LSX Computer Line

Operating Systems

X/OS UNIX® System V-based Operating System

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

X/OS UNIX® System V-based Operating System

User Guide

olivetti

PREFACE

This manual is a introductory user guide to the LSX X/OS operating system. It sets out to describe the most commonly used utilities. At the same time, some of the background topics are covered, such as the X/OS file and directory system, and use of the error message system.

SUMMARY

The manual begins with a brief Introduction which describes the conventions used in the tutorial chapters. The main body of the manual comprises 10 chapters, each of which contains tutorial-style coverage of one or more of the basic X/OS utilities. These are distributed as follows:

1. Introduction

2. File and Directory Commands:
   CAT, CD, CHMOD, CP, LS, MKDIR, MV, PWD, RM, RMDIR

3. File Handling Commands:
   COMM, CUT, DIFF, DIFF3, FILE, NL, PASTE, PG, PR, SORT, SPELL, SPLIT, TAIL, TEE, TEST, TR, UNIQ, WC

4. Data Display Commands:
   BANNER, CAL, CALENDAR, DATE, ECHO

5. Editors:
   ED, VI

6. File Storage Commands:
   AR, PACK, PCAT, UNPACK

7. Pattern Editors:
   AWK, GREP, SED

8. System Status Commands:
   DF, DU, SUM

9. Mail:
   MAIL

10. Process Handling Commands:
    KILL, NOHUP, PS, UMASK

11. Programming Support Commands:
    M4

REFERENCES

Read first ...

X/OS Operating Guide - Code 4055390 Y

For further information, read ...

Shell / C Shell X/OS Command Language User Guide (in this binder)

