indefinately using the shell's built-in **continue** command, or broken prematurely using **break**. Full details of these conditions, and use of the **continue** and **break** commands are given in the *csh*(1) entry of the *Utilities Reference Manual*.

The Modifiers

The shell supplies a number of devices that allow the modification of words in a command line, including lines in a shell script. These typically take the form of a colon (:) followed by a single letter (with the exception of the substitution modifier, which requires the definition of *before* and *after* strings). Some of the available modifiers are:

h removes trailing pathname elements from a filename, leaving only the first element in the path.

r removes trailing filename extensions.

e removes whole filenames, leaving only the extension.

t	removes leading pathname elements from a filename, leaving only the last element in the path.
s/x/y/	substitutes string *x* for *y*.
&	repeat the previous substitution.
g	applies a modification globally. The **g** precedes the modifier, for example **g&**.
p	prints the new version of the command, but does not execute it.
q	quotes the substituted words, preventing further substitutions.

The following example assumes the variable *i* has the value */spike/textfile.txt*:

```
>echo $i $i:r $i:e $i:s/text/test/ CR
/spike/textfile.txt  /spike/textfile  txt  /spike/testfile.txt

>
```

Note that only one modifier should be applied to each **$** substitution.

A Sample C Shell Script

The following script uses the control structures, the expression mechanism, the modifiers, and the variable substitution system. The file is called *scriptl*. It contains comments, following the # characters. The program code is in bold face.

```
#
#   Scriptl compares a series of C program files against copies
#   held in a directory called ~/Cbackup. If they are different,
#   the new version of the program file is copied into ~/Cbackup.
#
set noglob
foreach i ($argv)                  #  Variable i successively points
                                   #  to each file in the list.

   if ($i !~ *.c) continue         #  Not a C file so do nothing.
                                   #  Note use of !~ to allow file
                                   #  name expansion notation.

   if (! -r ~/Cbackup/$i:t) then
   #  Note use of :t modifier to strip off pathnames.
         echo $i:t Not in backup ... Not yet copied
         continue
   endif

   cmp -s $i ~/Cbackup/$i:t        #  Use of cmp command sets
                                      #  variable $status

   if ($status !=0) then           #  Reads variable $status
         echo new backup of $i
         cp $i ~/Cbackup/$i:t      #  Copies file.
   endif

end
```

The script begins by setting **noglob** which prevents accidental expansion of the filenames should any of the

arguments contain filename expansion metacharacters.

The second line instructs the shell to execute the commands between the **foreach** and **end** statements, for each value of *i*, where *i* is set to each successive file in the directory. Files not having the extension *.c* are filtered out of the operation.

The *.c* files are run through the **cmp** file comparison utility. When used with its **-s** option, **cmp** checks for differences and returns a status code rather than printing a message. This code is passed to the variable *status*. **Cmp** returns code **0** to indicate identical files; if a value other than **0** is stored in *status*, the file in the current directory is obviously different from its supposed backup in *~/Cbackup*, and a new copy must be made. The copy operation consists of printing a message, then running the **cp** command.

Executing a C Shell Script

Shell scripts written under the standard **sh** shell will not necessarily run under **csh**. However, they can be translated into **csh** format with a fair degree of certainty by adding a hash character (**#**) to the beginning of the script, as a separate line. If this does not appear, X/OS will use **sh** to execute the script. Note that because the file *script1*, above, began with a comment field, it can be executed by **csh** rather than **sh**.

Note that after they have been written, shell scripts must be made executable using the **chmod** command, explained above. The following will give execute permission to the owner of *script1*:

```
>chmod u+x script1 CR

>
```

As soon as the correct permissions have been made available, the user can execute the shell script. The shell breaks each line in the script into individual words, and performs any history substitutions that may be required. The input lines are then parsed into distinct commands. The final stage is that of *variable substitution* in which the variables are replaced with their actual values.

D

E

F

G

Q

R

S

Printed in Italy

olivetti