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

1. INTRODUCTION

1-1 GENERAL

1-1 MANUAL CONVENTIONS

2. THE FILE AND DIRECTORY SYSTEM

2-1 INTRODUCTION

2-2 FILES AND DIRECTORIES

2-4 STANDARD INPUT AND STANDARD OUTPUT

2-5 CAT: file printing and concatenation

2-5 INTRODUCTION

2-5 SYNTAX

2-5 DESCRIPTION

2-6 EXAMPLES

2-9 CD: change working directory

2-9 INTRODUCTION

2-9 SYNTAX

2-9 DESCRIPTION

2-10 EXAMPLES

2-12 CHMOD: file protection

2-12 INTRODUCTION

2-12 SYNTAX

2-12 DESCRIPTION

2-14 EXAMPLES

2-17 **CP: file copying**

2-17 INTRODUCTION

2-17 SYNTAX

2-17 DESCRIPTION

2-18 EXAMPLES

2-21 **LS: file and directory listing**

2-21 INTRODUCTION

2-21 SYNTAX

2-21 DESCRIPTION

2-23 EXAMPLES

2-26 FREQUENTLY USED LS OPTIONS

2-26 LISTING ALL FILES IN A DIRECTORY

2-27 LISTING FILES IN SHORT FORMAT

2-27 LISTING FILES IN LONG FORMAT

2-29 **MKDIR: create a directory**

2-29 INTRODUCTION

2-29 SYNTAX

2-29 DESCRIPTION

2-30 EXAMPLES

2-32 **MV: moving and renaming a file**

2-32 INTRODUCTION

2-32 SYNTAX

2-32 DESCRIPTION

2-33 EXAMPLES

2-36 **PWD: print working directory**

2-36 INTRODUCTION

2-36 SYNTAX

2-36 DESCRIPTION

2-38 PATH NAMES

2-38 FULL PATH NAMES

2-39 RELATIVE PATH NAMES

2-41 EXAMPLES

2-42 **RM: file deletion**

2-42 INTRODUCTION

2-42 SYNTAX

2-42 DESCRIPTION

2-43 EXAMPLES

2-45 **RMDIR: directory deletion**

2-45 INTRODUCTION

2-45 SYNTAX

2-45 DESCRIPTION

2-46 EXAMPLES

3. THE FILE HANDLING COMMANDS

3-1 INTRODUCTION

3-3 COMM: select or reject common lines

3-3 INTRODUCTION

3-3 SYNTAX

3-4 DESCRIPTION

3-4 EXAMPLES

3-7 CUT: print selected fields from a file

3-7 INTRODUCTION

3-7 SYNTAX

3-7 DESCRIPTION

3-8 EXAMPLES

3-11 DIFF: file difference identifier

3-11 INTRODUCTION

3-11 SYNTAX

3-11 DESCRIPTION

3-12 EXAMPLES

3-14 **DIFF3: three way differential file comparison**

3-14 INTRODUCTION

3-15 SYNTAX

3-15 DESCRIPTION

3-15 EXAMPLES

3-18 **FILE: determines file type**

3-18 INTRODUCTION

3-19 SYNTAX

3-19 DESCRIPTION

3-19 EXAMPLES

3-21 **NL: line numbering filter**

3-21 INTRODUCTION

3-21 SYNTAX

3-22 DESCRIPTION

3-24 EXAMPLES

3-25 **PASTE: file merge utility**

3-25 INTRODUCTION

3-25 SYNTAX

3-25 DESCRIPTION

3-26 EXAMPLES

3-28 **PG: paging through the contents of a file**

3-28 INTRODUCTION

3-28 SYNTAX

3-28 DESCRIPTION

3-32 EXAMPLES

3-35 **PR: print files**

3-35 INTRODUCTION

3-35 SYNTAX

3-35 DESCRIPTION

3-36 EXAMPLES

3-39 **SORT: sorting and merging files**

3-39 INTRODUCTION

3-39 SYNTAX

3-39 DESCRIPTION

3-42 EXAMPLES

3-46 **SPELL: the spelling check utilities**

3-46 INTRODUCTION

3-46 SYNTAX

3-46 DESCRIPTION

3-48 EXAMPLES

3-54 **SPLIT: splits a file**

3-54 INTRODUCTION

3-54 SYNTAX

3-54 DESCRIPTION

3-55 EXAMPLES

3-56 TAIL: print the last part of a file

3-56 INTRODUCTION

3-56 SYNTAX

3-56 DESCRIPTION

3-57 EXAMPLES

3-59 TEE: pipe fitting utility

3-59 INTRODUCTION

3-59 SYNTAX

3-59 DESCRIPTION

3-60 EXAMPLES

3-62 TEST: expression evaluation utility

3-62 INTRODUCTION

3-62 SYNTAX

3-62 DESCRIPTION

3-65 EXAMPLES

3-67 TR: translates characters

3-67 INTRODUCTION

3-68 SYNTAX

3-68 DESCRIPTION

3-69 EXAMPLES

3-72 **UNIQ: searches for repeated lines**

3-72 INTRODUCTION

3-72 SYNTAX

3-72 DESCRIPTION

3-73 EXAMPLES

3-76 **WC: count lines, words, and characters**

3-76 INTRODUCTION

3-76 SYNTAX

3-76 DESCRIPTION

3-77 EXAMPLES

4. **THE DATA DISPLAY COMMANDS**

4-1 **INTRODUCTION**

4-2 **BANNER: poster utility**

4-2 INTRODUCTION

4-2 SYNTAX

4-2 DESCRIPTION

4-2 EXAMPLES

4-4 **CAL: print calendar**

4-4 INTRODUCTION

4-4 SYNTAX

4-4 DESCRIPTION

4-5 EXAMPLES

4-6 **CALENDAR: reminder service**

4-6 INTRODUCTION

4-6 SYNTAX

4-6 DESCRIPTION

4-7 EXAMPLES

4-9 **DATE: print and set the date**

4-9 INTRODUCTION

4-9 SYNTAX

4-9 DESCRIPTION

4-11 EXAMPLES

4-12 **ECHO: echo arguments**

4-12 INTRODUCTION

4-12 SYNTAX

4-12 DESCRIPTION

4-13 EXAMPLES

5. THE EDITORS

5-1 INTRODUCTION

5-2 ED - a line editor

5-2 INTRODUCTION

5-3 GETTING STARTED

5-4 HOW TO ENTER ED

5-4 HOW TO CREATE TEXT

5-6 HOW TO DISPLAY TEXT

5-8 HOW TO DELETE A LINE OF TEXT

5-9 HOW TO MOVE UP OR DOWN IN A FILE

5-10 HOW TO SAVE THE BUFFER CONTENTS IN A FILE

5-11 HOW TO QUIT THE EDITOR

5-13 EXERCISE 1

5-14 GENERAL FORMAT OF ED COMMANDS

5-15 LINE ADDRESSING

5-17 SYMBOLIC ADDRESSES

5-19 RELATIVE ADDRESSES

5-24 SPECIFYING A RANGE OF LINES

5-25 SPECIFYING A GLOBAL SEARCH

5-27 EXERCISE 2

5-29 DISPLAYING TEXT IN A FILE

5-30 Displaying Text With Line Addresses

5-31 CREATING TEXT

5-34 Inserting Text

5-36 Changing Text

5-38 EXERCISE 3

5-40 DELETING TEXT

5-44 SUBSTITUTING TEXT

5-51 EXERCISE 4

5-53 SPECIAL CHARACTERS

5-62 EXERCISE 5

5-64 MOVING TEXT

5-73 EXERCISE 6

5-74 OTHER USEFUL COMMANDS AND INFORMATION

5-82 EXERCISE 7

5-84 **ANSWERS TO EXERCISES**

5-84 EXERCISE 1

5-85 EXERCISE 2

5-87 EXERCISE 3

5-89 EXERCISE 4

5-91 EXERCISE 5

5-93 EXERCISE 6

5-94 EXERCISE 7

5-96 VI - a screen editor

5-96 INTRODUCTION

5-98 SYNTAX

5-98 DESCRIPTION

5-99 EXAMPLES

5-100 HOW TO CREATE TEXT: THE APPEND MODE

5-101 HOW TO LEAVE APPEND MODE

5-101 EDITING TEXT: THE COMMAND MODE

5-102 HOW TO MOVE THE CURSOR

5-105 HOW TO DELETE TEXT

5-107 HOW TO ADD TEXT

5-108 QUITTING VI

5-110 EXERCISE 1

5-112 MOVING THE CURSOR AROUND THE SCREEN

5-127 POSITIONING THE CURSOR IN UNDISPLAYED TEXT

5-131 GO TO A SPECIFIED LINE

5-131 LINE NUMBERS

5-132 SEARCHING FOR A PATTERN OF CHARACTERS

5-138 EXERCISE 2

5-140 CREATING TEXT

5-144 EXERCISE 3

5-145 DELETING TEXT

5-151 EXERCISE 4

5-152 MODIFYING TEXT

5-159 CUTTING AND PASTING TEXT

5-163 EXERCISE 5

5-164 SPECIAL COMMANDS

5-167 USING LINE EDITING COMMANDS IN VI

5-173 QUITTING VI

5-176 SPECIAL OPTIONS FOR VI

5-178 EXERCISE 6

5-180 **ANSWERS TO EXERCISES**

5-180 EXERCISE 1

5-181 EXERCISE 2

5-183 **EXERCISE 3**

5-185 EXERCISE 4

5-187 EXERCISE 5

5-188 EXERCISE 6

6. THE FILE STORAGE COMMANDS

6-1 **INTRODUCTION**

6-2 **AR: archive and library utility**

6-2 INTRODUCTION

6-2 SYNTAX

6-2 DESCRIPTION

6-5 EXAMPLES

6-8 **PACK: compress files**

6-8 INTRODUCTION

6-9 SYNTAX

6-10 DESCRIPTION

6-10 EXAMPLES

6-12 **PCAT: concatenate and print packed files**

6-12 INTRODUCTION

6-12 SYNTAX

6-12 DESCRIPTION

6-13 EXAMPLES

6-14 **UNPACK: expands files**

6-14 INTRODUCTION

6-15 SYNTAX

6-15 DESCRIPTION

6-15 EXAMPLES

7. THE PATTERN EDITORS

7-1 **INTRODUCTION**

7-2 **AWK: pattern scanning and processing**

7-2 INTRODUCTION

7-2 SYNTAX

7-3 DESCRIPTION

7-5 LEXICAL UNITS

7-11 PRIMARY EXPRESSIONS

7-17 TERMS

7-19 EXPRESSIONS

7-21 EXAMPLES

7-54 SPECIAL FEATURES

7-66 **GREP: file pattern search utility**

7-66 INTRODUCTION

7-66 SYNTAX

7-66 DESCRIPTION

7-68 EXAMPLES

7-70 **SED: stream editor**

7-70 INTRODUCTION

7-70 SYNTAX

7-70 DESCRIPTION

7-73 EXAMPLES

7-74 USING ADDRESSES

7-76 THE WHOLE-LINE FUNCTIONS

7-78 SUBSTITUTE FUNCTIONS

7-80 INPUT/OUTPUT FUNCTIONS

7-82 MULTIPLE LINE FUNCTIONS

7-83 HOLD AND GET FUNCTIONS

7-83 FLOW OF CONTROL FUNCTIONS

7-84 QUITTING FROM SED

8. THE SYSTEM STATUS COMMANDS

8-1 INTRODUCTION

8-2 DF: reports free disk blocks and inodes

8-2 INTRODUCTION

8-2 SYNTAX

8-2 DESCRIPTION

8-3 EXAMPLES

8-4 DU: disk usage summary utility

8-4 INTRODUCTION

8-4 SYNTAX

8-4 DESCRIPTION

8-5 EXAMPLES

8-7 **SUM: print a file's checksum and block count**

8-7 INTRODUCTION

8-7 SYNTAX

8-7 DESCRIPTION

8-8 EXAMPLES

9. **THE MAIL SYSTEM**

9-1 **INTRODUCTION**

9-1 **MAIL: message transmission utility**

9-1 INTRODUCTION

9-2 EXCHANGING MESSAGES

9-3 SYNTAX

9-3 DESCRIPTION

9-5 SENDING MESSAGES

9-6 UNDELIVERABLE MAIL

9-7 SENDING MAIL TO ONE PERSON

9-9 SENDING MAIL TO SEVERAL PEOPLE SIMULTANEOUSLY

9-9 SENDING MAIL TO REMOTE SYSTEMS

9-13 MANAGING INCOMING MAIL

10. THE PROCESS HANDLING COMMANDS

10-1 INTRODUCTION

10-2 KILL: terminate a process

10-2 INTRODUCTION

10-2 SYNTAX

10-2 DESCRIPTION

10-3 EXAMPLES

10-6 NOHUP: runs a command immune to hangup and quit signals

10-6 INTRODUCTION

10-6 SYNTAX

10-6 DESCRIPTION

10-7 EXAMPLES

10-10 PS: report process status

10-10 INTRODUCTION

10-10 SYNTAX

10-10 DESCRIPTION

10-12 EXAMPLES

10-13 UMASK: set file-creation mode

10-13 INTRODUCTION

10-13 SYNTAX

10-13 DESCRIPTION

10-14 EXAMPLES

11. THE PROGRAMMING SUPPORT SYSTEM

11-1 INTRODUCTION

11-1 M4: C and RATFOR macro processor

11-1 GENERAL

11-4 DEFINING MACROS

11-9 ARGUMENTS

11-11 ARITHMETIC BUILT-IN MACROS

11-12 FILE MANIPULATION

11-14 SYSTEM COMMAND

11-14 CONDITIONALS

11-15 STRING MANIPULATION

11-18 PRINTING

GENERAL

This manual is the *User Guide* for the X/OS operating system. It takes the form of an Introduction and 10 tutorial chapters. Each of the tutorials covers a number of separate utilities. The coverage of each utility takes the same overall form, and consists of an brief introduction, a syntax definition, a description of the syntax and of the capabilities of the utility, and a series of examples which illustrate the utility in use. The conventions used in these chapters are described at the end of this Introduction.

MANUAL CONVENTIONS

Throughout this manual, certain conventions of presentation are used. Each tutorial chapter is divided into a number of sections. Each begins with a short *Introduction* which gives a summary of the utility in a few lines. The *Syntax* section defines the formal structure of the utility, with its options and arguments. In this section, optional elements are enclosed in square brackets, and elipses indicate that the element can be repeated.

The next section is the *Description*, which explains the syntax of the command. Not all the elements are necessarily described, as it is the intention of this manual to cover the basic operations of the X/OS operating system. Each command is covered in more detail in the Reference Manuals.

The fourth section is the *Examples*. These set out to illustrate the various elements of the utility in use. The tutorials include sample *screens*, which can be followed by typing in the command lines shown. These screens use the following conventions:

> 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. Most screens will end with a single system prompt symbol, which indicates that the operation is complete. Note that in some tutorials, use of the **>** symbol would cause confusion, as this character is a part of the utility's output. In this case, the **$** symbol is used instead.

Note also, that where the **root** user appears in an example, the system prompt is represented by a **#** symbol.

key many of the tutorials involve the use of specific keystrokes. 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 and **DEL** for delete. A further convention is use of the **CTRL** notation to indicate that the key marked **CONTROL** or **CTRL** should be pressed. For example **CTRL-d** indicates that the **CTRL** and **d** keys should be pressed together. This is called an *control sequence*.

Note that a list of associated manuals, and their reference codes, is given in the *Preface*, at the beginning of this manual.

The syntax descriptions of the individual utilities adopt the following conventions:

bold elements in bold type are those which, if used, are to be typed exactly as shown. They include command names, options, and special characters.

italics elements in italics are variables, which stand for actual values. For example, the word *name* indicates that an actual filename should be entered in the place of the variable name.

[ ] square brackets indicate that an element is optional.

... elipses indicate that the preceding element may be repeated.

| the logical OR symbol indicates that one of the elements linked by the |, but not both, should be entered.

INTRODUCTION

This first chapter of tutorials covers the ten X/OS commands considered to be the basis of the file and directory handling system. The chapter begins with a general explanation of how files and directories are used to structure a user's share of the available disk space. An explanation of what is meant by the *standard input* and *standard output* is also given. The ten utilities covered in this chapter are as follows:

CAT the file printing and concatenation utility

CD changes the current working directory

CHMOD the file protection utility

CP the file copying utility

LS the file and directory listing utility

MKDIR the directory creation utility

MV moves and renames files

PWD identifies the current working directory

RM removes a file from the system

RMDIR removes a directory from the system

These tutorials are arranged in alphabetical order.

FILES AND DIRECTORIES

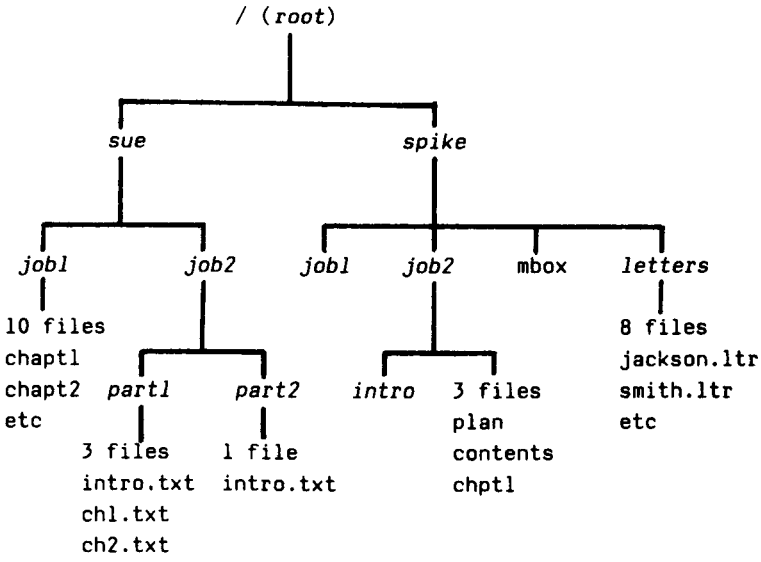
A *file* is a collection of information gathered together as a single unit. It is identified by a *filename*, with an optional *extension*. The two elements are usually separated by a period or full stop. The extension is often used to identify the type of file. Therefore, a file called *manual.txt* would appear to contain a collection of text lines, having a common subject matter, making up a manual of some sort. Note that it is always better to give a file a name that is meaningful.

X/OS allows the user to group files together into *directories*. These may contain only files, or only other directories, called *sub-directories* or a mixture of files and sub-directories. There is no practical limit to the number of levels of sub-directory.

With a large computer system, there may be a large number of users. To keep their work separate, the available user disk space is usually divided up into user areas. The following diagram shows a simple system with two users, called *spike* and *sue*. In the diagram, directories are named in italics, files in normal text.

There are two user directories, identified as *sue* and *spike*. User Sue has two sub-directories, which she uses to keep her projects separate. The first, called *job1*, contains ten files, called *chpt1*, *chpt2*, etc. Note that these have filenames, but no filename extensions. The second directory, called *job2* contains two further sub-directories, called *part1* and *part2*. These both contain files, this time, with filename extensions.

User Spike has three directories and a single file on his top level. Directory *spike* contains a sub-directory called *job2* which contains a further sub-directory and some files, and a directory called *letters* which holds only files.



In addition to their filenames, files are also identified by their position in the directory tree. This position is given by a *pathname*, which is the route taken from the top point of the directory system (called */* or *root*), to the file itself. this system also allows files in different locations to have the same filename.

As an example, Sue has two files called *intro.txt*. The first is identified as */sue/job2/part1/intro.txt* while the second is */sue/job2/part2/intro.txt*. Similarly, there are two files called *chpt1*, identified as */sue/job1/chpt1* and */spike/job2/chpt1*.

Many of the X/OS commands require the user to specify a filename. It is usually possible to give either a simple filename or a full pathname. This depends on the user's *current directory*. To write a letter, Spike would typically move into the directory called */spike/letters*. In this case, Spike's current directory would be *letters*. By moving to the directory called */spike/job2/intro*, this

would become his current directory.

Generally, it is sufficient to identify a file by its filename only if the file is contained in the current directory. If the file is elsewhere, it is necessary to give a full pathname.

For a list of the commands used to create and handle the directory and file system, see the list at the beginning of this chapter.

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 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 rest of this chapter illustrates these ideas in action.

CAT: file printing and concatenation

INTRODUCTION

This section is a brief introduction to the **cat** utility which is used to concatenate and print files. Files may be joined together, and new text may be added to existing files.

SYNTAX

```
cat [-u] [-s] [-v[-t][-e]] [-lfile ...]
```

DESCRIPTION

The arguments to **cat** are as follows:

-u output from **cat** is buffered unless this option is given

-s suppresses error messages regarding non-existent files

-v non-printing characters, with the exception of tabs, newlines and formfeeds, are printed visibly.

-t used with the **v** option: causes tabs to be displayed as ^I

-e used with the **v** option: causes a **$** to be printed at the end of each line, immediately before the newline character.

Neither the **t** nor the **e** options are executed if

the **v** option has not been specified.

file specifies the file or files to be processed. Filenames may be linked using the > and >> characters, as illustrated in the examples, below. If no input filename is given, or if *file* takes the form -, **cat** reads from the standard input.

EXAMPLES

In the following examples, remember that the > in bold face represents the system prompt, and that **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

When used with one or more filenames, **cat** will display the contents of the named files. In the first example, **cat** is used to display the contents of a single file, called *knowledge*.

```
>cat knowledge CR
What is felicific calculus?
A system, invented by Bentham, for preforming
quantitative comparisons of the amount of
pleasure and pain arising from alternative
courses of action.

>
```

More than one file can be displayed in this way, simply by giving a list of filenames on the command line.

Files can be concatenated using the > symbol. This is one of the shell's *re-direction operators*. These are explained in detail in the **sh** and **csh** tutorials, in the *Shell / C Shell X/OS Command Language User Guide*. The next example joins the contents of two files and places

them in a third. The contents of the original two files are not affected. The contents of the three files are displayed.

```
>cat Monroe1 CR
What is the Monroe Doctrine?
The principle formed in 1823 that the territories
of the American continents are not be considered
as subjects for European annexation.

>cat Monroe2 CR
It also stated that The United States should not
become involved with existing European colonies in
the Americas, or become embroiled in European wars.

>cat Monroe1 Monroe2 > Monroe3 CR
What is the Monroe Doctrine?
The principle formed in 1823 that the territories
of the American continents are not be considered
as subjects for European annexation.
It also stated that The United States should not
become involved with existing European colonies in
the Americas, or become embroiled in European wars.

>
```

Another redirection operator used by **cat** is **>>**, which appends one file onto the end of another. This is useful when the existing contents of a file must not be overwritten. The next example again combines the files called *Monroe1* and *Monroe2*, by adding the second to the end of the first.

```
>cat Monroe2 >> Monroe1 CR
```

```
>cat Monroel CR
What is the Monroe Doctrine?
The principle formed in 1823 that the territories
of the American continents are not be considered
as subjects for European annexation.
It also stated that The United States should not
become involved with existing European colonies in
the Americas, or become embroiled in European wars.

>
```

CD: change working directory

INTRODUCTION

This is a short tutorial-style introduction to the **cd** utility which enables the user to move from one directory to another. A detailed description of files and directories is given in the Introduction to this manual. The present chapter will restrict itself to explaining how **cd** uses this system.

SYNTAX

---

cd [directory]

DESCRIPTION

Immediately after logging in, users are located in their home directory. For as long as they work in that directory, it is also the current working directory. However, the **cd** command (which stands for *change directory*), allows work to be done in other directories as well. The argument to **cd** is as follows:

directory specifies the destination directory, that is, the directory in which work is to be done. It may take the form of a filename or a pathname. The full range of X/OS filename expansion metacharacters are available. These are explained in detail in the **sh** and **csh** tutorials in the *Shell / C Shell X/OS Command Language User Guide*.

If no *directory* is specified, **cd** makes the user's home directory the current working

directory. The location of this directory is stored in the environment variable **$HOME**, which is described in the tutorials in the *Shell / C Shell X/OS Command Language User Guide*.

Note that the **pwd** (*print working directory*) command can be used to find out which is the currently active directory. This is described later in this chapter.

EXAMPLES

The first example illustrates how to move into a new directory, called *project2*. Remember that the **>** in bold face represents the system prompt, and that the **CR** symbol indicates that the carriage return key should be pressed in order to enter the command line.

These examples use the diagram that appears at the beginning of this chapter. The first command assumes that user Spike has just logged on to the system, and that his home directory, */spike*, is now his working directory.

```
>cd project2 CR

>
```

This command makes */spike/job2* the new current directory.

The next example shows how to use the *dot dot* metanotation to climb up one level in the directory tree. For example, to move from a directory identified by the pathname */spike/job2* to the directory */spike*, this metanotation can be used. Notice that the **pwd** command is used in order to identify the current location.

```
>pwd CR
/spike/job2

>cd .. CR

>pwd CR
/spike

>
```

To move from */spike/job2* to a parallel directory, called */sue/job2*, the route taken would have involved moving up two levels, with the *dot dot* notation, then back down two levels:

```
>pwd CR
/spike/job2

>cd ../../sue/job2 CR

>pwd CR
/sue/job2

>
```

For a full description of the **pwd** command, see its tutorial in this manual. The metanotation available for use with the **cd** command is described in detail in both of the shell tutorials, in the *Shell / C Shell X/OS Command Language User Guide*.

CHMOD: file protection

INTRODUCTION

This is a short tutorial-style introduction to the **chmod** utility (short for *change mode*). It allows the user to decide who can read, write, and use files and who cannot. Because the X/OS operating system is a multi-user system, users usually do not work alone in the file system. System users of files and directories can write, read and use files belonging to one another, as long as **chmod** has been used to set the appropriate permissions.

SYNTAX

```
chmod mode files
```

DESCRIPTION

The arguments to **chmod** are as follows:

mode sets the permissions available to users. *Mode* may take an absolute or symbolic form. Absolute modes are derived from the OR of the octal codes representing certain conditions of access. These are listed in full in the *chmod*(1) entry of the *Utilities Reference Manual*. An example of an absolute mode is given below.

Symbolic modes take the form

```
[who] op perm [op perm]
```

where:

who takes the form of one or more letters identifying the class of user, as follows:

- **u** the owner of the file or directory, that is, the *user*
- **g** members of the arbitrarily designated *group* of users, perhaps those working on the same project
- **o** all *other* users with access to the system
- **a** *all* users, that is **ugo**

op one of the following operators:

- **+** add a permission
- **-** remove a permission
- **=** assign a permission

perm The permission to be assigned to the files or directories specified on the command line. Permissions are as follows:

- **r** allows the specified class of users to *read* a file or directory, and to copy its contents
- **w** allows the specified class of users to *write* changes to a file or directory
- **x** allows the specified class of users to *execute* a file, or gain access to a directory, as long as it takes the correct form

files specifies the files to be assigned permissions

Note that a file's existing permissions can be determined by using the **ls -l** command, which gives a *long* listing of the file specifications. The meaning of the codes used by **ls** when executed with the **-l** option are explained in the **ls** tutorial in this manual, and in the *ls*(1) entry of the *Utilities Reference Manual*.

EXAMPLES

This section provides a number of simple examples of **chmod** in use. Remember that the **>** in bold face represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

In the first example, **ls -l** is used to produce a long listing of some files. Note that the first entry is a directory. This is indicated by the letter *d* in the first column of the output. Note that columns 2 to 4 indicate the permissions for the owner, columns 5 to 7 the permissions for the group, and columns 8 to 10 the permissions for all users. The example then goes on to assign read and write permissions to a file called *plan* for members of the owner's group.

```
>ls -l CR
drwxr-xr-x  2 spike    2536  Jul  9  16:42  intro
-rw-------  1 spike    4022  Jul 21  13:09  plan
-rw-------  1 spike   10885  Jul 21   9:55  contents
-rw-------  1 spike    8803  Jul 19  10:04  chptl

>chmod g+rw plan CR

>ls -l plan CR
-rw-rw----  1 spike    4022  Jul 21  13:09  plan
```

The second example removes execute permission from the directory *intro* for those users classed as *others*, that is, not the owner of the directory, and not a member of the owner's group.

```
>ls -l intro CR
drwxr-xr-x  2 spike    2536  Jul  9  16:42  intro

>chmod o-x intro CR

>ls -l intro CR
drwxr-xr--  2 spike    2536  Jul  9  16:42  intro

>
```

Granting execute permission for a directory allows the specified users access to it, using **cd**, and allows them to list its contents with the **ls** command. By removing such permission, the *other* users can no longer enter that directory.

The third example sets read, write and execute permission for the other two text files in the listing, for all users. Note that the class of *all* users can be referred to by the combination **ugo** (*user-group-others*) or simply by **a** for *all*. Note the use of the asterisk to set the mode for more than file in the single operation.

```
>chmod a+rwx c* CR

>ls -l c* CR
-rwxrwxrwx  1 spike   10885  Jul 21   9:55  contents
-rwxrwxrwx  1 spike    8803  Jul 19  10:04  chptl

>
```

The last example uses the absolute method of setting permissions. The full list of mode values is available in the *chmod*(1) entry of the *Utilities Reference Manual*. To combine permissions, the octal values are added. Here are some examples from the list:

00400 sets read permission for the owner

00200 sets write permission for the owner

00040 sets read permission for the group

00020 sets write permission for the group

To assign these four permissions to the file called *contents*, the four values are added together, and the total assigned to the file, as follows:

```
>chmod 660 contents CR

>ls -l contents CR
 -rw-rw----  1 spike    4022  Jul 22   9:55  contents

>
```

CP: file copying

INTRODUCTION

This is a brief tutorial-style introduction to the **cp** file copying utility. If the destination is a directory, a copy of the file is added to that directory. If the file already exists, the existing contents are over-written, unless precautions are taken. The destination file may be given a new name in the process.

SYNTAX

```
cp file ... target
```

DESCRIPTION

The arguments to **cp** are as follows:

file specifies the file or files to be copied

target specifies the destination of the copy. *Target* may be a directory or a file. In the case of *target* being a directory, a new copy of the file is created there, using the same set of access permissions as the original (see the **chmod** tutorial in this manual for an explanation of permissions).

If *target* is a file, the existing contents are over-written, unless the access permissions preclude this. The modification time (as obtained using the **ls -l** command, see the **ls** tutorial in this manual) is up-dated to the time and date the copy was made.

Note that under no circumstances will X/OS allow a file to be copied to itself. Trying to do this will generate an error message. This implies that it is impossible to have two files of the same name in the same directory. Note also that **cp** operations on a file will leave the original in place. To remove a file from its original location, the **mv** command must be used instead.

EXAMPLES

These examples assume that the current working directory is */sue/job1*. The first uses **cp** to make a copy of a file called *chapt1*. The destination of the copy is to be the same directory. This means that the copy *must* be given a new name. The first command line accidentally specifies the same name for both the original and the target. The error message warns that a file cannot be copied onto itself. In the second command line, the copy is correctly named *chapt1.new*. The advantage of this kind of operation is that time can be saved when two documents are required which have much in common: the first can be used as the basis for the second.

Remember that the **>** in bold represents the system prompt, and that the **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>cp chaptl chaptl CR
cp: chaptl and chaptl are identical

>cp chaptl chaptl.new CR

>ls -l CR
-rw-------   1 spike    78691  Jul 21  09:09  chaptl
-rw-------   1 spike    78691  Jul 24  11:43  chaptl.new

>
```

The second example copies *chaptl.new* from the current directory to another directory called */sue/job2/part2* while leaving the original files in place. Note the use of the filename expansion metanotation to access both files at once. Metanotation is explained in both of the shell tutorials, in the manual called *Shell/ C Shell X/OS Command Language User Guide*. The second and third command lines in this example use the **ls** command to check the original directory and the destination directory. Both have copies of the two files.

```
>cp chaptl.new /sue/job2/part2 CR

>ls CR
chaptl
chaptl.new
chapt2
.
.

>ls /sue/job2/part2 CR
intro.txt
chaptl.new

>
```

Note that the destination directory */sue/job2/part2* had to already exist, and that the original filenames were retained.

The third example copies a file called *chapt4* from the current directory to a directory called */spike/jobl*, and renames it *chapt2.txt* in the process. The **ls -l** command lines confirm that the new file exists, and that it has the same attributes as the original, except for the filename and modification time.

```
>ls -l CR
-rw-r--r--   1 spike     5541  Jul 21  10:49  chapt4

>cp chapt4 /spike/jobl/chapt2.txt CR

>ls -l /spike/jobl CR
-rw-r--r--   1 spike     5541  Jul 21  10:49  chapt2.txt

>
```

The final example copies the file called */sue/job2/part1/intro.txt* to directory */sue/job2/part2*, where a file with this name already exists. The copy operation over-writes the existing file. This can be seen from the **ls -l** listings.

```
>ls -l /sue/job2/part1/intro.txt CR
-rw-rw-rw-   1 sue     5541  Apr 11  10:49  intro.txt

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

>ls -l /sue/job2/part1 CR
-rw-rw-rw-   1 sue    10565  Jul 21  14:32  intro.txt

>
```

The first **ls -l** listing gave the details of a small file called *intro.txt*, created on April 11. After the copy operation has been carried out, a listing of the same directory shows a much larger file created on July 21. Note that the original had write permission, and so was susceptible to change.

LS: file and directory listing

INTRODUCTION

This chapter is a short tutorial-style introduction to the X/OS **ls** utility which is used to list the files and sub-directories held by a directory. Without arguments, **ls** will list the names of files and directories held in the current working directory. There are many options available, allowing different types of listing to be printed.

SYNTAX

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

DESCRIPTION

The **ls** utility supplies 21 possible options. These are described in full in the *ls*(1) entry of the *Utilities Reference Manual*. Accordingly, only a few options will be described here.

-R recursively lists the contents of all sub-directories below the current working directory. This option is useful as an indicator of the overall directory and file system.

-a lists all entries. Usually, filenames beginning with a period or full stop (.) are not printed. These are special files such as the directories (which cannot be read by the user) that contain information needed by X/OS to maintain the directory system. The user's home directory will also contain a number of files that establish the

various shell variables. These are described in the shell tutorials in the *Shell / C Shell X/OS Command Language User Guide*.

-C lists directory contents in multi-column format, with entries sorted down the columns.

-l lists in long format the contents of the current working directory. Long format involves giving details of the file or directory in the following format:

```
-rw-r--r--  1 spike   GRP2  26538 Aug  6 15:33 chapter1.txt
```

where the first ten characters give the file's *permission* settings, and the following number indicates the number of links for the file. The names indicate the owner of the file and the group to which the user belongs. The next number gives the file's size. The date and time indicate when the file was last modified, and the last field is the filename. These items of information are described in greater detail in the shell tutorials, and in the **chmod** tutorial in this manual.

-t sorts the file listing according to the time of the last modification, with the most recently modified file appearing first.

-u sorts the file listing according to the time of the last access.

names identifies the directory or the class of files to be listed. If no *name* is given, the contents of the current working directory is listed. Note that the full range of filename expansion notations is available. These are described in the shell tutorials.

-F places a slash (/) after each entry if the entry is a directory, and an asterisk after each entry if the entry is an executable file.

EXAMPLES

In the following examples, the > character in bold type represents the system prompt, and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

The **ls** command lists the names of all files and subdirectories in a specified directory. If you do not specify a directory, **ls** lists the names of files and directories in your current directory. To understand how the **ls** command works, consider the sample directory and file system shown at the beginning of this chapter.

After logging on to the X/OS system, you run the **pwd** command. The system responds with the path name */spike*. This is user *spike*'s home directory. To display the names of files and directories in this current directory, you then type **ls** and press the **CR** key. After this sequence, your terminal will read:

```
>pwd CR
/spike

>ls CR
jobl
job2
letters
mbox

>
```

As you can see, the system responds by listing, in alphabetical order, the names of files and directories in

the current directory */spike*. (If the first character of any of the file or directory names had been a number or an upper case letter, it would have been printed first.)

To print the names of files and subdirectories in a directory other than your current directory without moving from your current directory, you must specify the name of that directory as follows:

ls *pathname*

The directory name can be either the full or relative path name of the desired directory. For example, you can list the contents of *job2* while you are working in */spike* by entering **ls job2** and pressing the **CR** key. Your screen will look like this:

```
>ls job2 CR
intro
plan
contents
chptl

>
```

You can also use a relative path name to print the contents of a parent directory when you are located in a child directory. The .. (dot dot) notation provides an easy way to do this. For example, the following command line specifies the relative path name from */spike* to */root*:

```
>ls .. CR
spike
sue
```

If you type **ls /** and press the **CR** key, the system will respond by printing the same list.

Similarly, you can list the contents of any system directory that you have permission to access by executing the **ls** command with a full or relative path name.

The **ls** command is useful if you have a long list of files and you are trying to determine whether one of them exists in your current directory. For example, if you are in the directory *job2* and you want to determine if the files named *contents* and *notes* are there, use the **ls** command as follows:

```
>ls contents notes CR
contents
notes not found

>
```

The system acknowledges the existence of *contents* by printing its name, and says that the file *notes* is not found.

The **ls** command does not print the contents of a file. If you want to see what a file contains, use the **cat**, **pg**, or **pr** command. All of these commands are described in their own tutorials in this manual.

FREQUENTLY USED LS OPTIONS

The **ls** command also accepts options that cause specific attributes of a file or subdirectory to be listed. Of these, the **-a** and **-l** will probably be most valuable in your basic use of the X/OS system. Refer to the *ls*(1) page in the *Utilities Reference Manual* for details about other options.

LISTING ALL FILES IN A DIRECTORY

Some important file names in your home directory, such as *.profile* (pronounced dot-profile), begin with a period or full stop. (As you can see from this example, when a period is used as the first character of a file name it is pronounced dot.) When a file name begins with a dot, it is not included in the list of files reported by the **ls** command. If you want the **ls** to include these files, use the **-a** option on the command line.

For example, to list all the files in your current directory (*spike*), including those that begin with a dot, type **ls -a** and press the **CR** key.

```
>ls -a CR
.
..
.profile
job1
job2
letters
mbox

>
```

LISTING FILES IN SHORT FORMAT

The **-C** and **-F** options for the **ls** command are frequently used. Together, these options list a directory's subdirectories and files in columns, and identify executable files (with an *) and directories (with a /). Thus, you can list all files in your working directory *spike* by executing the command line shown here:

```
>ls -CF CR
job1/      job2/      letters/      mbox

>
```

LISTING FILES IN LONG FORMAT

Probably the most informative **ls** option is **-l**, which displays the contents of a directory in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file. For example, say you run the **ls -l** command while in the *spike* directory.

```
>ls -l CR
total 10
drwxr-xr-x  4 spike  GRP2   512 Jul 21 15:43  job1
drwxr-xr-x  2 spike  GRP2   512 Jul 18 12:13  job2
drwxr-xr-x  3 spike  GRP2   512 Aug  6  9:06  letters
-rwxr-xr-x  1 spike  GRP2  1273 Aug 13 12:09  mbox

>
```

The first line of output, *total 9* shows the amount of disk space used, measured in blocks. Each of the rest of the lines comprises a report on a directory or file in *spike*. The first character in each line (**d**, **-**, **b**, or **c**)

tells you the type of file, where

d directory

\- ordinary disk file

b block special file

c character special file

Using this key to interpret the previous screen, you can see that the *spike* directory contains three directories, and no files.

The next several characters, which are either letters or hyphens, identify who has permission to read and use the file or directory. (Permissions are discussed in the description of the **chmod** command in this manual.)

The following number is the link count. For a file, this equals the number of users linked to that file. For a directory, this number shows the number of directories immediately under it plus two (for the directory itself and its parent directory).

Next, the login name of the file's owner appears (here it is *spike*), followed by the group name of the file or directory.

The following number shows the length of the file or directory entry measured in units of information (or memory) called bytes. The month, day, and time that the file was last modified is given next. Finally, the last column shows the name of the directory or file.

MKDIR: create a directory

INTRODUCTION

This chapter is a short tutorial-style introduction to the X/OS **mkdir** utility, which will create a new directory at the specified location in the directory tree system. The specified location may be either the current working directory, or a remote directory specified by an absolute or relative pathname.

SYNTAX

```
mkdir pathname ...
```

DESCRIPTION

The *pathname* argument specifies the name and position of the new directory to be created. For a description of how the file and directory system works, see the Introduction to this manual.

Note that when creating a new directory, you can specify the pathname of the directory using the full range of shell file name expansion notations. These are also described in the shell tutorials. The most important of these are . for the current directory, and .. for the parent of the specified directory, that is, the directory one level up in the tree.

The following section again uses the sample file and directory system outlined at the beginning of this chapter. For example, the parent of the directory called *job2* is */spike*. The notation .. defines the relationship between *job2* and */spike*.

EXAMPLES

In the following examples, the > character in bold type represents the system prompt, and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

It is recommended that you create sub-directories according to a logical and meaningful scheme that will facilitate the retrieval of information from your files. If you put all files pertaining to one subject together in a directory, you will know where to find them later.

To create a directory, use the command **mkdir** (short for *make directory*). Simply enter the command name, followed by the name you are giving your new directory or file. For example, in the sample file system, the user called Spike created *jobl* by issuing the following command from the home directory (*/spike*):

```
>mkdir jobl CR

>
```

The second prompt shows that the command has succeeded; the sub-directory *jobl* has been created.

Still in the home directory, this user created other sub-directories, called *job2* and *letters*, in the same way.

```
>mkdir job2 CR

>mkdir letters CR

>
```

The user could have created all three subdirectories (*job1*, *job2*, and *letters*) simultaneously by listing them all on a single command line.

```
>mkdir job1 job2 letters CR

>
```

From */spike*, it is possible to create a new sub-directory in the directory called */spike/job2* by specifying a pathname for **mkdir**.

```
>mkdir job2/intro CR

>
```

In this way, the directory called */spike* would now contain three directories called *job1*, *job2* and *letters*, and a sub-directory called *intro*.

MV: moving and renaming a file

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **mv** utility (short for *move*), which allows you to rename a file or sub-directory while retaining its location, or to move a file from one directory to another. If you move a file to a different directory, the file can be renamed or it can retain its original name.

SYNTAX

```
mv [-f] file [file ...] target
```

DESCRIPTION

The **mv** utility checks that the access permissions of the target location allow the move to take place. If they preclude the move, **mv** will print the access permissions of the target location, and prompt for permission from the user to continue. Any string beginning with *y* cause the move to take place. A string beginning with any other character cause **mv** to terminate.

The options and arguments to **mv** are as follows:

-f performs the move operation silently, without asking for conformation, in the event of adverse access permissions.

file specifies the source file to be moved or renamed. Note that *file* can be a directory in the case of a rename operation. Note also that more than one file can be relocated.

target specifies the destination of the move or rename. If *target* is a directory, then *file* is relocated. If *target* is a filename, then *file* is renamed. If *target* is a pathname ending in a filename, then *file* is both relocated and renamed.

Where *file* identifies one or more files, they retain their original mode, owner and group. The time and date of modification of both *file* and *target* are updated.

EXAMPLES

In the following examples, the **>** symbol in bold type represents the system prompt, while **CR** indicates that the carriage return key should be pressed in order to enter the command line.

The first example renames a file called *intro.txt* to *interim*, within one directory. The first command line in the example uses the **ls** command to confirm the existence of *intro.txt*. The second command line renames the file, then the third confirms that the rename was successful.

```
>ls CR
intro.txt

>mv intro.txt interim CR

>ls CR
interim

>
```

The **mv** command changes a file's name from *intro.txt* to *interim* and deletes *intro.txt*. Remember that *file* and *target* can be any valid names, including path names.

In the second example, the current working directory is called */sue/job2/part2*. A file called *intro.txt* is to be moved to the directory called */sue/jobl*. It is to retain its original name. The first command line checks the contents of the current directory, using **ls**. The second command carries out the move, while the third again checks the current directory. The last command line confirms that the target directory contains the new file.

```
>ls CR
intro.txt

>mv intro.txt /sue/jobl CR

>ls CR

>ls /sue/jobl CR
intro.txt
chaptl
chapt2
.
.

>
```

The third example is a variation on the last sequence, which renames *intro.txt* in the process. The new filename is *intro.new*.

```
>ls CR
intro.txt

>mv intro.txt /sue/jobl/intro.new CR

>ls CR
```

```
>ls /sue/jobl CR
intro.new
chaptl
chapt2
.
.

>
```

PWD: print working directory

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **pwd** utility. It will print out the name of the current working directory. It is therefore useful when using the X/OS directory system. In addition to the **pwd** utility, this chapter will cover the basic outlines of the directory and file system.

SYNTAX

---

pwd

DESCRIPTION

When you log on to an X/OS system, you will be located in your *home* directory. This acts as the entry point to your own area of user memory. As long as you continue to work in your home directory, it is considered your current working directory. If you move to another directory, that directory becomes your new current directory.

The X/OS system command **pwd** (short for *print working directory*) prints the name of the directory in which you are now working.

In the following examples, remember that the **>** symbol in bold face type represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line. For the sake of the examples, your login name is *spike*.

If you execute the **pwd** command in response to the first prompt after logging in, the X/OS system will give a reply that may look like the following:

```
>pwd CR
/spike

>
```

In this way, the system response gives you both the name of the directory in which you are working (*spike*) and the location of that directory in the file system. The path name */spike* tells you that the root directory (shown by the leading / in the line) contains the directory *spike*. (All other slashes in the path name other than root are used to separate the names of directories and files, and to show the position of each directory relative to root.) A directory name that shows the directory's location in this way is called a full or complete directory name or path name. In the next few pages we will analyze and trace this path name so you can start to move around in the file system.

Remember, you can determine your position in the file system at any time simply by issuing a **pwd** command. This is especially helpful if you want to read or copy a file and the X/OS system tells you the file you are trying to access does not exist. You may be surprised to find you are in a different directory than you thought.

The next section examines path names in a little more detail. The chapter ends with some examples of **pwd** in use.

PATH NAMES

Every file and directory in the X/OS system is identified by a unique path name. The path name shows the location of the file or directory, and provides directions for reaching it. Knowing how to follow the directions given by a path name is your key to moving around the file system successfully. The first step in learning about these directions is to learn about the two types of path names: full and relative.

FULL PATH NAMES

A full path name (sometimes called an absolute path name) gives directions that start in the root directory and lead you down through a unique sequence of directories to a particular directory or file. You can use a full path name to reach any file or directory in the X/OS system in which you are working.

Because a full path name always starts at the root of the file system, its leading character is always a / (slash). The final name in a full path name can be either a file name or a directory name. All other names in the path must be directories.

To understand how a full path name is constructed and how it directs you, consider the following example. Suppose you are working in the */spike/job2* directory. You issue the **pwd** command and the system responds by printing the full path name of your working directory: */spike*. Analyze the elements of this path name using the following list:

first /
: the slash that appears as the first character of the path name signifies the *root* position of the directory system, that is, the anchor point of the tree

spike	the first path name element names the directory one level below *root*
following /'s	the second and subsequent slashes act as dividers, separating the various directory names in the path name
job2	the last name in the path name identifies the current working directory. This is the directory that is identified by the **pwd** command

RELATIVE PATH NAMES

A relative path name gives directions that start in your current working directory, and lead you up or down through a series of directories to a particular file or directory. By moving down from your current directory, you can access files and directories you own. By moving up from your current directory, you pass through layers of parent directories to the grandparent of all system directories, *root*. From there you can move anywhere in the file system.

A relative path name begins with one of the following: a directory or file name; a . (pronounced dot), which is a shorthand notation for your current directory; or a .. (pronounced dot dot), which is a shorthand notation for the directory immediately above your current directory in the file system hierarchy. The directory represented by .. (dot dot) is called the parent directory of . (your current directory).

For example, say you are in the directory */spike* in the sample system and */spike* contains directories named *job1*, *job2*, and *letters*, and a file named *mbox*. The relative path name to any of these is simply its name, such as *letters* or *mbox*. The diagram at the beginning of this chapter illustrates this relationship.

The *job2* directory belonging to */spike* contains three files and a sub-directory. The relative path name from */spike* to the file *contents* is *job2/contents*. Notice that the slash in this path name separates the directory named *job2* from the file named *contents*. Here, the slash is a delimiter showing that *contents* is subordinate to *job2*; that is, *contents* is a child of its parent, *job2*.

So far, the discussion of relative path names has covered how to specify names of files and directories that belong to, or are children of, your current directory. You now know how to move down the system hierarchy level by level until you reach your destination. You can also, however, ascend the levels in the system structure or ascend and subsequently descend into other files and directories.

To ascend to the parent of your current directory, you can use the .. notation. This means that if you are in the directory named *intro* in the sample file system, .. is the path name to *job2*, and *../..* is the path name to *spike*'s parent directory, /, called *root*.

From *intro*, you can also trace a path to the file *smith.ltr* by using the path name *../../letters/smith.ltr*. The first .. brings you up to *job2*, and the second to */spike*. Then the names *letters* and *smith.ltr* take you down through the *letters* directory to the *smith.ltr* file.

Keep in mind that you can always use a full path name in place of a relative one.

EXAMPLES

The following examples use the **pwd** command to check on the current working directory. Also used is the **cd** command. This is described in its own tutorial in this manual. Its name stands for *change directory*. It is here used with the path name notation described above, using the directory system laid out in the diagram.

Remember that the > symbol in bold type represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>pwd CR
/spike

>cd letters CR

>pwd CR
/spike/letters

>cd ../job2 CR

>pwd CR
/spike/job2

>cd ../.. CR

>pwd CR
/

>
```

Note that the full range of shell file name expansion metanotation can be used when moving around the directory system. This is explained in the two shell tutorials in the *Shell / C Shell X/OS Command Language User Guide*.

RM: file deletion

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **rm** utility which is used to remove one or more files.

SYNTAX

```
rm [-fri] file ...
```

DESCRIPTION

When entered without the options, **rm** deletes the named *file*. This argument may be one or more filenames, and may use the full range of shell filename and directory name expansion notation. This is described in the tutorials in the *Shell / C Shell X/OS Command Language User Guide*. In order to delete a file, the directory that contains it must have write permission, otherwise an error message is displayed. Access permissions are also described in the shell tutorials, as well as the **chmod** tutorial in this manual. Attempting to use **rm** to delete a directory will also cause an error message.

The behaviour of **rm** can be modified using the following options:

-f suppresses error messages arising from the directory that holds the file having the wrong access permissions.

-r where *file* is entered as a directory name, this option deletes the entire contents of the directory, including sub-directories, then deletes the

directory itself. Note that this operation will fail if the current directory is the directory to be deleted. A directory can be deleted only from outside.

-i requests permission from the user to proceed. This is a useful option to use when the * and ? notation is used to delete a number of files at once. If the user types a string beginning with the letter *y*, the file will be deleted, and the user will be prompted about the next file. Any other response will cause **rm** to ignore that file.

These options can be combined. For example, a combination of the **-i** and **-r** options can prevent an accidental deletion of more data than was intended.

EXAMPLES

In the example which follows, the **>** symbol in bold represents the system prompt, and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

In the example, the contents of the current directory are listed using the **ls -l** command. This is explained in its own tutorial in this manual. Two command lines follow; the first deletes a single file, while the second uses the **-i** option to ensure that the shell's filename expansion notation deletes only the required files. The current working directory is */spike/job2* in this example.

```
>ls -l CR
-rwxr--r--  1 spike   35648  12:06 12 Jul 1987  plan
drwxr-xr-x  1 spike     512  11:43  8 Jul 1987  intro
-rwxr--r--  1 spike    2268   9:26 11 Jul 1987  contents
-rwxr--r--  1 spike   11451  15:22 16 Jul 1987  chptl
```

```
>rm contents CR

>rm -i * CR
rm: remove plan? CR
rm: remove chptl? y CR

>ls -l CR
-rwxr--r--  1 spike   35648  12:06 12 Jul 1987  plan
drwxr-xr-x  1 spike     512  11:43  8 Jul 1987  intro

>
```

Note that by typing only **CR**, the user told **rm** that the file called *plan* was not to be removed, while typing **y** followed by **CR** caused *chptl* to be deleted.

RMDIR: directory deletion

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **rmdir** utility, which is used to remove an empty directory.

SYNTAX

```
rmdir dir [dir ...]
```

DESCRIPTION

Directories that are no longer required can be removed using **rmdir** (short for *remove directory*). This utility accepts only one argument, the name of the directory to be deleted. Note that more than one directory name may be specified on the command line.

The **rmdir** command will not remove a directory if you are not the owner of it or if the directory is not empty. If you want to remove a file in another user's directory, the owner must give you write permission for the parent directory of the file you want to remove.

If you try to remove a directory that still contains subdirectories and files (that is, is not empty), **rmdir** prints the message *dir not empty*, where *dir* is the name of the directory entered on the command line. All subdirectories and files must be removed; only then will the command succeed. Note also that it is forbidden to remove a directory while it is still the current directory. Removing a directory can be done only from above.

Note that under certain circumstances, there may be hidden files, which are not listed by the **ls** or **ls -l** commands. If you have attempted to remove all files and directories, and **rmdir** continues tp display the *not empty* error message, use **ls -a**, which lists the so-called *dot* files. Note also that the two entries called *dot* and *dot dot* cannot be removed manually, and will be deleted automatically by **rmdir**.

EXAMPLES

In the following example, the > symbol represents the system prompt, while **CR** indicates that the carriage return key must be pressed in order to enter the command line.

In the example, there is a directory called */spike/job2* which contains three files, *plan*, *contents* and *chptl*, and a directory called *intro*.

The first command displays the name of the current directory. The second attempts to delete *job2*. The error message indicates that it must be emptied first. The third uses **ls** to find out what is still in the directory, while the fourth deletes the three files. The next command successfully deletes the directory called *intro*. The second attempt at deleting *job2* also fails, this time for a different reason. In order for the command to have been successful, the current directory would have had to be above *job2* in the directory system. The **cd** command is therefore used to climb up one level, before the **rmdir** command is used again, this time successfully.

```
>pwd CR
/spike/job2

>rmdir job2 CR
rmdir: job2 not empty
```

```
>ls CR
intro
plan
contents
chptl

>rm * CR

>rmdir intro CR

>ls CR

>rmdir job2 CR
rmdir: job2: No such file or directory

>cd .. CR

>rmdir job2 CR

>
```

INTRODUCTION

This second set of tutorials covers the 18 commands supplied by X/OS that are regularly used in various areas of file handling. Some are used to reshape existing files, some for comparing files, and others for supplying information about the size or type of files.

The commands covered are as follows:

COMM	selects or rejects common lines from files
CUT	prints selected fields from a file
DIFF	identifies the differences between two files
DIFF3	identifies the differences between three files
FILE	determines the type of a file
NL	the line numbering utility
PASTE	the file merge utility
PG	pages through the contents of a file
PR	prints the contents of a file
SORT	the file sorting and merging utility
SPELL	the spelling check utilities
SPLIT	splits a file
TAIL	prints the last part of a file
TEE	the pipe fitting utility
TEST	the expression evaluation utility

TR the character translation utility

UNIQ searches for repeated lines in a file

WC the file size reporter

These tutorials are arranged in alphabetical order.

COMM: select or reject common lines

INTRODUCTION

This section is a tutorial introduction to the X/OS **comm** file comparison utility. Two files are compared, and the output consists of pointers to the differences and similarities between them. The two files should already have been sorted into alphabetical order, that is, according to ASCII code order. Note that special characters such as parentheses, asterisks, punctuation and quote marks, and others, have their own ASCII values, and that sorting will also take account of these. A copy of the ASCII code tables can be found in the *ascii*(5) entry of the *System Interfaces and Libraries Reference Manual*.

The output is laid out in three columns. The first contains lines found only in the first named file. The second column contains lines found only in the second file. The third lists elements common to both files.

SYNTAX

```
comm [-[123]] file1 file2
```

DESCRIPTION

The arguments and options to **comm** are as follows:

file1 identifies the first file to be compared.

file2 identifies the second file to be compared.

- specifies that the standard input is to be compared to a named file.

123 a single digit identifying the output column to be suppressed. For example, to print only the second and third columns, **-1** would be entered.

EXAMPLES

The following examples use files containing alphabetically sorted data. The first screen shows their contents, displayed using the **cat** command. Remember that the **>** symbol in bold represents the system prompt, and that **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

```
>cat fishlist1 CR
basking shark
cod
haddock
lamprey
trigger fish

>cat fishlist2 CR
anchovy
basking shark
cod
halibut
lamprey
```

In the first example, **comm** is used in its simple form to compare the two files *fishlist1* and *fishlist2*.

```
>comm fishlist1 fishlist2 CR
            anchovy
                    basking shark
                    cod
haddock
            halibut
                    lamprey
trigger fish

>
```

The following example shows how to use **comm** to print only those lines common to the two files. This is done by suppressing columns 1 and 2.

```
>comm -12 fishlist1 fishlist2 CR
basking shark
cod
lamprey

>
```

The last example illustrates how to put the contents of a file called *fishlist3* into ASCII code order, using **sort** and to pipe the output into the **comm** utility for comparison with *fishlist1*.

```
>cat fishlist3 CR
trout
parrot fish
cod
pike
trigger fish

>sort fishlist3 | comm - fishlist1 CR
basking shark
                            cod
haddock
lamprey
                parrot fish
                pike
                            trigger fish
                trout

>
```

CUT: print selected fields from a file

INTRODUCTION

This is a short tutorial-style introduction to the **cut** utility, which prints selected columns or fields from a data input. The input may take the form of one or more files, output from another command, or the terminal's standard input.

SYNTAX

```
cut [-s] [-dchar] [-clist | -flist] file ...
```

DESCRIPTION

The options and arguments to **cut** are as follows:

-c*list* identifies the character positions that are to be output from each line. Positions are identified by integers. Position identifiers are separated by commas (,), and ranges of positions are given using the hyphen (-). To specify character positions 15 to 21, 24 and 30, the following argument would be passed to **cut**:

```
-c15-21,24,30
```

-f*list* identifies the fields to be output from each line. Fields are identified by integers. The comma is used to separate identifiers, and ranges are given using the hyphen. To specify fields 3, 6 to 8, and 11, the following argument would be

passed to **cut**:

```
-f3,6-8,11
```

-d*char* identifies the character used to delimit fields. The default is the tab. To set the colon as delimiter, the following argument would be passed:

```
-d':'
```

Note that characters with a special meaning to the shell should be either enclosed in single quotes, or preceded by a backslash (\).

-s is used with the **-f***list* option, to prevent the output of lines that do not contain the field delimiter.

file identifies the file to be processed. Note that one or more files can be specified in the same command line.

EXAMPLES

In the following examples, the > character in bold face represents the system prompt, and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

The first example uses a file called *letters*. It's contents are displayed using the **cat** command. (This utility is described in its own tutorial, in this manual.)

```
>cat letters CR
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ZYXWVUTSRQPONMLKJIHGFEDCBA
zyxwvutsrqponmlkjihgfedcba

>cut -c4-11,16,22 letters CR
DEFGHIJKPV
dfeghijkpv
WVUTSRQPKE
wvutsrqpke

>
```

The next example uses a file called *list*. It's contents are also displayed using **cat**. The field delimiter is the colon.

```
>cat list CR
field 1:field 2:field 3:field 4: field 5
field 1:field 2:field 3:field 4: field 5
field 1:field 2:field 3:field 4: field 5
field 1:field 2:field 3:field 4: field 5

>cut -f1-3,5 -d':' list CR
field 1:field 2:field 3:field 5
field 1:field 2:field 3:field 5
field 1:field 2:field 3:field 5
field 1:field 2:field 3:field 5

>
```

In the next example, the field delimiter is redefined as a space, **-d' '**. This has the effect of altering the output from *list*.

```
>cut -f2-4 -d' ' list CR
1:field 2:field 3:field
1:field 2:field 3:field
1:field 2:field 3:field
1:field 2:field 3:field

>
```

DIFF: file difference identifier

INTRODUCTION

This is a short tutorial-style introduction to the **diff** utility which locates and reports all differences between two files, and indicates how to change the first file so that it is a duplicate of the second.

SYNTAX

```
diff [-efbh] file1 file2
```

DESCRIPTION

The options and arguments to **diff** are as follows:

-e produces a script file of **ed** commands that will amend *file1* so that it duplicates *file2*. Details of how this is used are to be found in the *diff*(1) entry in the *Utilities Reference Manual*

-f produces a script file that turns *file2* into a duplicate of *file1*. This is also described in the *diff*(1) entry of the *Utilities Reference manual*

-b causes trailing blanks (that is, spaces and tabs) to be ignored during the comparison. Other strings of blanks are regarded as being equal

-h causes **diff** to do a fast, but not necessarily complete comparison. The changed sections of text must be small and well separated for this option to work effectively. Note that the **e** and **f**

options are not available when this option is used

file1 specifies the first file to compared

file2 specifies the second file to be compared

In the case of *file1* and *file2*, if - is specified instead of a filename, the standard input is used in the comparison.

Where the two files are identical, **diff** generates no output, and the shell returns the system prompt. Where differences do exist, **diff** prints the affected lines, using the following identifiers:

< highlights the lines in *file1*

> highlights the lines in *file2*

x1,y1 f x2,y2
which means that lines *x* to *y* of *file1* must be changed into lines *x* to *y* of *file2* if the two files are to be the same. The function *f* would here be a letter **c** which stands for *change*. Other functions are **a** for *append* and **d** for **delete**.

--- separates the blocks of text from the two files

EXAMPLES

In the following examples, the **$** symbol in bold is used to represent the system prompt, and the **CR** symbol is used to indicate that the carriage return key is to be pressed in order to enter the command line.

The first example compares two files containing mail-shot letters. The text is the same, but the names and addresses have been changed. The first file is called *jones* and the second is called *spottiswoode*. The output

from **diff** identifies the differences as follows:

```
$diff jones spottiswoode CR
3,6c3,6
< Mr. Sid Jones
< 12 Gas Works Street
< Bootle
< Merseyside
---
> Sir J.K.X.P.J. Spottiswoode
> McGurgenthwerker Park Estate
> Whacket-under-Bottley
> Somerset
9c9
< Dear Mr. Jones:
---
> Dear Mr. Spottiswoode:
```

The first line of output from **diff** is:

```
3,6c3,6
```

This means that to match *jones* with *spottiswoode*, lines 3 through 6 of *jones* must be changed to read the same as lines 3 through 6 of *spottiswoode*. **Diff** command then displays both sets of lines.

After making these changes (using a text editor such as **ed** or **vi**), the *jones* file will be identical to the *spottiswoode* file. Remember that **diff** identifies differences between specified files. To make an identical copy of a file, the **cp** command can be used.

DIFF3: three way differential file comparison

INTRODUCTION

This section is a tutorial introduction to the **diff3** utility which compares three files and outputs pointers to the differences between them. The disagreeing text is printed, along with a code line selected from the following series:

```
====    all three files differ
====1   file1 is different
====2   file2 is different
====3   file3 is different
```

Following this code, is a diagnostic line which indicates the changes that must be made in order to bring the files into agreement. The diagnostic line should be interpreted as follows:

```
f:ln ed
```

where:

f the number of the file as entered on the command line

ln the number of the line that differs. This may be a range

ed the editor command that must be used to bring the file into alignment with the others. If a change (c) operation is indicated, the existing contents of the line are shown

SYNTAX

```
diff3 [-ex3] file1 file2 file3
```

DESCRIPTION

The arguments to **diff3** are as follows:

-e, -x and **-3** these three options publish a shell script for the **ed** editor that will incorporate all the changes needed to bring *file1* into agreement with *file3*. This shell script may be applied directly to a file.

file[1-3] the files to be compared.

EXAMPLES

The first example begins by displaying the contents of three files, called *file-a*, *file-b* and *file-c*. It then goes on to run **diff3** on these three files.

Remember that the symbol **>** in bold represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>cat file-a CR
A
B
C
D
E
```

```
>cat file-b CR
B
C
D
E
A

>cat file-c CR
C
D
E
A
B

>diff3 file-a file-b file-c CR
====
1:1,2c
  A
  B
2:1c
  B
3:0a
====
1:5a
2:5c
  A
3:4,5c
  A
  B

>
```

The next example shows how to compare the same three files, and receive a editor script that will make the changes necessary for *file-a* to agree with *file3*.

```
>diff3 -e file-a file-b file-c CR
5a
A
B
1,2c
w
q

>
```

This shell script can be applied directly to the file being changed. Using a pipe, it is possible to combine the **diff3** operation and the editor operation in the same command line. The third example shows how to combine the three files, and apply the editor script directly to *file-a*.

```
>diff3 -e file-a file-b file-c | ed - file-a CR

>
```

For full details of **ed**, see the tutorial in this manual.

FILE: determines file type

INTRODUCTION

This section is a tutorial introduction to the **file** command which is used to check the contents of a file or files specified on the command line. The first block of each file (1024 bytes) is examined, and the information held there is classified. If a file contains a *magic number*, the file type can be identified when **file** is used. Magic numbers are numeric or string constants added to a file on creation. The meaning of each magic number is stored in the file */etc/magic*. The contents of this file can be read using the **cat** command. An explanation of the data format used is given at the beginning of the file.

Examples of the file types that can be recognised are:

- directory
- empty
- C program text
- ASCII text
- **nroff**, **troff**, **tbl** or **eqn** input text

Note that a file must have read permission before it can be classified. For further details of access permissions, see the **chmod** tutorial in this manual.

SYNTAX

```
file [-c] [-f ffile] [-m mfile] name [name ...[name]]
```

DESCRIPTION

The options and arguments to **file** are as follows:

-c causes **file** to check the format of the */etc/magic* file. No file classification is carried out.

-f *ffile* specifies a file containing the names of files to be classified.

-m *mfile* specifies an alternative magic file to */etc/magic*.

name specifies a file to be classified. Full pathnames must be given, where appropriate.

EXAMPLES

The first example shows **file** being used on a file whose name and path are given. The output consists of the name of the file being examined and its type. Remember that the symbol **>** in bold represents the system prompt, and that **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

```
>file /usr2/spike/manual1/preface CR
/usr2/spike/manual1/preface:    ascii text

>
```

The next example shows how a file containing a list of filenames can be used to automate file classification. The contents of this list file, called *checklist*, are displayed using the **cat** command, then the **file** command line is entered, using this list.

```
>cat checklist CR
/usr2/spike/manual1/preface
/usr2/spike/manual1/contents
/usr2/spike/manual4/preface

>file -f checklist CR
/usr2/spike/manual1/preface:    ascii text
/usr2/spike/manual1/contents:    ascii text
/usr2/spike/manual4/preface:    ascii text

>
```

NL: line numbering filter

INTRODUCTION

This section is a tutorial introduction to the **nl** utility which reads lines from a file or from the standard input and numbers them. The style of numbering is determined by the options and arguments selected.

Input is read by **nl** in terms of logical pages. Each logical page consists of a *header*, a *body* and a *footer*. Sections may be empty. To be recognisable by **nl**, each section must contain an identification code on a separate line:

Section	Identifier
Header	\:\:\:
Body	\:\:
Footer	\:

The default mode of operation assumes that input is read as a single logical page.

SYNTAX

```
nl [-htype] [-btype] [-ftype] [-vstart#] [-iincr]
[-p] [-lnum] [-ssep] [-wwidth] [-nformat] [-ddelim]
file
```

DESCRIPTION

The options and arguments to **nl** are as follows:

-h*type* specifies which lines contained in the header are to be numbered, where *type* is one of

a number all lines.

t number only lines containing printable text.

n no header line numbering.

p*string* number only lines containing the regular expression *string*.

Note that the default *type* for the logical page header is **n**.

-b*type* specifies which lines contained in the body are to be numbered, where *type* takes the same options as **-h***type*, above. The default *type* for the logical page body is **t**.

-f*type* specifies which lines contained in the footer are to be numbered, where *type* takes the same options as **-h***type*, above. The default *type* for the logical page footer is **n**.

-v*start#* initialises the logical page lines. The default starting point is 1.

-i*incr* sets the increment interval for the logical page lines. The default interval is 1.

-p overrides page numbering restart after logical page delimiters.

-l*num* specifies how many consecutive blank lines are to be treated as being one. For example, setting -l2 tells **nl** to assign one line number to every double blank line.

-s*sep* specifies the character to be used as a separator between the line number and the start of the text. The default is a tab.

-w*width* specifies the number of characters used for the line number. the default is 6.

-n*format* specifies the format of the line number, where *format* is one of the following

ln left justified, with no leading zeroes

rn right justified with no leading zeroes

rz right justified with leading zeroes

The default format is **rn**.

-d*delim* sets new delimiter characters. If only one character is set, the second delimiter remains the colon (:). To specify a backslash (\) it must be entered as two backslashes (\\).

file specifies the file to be read by **nl**.

EXAMPLES

In the following example, a text file called *article* is processed by **nl**. The first operation displays the contents of *article* with the **cat** command. For details of this command, see the **cat** tutorial in this manual. Remember that the **>** symbol in bold represents the system prompt, and that **CR** indicates that the carriage return key must be pressed in order to enter the command line.

```
>cat article CR
\:\:\:
This is sample text for the header section
and this is a bit more ...
\:\:
This is the start of the body section
and this is the end
\:
This is a sample for the footer section

>nl -ha -fa -v10 -i5 -nrz -w3 -s\ article CR
010This is sample text for the header section
015nd this is a bit more ...

020This is the start of the body section
025nd this is the end

030This is a sample for the footer section

>
```

Note that the **-ha** and **-fa** caused numbering of the header and footer sections respectively, that the **-v10** and **-i5** set numbering to start at 10 and increase in units of 5, and that the format of the numbers was set to three right justified digits with leading zeroes, separated from the text by a backslash (\).

PASTE: file merge utility

INTRODUCTION

This is a tutorial introduction to the **paste** utility which merges two or more files side by side, as if each were a separate column in a table. If only one file is specified, subsequent lines from that file are merged. Output can be displayed on the standard output, piped into another command, or directed into a file. For vertical file merging (that is, end to end), see details of the **cat** command, which has its own tutorial in this manual.

SYNTAX

```
paste file ...

paste -dlist file ...

paste -s [-dlist] file ...
```

DESCRIPTION

The arguments and options to **paste** are as follows:

file specifies the files to be merged. If a filename is given as -, a line is read from the standard input.

-d*list* defines the delimiters used to separate the merged lines. The default is the tab. The characters contained in *list* are used in rotation until the end of the list, then the list is re-used as required. It is safest to enclose the

list in double quotes. To specify the backslash (\) as the separator, it should be given twice, in the form **-d'\\'**. A space can be specified as the separator using the form **-d' '**. Special characters are defined by escape sequences, as follows:

- for a new-line character
- for a tab character
- for the backslash
- for an empty string.

-s specifies that several lines from each file are to be merged as opposed to just one.

EXAMPLES

The first example begins with the **cat** command which displays the contents of two files, called *mergefile1* and *mergefile2*. It goes on to show these two files being used by the simplest form of **paste**, which merges the two files side by side, placing the default separator (tab) between the two elements. The output is re-directed into a file called *double* by way of the *re-direction indicator*. Remember that the **>** character in bold represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>cat mergefile1 CR
fruit
horse
football
```

```
>cat mergefile2 CR
bat
fly
boot

>paste mergefile1 mergefile2 > double CR

>cat double CR
fruit   bat
horse   fly
football   boot

>
```

Note that *double* now contains three lines made up of the elements of the original two files.

The second example uses the **-d***list* option to specify that the backslash (\) should be used instead of the tab character to separate the text elements.

```
>paste -d'\\' mergefile1 mergefile2 > double CR

>cat double CR
fruit\bat
horse\fly
football\boot

>
```

PG: paging through the contents of a file

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **pg** utility, which allows the contents of a file to be displayed one screen at a time, according to conditions specified by the user.

SYNTAX

```
pg [-number] [-p string] [-cefns] [+line] [+/pattern/]
[files ...]
```

DESCRIPTION

The command **pg** (short for *page*) allows you to examine the contents of a file or files, page by page, on a terminal. The routine begins with **pg** scanning your terminal's **terminfo** database, to determine the attributes of the terminal. This is read from the **TERM** environment environment. (Environment variables are described in the shell tutorials in the *Shell / C Shell X/OS Command Language user Guide.*) This allows the correct display routines to be chosen.

The **pg** command displays the text of a file in pages (chunks) followed by a colon prompt (:), a signal that the program is waiting for your instructions. Possible instructions you can then issue include requests for the command to continue displaying the file's contents a page at a time, and a request that the command search through the file(s) to locate a specific character pattern.

The **pg** command is useful when you want to read a long file or a series of files because the program pauses after displaying each page, allowing time to examine it. The size of the page displayed depends on the terminal. For example, on a terminal capable of displaying twenty-four lines, one page is defined as twenty-three lines of text and a line containing a colon. However, if a file is less than twenty-three lines long, its page size will be the number of lines in the file plus one (for the colon).

The options and arguments to **pg** are as follows:

-number specifies the number of lines in a window. If not specified, the default value is one less than the terminal's maximum.

-p *string* replaces the default prompt (:) with *string*. If *string* is specified as **%d**, the first occurrence of **%d** in the prompt is replaced by the number of the current page being displayed.

-c sets the cursor to the home position (top left corner of the screen), and clears the screen before printing the next page. Note that if **clear_screen** is not set in the **terminfo** database, this option will be ignored.

-e causes **pg** to pass on to the next file without pausing.

-f causes **pg** to refrain from splitting long lines. This is to prevent undesirable results caused by splitting text controlled by special effect commands, for example underlining.

-n causes **pg** to add an automatic end-of-command whenever a recognisable command

letter is typed.

-s causes **pg** to print messages and prompts in standout mode, as opposed to inverse video.

+*line* starts displaying the file contents at the line identified by *line*.

+/*pattern***/** starts displaying the file contents at the first line containing the string *pattern*.

files specifies the files to be examined. If - is specified, **pg** reads from the standard input.

Once the first page of text has been displayed, **pg** pauses, and displays the prompt. Further commands can then be entered. These usually take the form of the command itself, and a preceding address, either signed or unsigned. The address specifies the point at which further examination is to occur. If signed, the starting point is relative to the current position: if unsigned, the starting point is relative to the start of the file. The available commands are as follows:

CR or **space bar**
causes a further page to be displayed. The default address is +1. If an address is specified, **pg** scans forward (+) or backward (-) the appropriate number of pages in order to find the required page

l causes an effect of scrolling, either forward (+) or backward (-), by the number of lines specified in the address. If no address is specified, **pg** scrolls forward one line.

d or **CTRL-d**
scrolls forward (+) or backward (-) by the number of half-screens specified in the

address. If no address is specified, **pg** scrolls forward one half-screen.

. or **CTRL-l**
causes the current page to be redisplayed. This command takes no address.

$ causes the last page in the file to be displayed. This command takes no address.

There are also a number of commands for locating particular text patterns. Note that the regular expressions available to the **ed** utility can be used here. For details, see the **ed** tutorial in this manual. The **pg** pattern searching commands are as follows:

i/pattern/ searches forward for the *i*th occurrence of *pattern*. (The default is 1). The search begins after the current page, and continues until the end of file is reached. Note that the pattern must all occur on one line.

i?*pattern*? or *i*^*pattern*^
searches backward for the *i*th occurrence of *pattern*. (The default is 1). The search begins before the current page, and continues until the start of file is reached. Note that the pattern must all occur on one line. The ^ variant is useful for terminals that do not correctly handle the ?.

The **pg** environment can be altered with the following commands:

*i***n** examine the *i*th next file as listed in the **pg** command line, where *i* is an unsigned integer. The default value of *i* is 1.

*i***p** examine the *i*th previous file as listed in the **pg** command line, where *i* is an unsigned integer. The default value of *i* is 1.

iw display another window of text. If *i* is specified, the window size is set to *i* lines.

s *filename* save the file currently under examination in *filename*. This command must always end with a **CR**, even when the **-n** option is in use.

h display a help screen, consisting of a command list.

q or **Q** quit from **pg**.

!*cmd* escape to the **pg**, to execute the shell command *cmd*. This command must always end with a **CR**, even when the **-n** option is in use.

Note that **pg** output to the screen can be terminated using the **quit** signal, **CTRL-**, or the interrupt signal, **BREAK**. The **pg** prompt is displayed.

Note also that to obtain satisfactory results, terminal tabs should be set every eight positions.

EXAMPLES

In the following examples, the **>** character in bold type represents the system prompt, and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

The first example displays the contents of a file called *outline*. When the command is entered, the first page of the file appears on the screen. Because the file has more lines in it than can be displayed on one page, a colon appears at the bottom of the screen. This is a reminder to you that there is more of the file to be seen. When you are ready to read more, press the **CR** key and **pg** will print the next page of the file.

```
>pg outline CR
After you analyze the subject for your
report, you must consider organising and
arranging the material you want to use in
writing it.
.
.
.
An outline is an effective method of
organiszing the material.  The outline
is a type of blueprint or skeleton,
a framework for you the builder-writer
of the report; in a sense it is a recipe
: CR
```

After you press the **CR** key, **pg** will resume printing the file's contents on the screen:

```
that contains the names of the
ingredients and the order in which
to use them.
.
.
.
Your outline need not be elaborate or
overly detailed; it is simply a guide you
may consult as you write, to be varied,
if need be, when additional important
ideas are suggested in the actual writing.
(EOF):
```

Notice the line at the bottom of the screen containing the string *(EOF)*. This expression means you have reached the end of the file. The colon prompt is a cue for you to issue another command.

When you have finished examining the file, press the **CR** key; a prompt will appear on your terminal. (Typing **q** or **Q** and pressing the **CR** key also gives you a prompt.) Or you can use one of the other available commands, depending on your needs.

The next example uses the **-p** option to change the **pg** prompt.

```
>pg -p "[page %d]:" outline CR
After you analyze the subject for your
report, you must consider organising and
arranging the material you want to use in
writing it.
.
.
.
An outline is an effective method of
organising the material.  The outline
is a type of blueprint or skeleton,
a framework for you the builder-writer
of the report; in a sense it is a recipe
[page 1]:
```

To view the next page, you would press **CR**, as before.

PR: print files

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **pr** utility, which prints files on the standard output, usually the screen. Files are displayed a page at a time, each page headed by a page number, the date and time, and the name of the file.

SYNTAX

---

pr [options] [name ...]

DESCRIPTION

The **pr** utility is particularly useful when used with a line printer. (See the **lp** tutorial in the *Advanced Utilities User Guide*.) The files identified by *name* are printed according to the following options. Note that this is a partial list, and that full details may be obtained from the *pr*(1) entry of the *Utilities Reference manual*.

+k begin printing at page *k*. The default is 1.

-k print the output in *k* columns. The default is 1.

-a print the output in multi-column format across the page.

-m merge and print all specified files, with one column for each file. This overrides the **-a** and **-k** options.

-d double-space the output.

-w*k* sets the width of a line to *k* character positions. The default is 72 for equal-width, multi-column output.

-o*k* offset each line by *k* character positions. The default is 0. The true width of a line is the **-w***k* setting minus the offset.

-l*k* sets the page length to *k* lines. The default is 66.

-h uses the next argument as the header, rather than the filename.

-p pauses before the beginning of a new page.

EXAMPLES

In the first example, a file called *textfile.txt* is printed, using none of the available options. Accordingly, **pr** produces output in a single column that contains sixty-six lines per page and is preceded by a short heading. The heading consists of five lines: two blank lines; a line containing the date, time, file name, and page number; and two more blank lines. The formatted file is followed by five blank lines.

Remember that the **>** character in bold type represents the system prompt, and that **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

```
>pr textfile.txt CR

Aug 31 15:43 1987 textfile.txt Page 1


August 31, 1987

Memo:    Documentation Project

cc:      spike, sue

The documentation project is to consist of 11 volumes, listed
in the attached document. A meeting will be held on Thursday
to finalise such details as titles and syntax layouts. Manual
codes will be available at the meeting.

Norman Bates
.
.
```

The ellipses after the last line in the file represent the remaining lines (all blank in this case) that **pr** formatted into the output so that each page contains a total of 66 lines.

Note that 66 lines will not fit onto a standard video display terminal, which usually displays 24 lines at a time. The entire 66 lines of the formatted file are printed rapidly without pause. This means that the first 42 lines roll off the top of the screen, making it impossible to read them unless one of two measures is taken.

The first is to type **CTRL-s** to interrupt the flow of output, then **CTRL-q** to resume. The second is to change the parameters passed to **pr**. The second example alters the page length to 24 lines, and tells **pr** to pause before

displaying the next page.

```
>pr -124 -p textfile.txt  CR
```

SORT: sorting and merging files

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **sort** utility, which will sort the contents of an input file, or text entered on the standard input. It will also merge the contents of two or more files, sorting them at the same time. A variety of sort and merge criteria can be specified by the user.

SYNTAX

```
sort [-cmu] [-ooutput] [-ykmem] [-zrecsz] [-dfiMnr]
[-btx] [+pos1 [-pos2]] [file ...]
```

DESCRIPTION

The contents of one or more files (or text entered on the standard input) are sorted according to the following default criteria:

- lines beginning with a number are sorted by value, and appear before those lines beginning with a letter

- lines beginning with upper case letters are sorted alphabetically, and appear before those beginning with a lower case letter

- lines beginning with special characters such as **%**, **&** and **@**, are sorted according to their ASCII value. The full range of ASCII characters, and their numerical values in octal and hexadecimal, is given in the *ascii*(5) entry of the *System Interfaces and Libraries Reference manual*.

The sorting is carried out according to one or more sort keys, extracted from each line of input. In the event of these not being explicitly stated, the default is used, that is, the the key is taken to be the whole line.

The options and arguments to **sort** are as follows:

-c checks that the input file is sorted according to the ordering rules specified by the user. This option will give no output unless the file is out of sort.

-m merges only. The input files are already sorted.

-u suppress all but one in each set of lines that have equal sort keys.

-o*output* specifies the output file to use instead of the standard output. This file may be the same as one of the inputs.

-y*kmem* specifies *kmem* number of kilobytes of memory space in which to carry out the sort. This can improve the efficiency of the operation. See the *sort*(1) entry in the *Utilities Reference Manual* for further details.

-z*recsz* specifies the length of the longest line to be sorted and merged. This can prevent abnormal termination of the program under certain circumstances. Again, see the *sort*(1) entry in the *Utilities Reference Manual* for further details.

Using the following options, it is possible to override the default sorting rules:

-d uses *dictionary* order: only letters, digits, and blanks (spaces and tabs) are significant in comparisons.

-f merges text by letter, irrespective of case.

-i ignores characters outside the ASCII range 21-7e (Hex) in non-numeric comparisons. (Characters 21-7e are printable characters.)

-M compares the first three non-blank characters in the order *JAN* to *DEC*. Invalid fields are regarded as of lower order than *JAN*. The **-M** option implies the **-b** option (see below).

-n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. The **-n** option implies the **-b** option (see below).

-r reverses the sense of comparisons.

In some cases, it will be necessary to specify a part of the input line as the sort key, rather than the whole line. This is performed using the following arguments:

+pos1 specifies the start of the sort key. It takes the form of an integer, representing the character position in the line.

-pos2 specifies the end of the sort key. Where this argument is missing, the end of the line is assumed. It takes the form of an integer, representing the character position in the line.

The use of *+pos1* and *-pos2* implies the use of discrete fields. A field in this context, is taken be a sequence of characters, separated by a field delineator or a newline character. In the case of an input file containing names and telephone numbers, the separate fields would be the names and the addresses, and the delineators might be spaces. The default field separator is the first of one or more spaces encountered between the fields.

Using the following arguments, it is possible to specify the exact nature of the field separator.

-t*x* uses *x* as the field separator character; *x* is not considered to be part of the field (although it may be included in a sort key). Each occurrence of *x* is significant (that is, *xx* specifies an empty field).

-b ignores leading blanks when determining the starting and ending positions of a field.

For more details of how **sort** handles fields, see the *sort*(1) entry in the *Utilities Reference manual.*

Note that where more than one sort key is specified on the command line, later keys are used only where earlier keys return equal values for two or more input lines.

EXAMPLES

In the following examples, the > symbol in bold face represents the system prompt, while **CR** indicates that the carriage return key should be pressed in order to enter the command line.

In the first example, a single file, called *input1* is to be sorted. The first command line in the example displays its contents, using the **cat** command, which is explained in more detail in its own tutorial chapter in this manual. The second command sorts *input1* according to the default criteria listed above.

```
>cat input1 CR
Turner, D; 238-5540
Bates, M; 602-6490
Williams, F; 338-3877
Lawrence, L; 330-4429
Simmonds, V; 237-8732
```

```
>sort input1 CR
Bates, M; 602-6490
Lawrence, L; 330-4429
Simmonds, V; 237-8732
Turner, D; 238-5540
Williams, F; 338-3877

>
```

The second example takes two files, called *input1* and *input2*, and after sorting, interleaves the two lists into one. The contents of the two files are displayed using the **cat** command on each.

```
>cat input1 CR
Turner, D; 238-5540
Bates, M; 602-6490
Williams, F; 338-3877
Lawrence, L; 330-4429
Simmonds, V; 237-8732

>cat input2 CR
Smith, A; 553-7832
Jones, B; 996-3386
Cook, K; -
Moore, P; 654-9429

>sort input1 input2 CR
Bates, M; 602-6490
Cook, K; -
Jones, B; 996-3386
```

```
Lawrence, L; 330-4429
Moore, P; 654-9429
Simmonds, V; 237-8732
Smith, A; 553-7832
Turner, D; 238-5540
Williams, F; 338-3877

>
```

In the third example, the same two files are sorted by telephone number. The default field separator, the space, is used. Note that because the sort key is the second field of two, there is no need to specify the end point.

```
>sort +2 inputl input2 CR
Cook, K; -
Simmonds, V; 237-8732
Turner, D; 238-5540
Lawrence, L; 330-4429
Williams, F; 338-3877
Smith, A; 553-7832
Bates, M; 602-6490
Moore, P; 654-9429
Jones, B; 996-3386

>sort -ooutputl +2 inputl input2 CR

>cat outputl CR
Cook, K; -
Simmonds, V; 237-8732
Turner, D; 238-5540
Lawrence, L; 330-4429
Williams, F; 338-3877
Smith, A; 553-7832
Bates, M; 602-6490
Moore, P; 654-9429
Jones, B; 996-3386
```

```
>
```

The third example ended by repeating the sort and merge operation on the two files, but this time, redirecting the output away from the screen, and into a file called *output1*. The **cat** command was then used to confirm that the operation was successful.

SPELL: the spelling check utilities

INTRODUCTION

This chapter is a tutorial-style introduction to the X/OS **spell** utilities. Text files are run through **spell** to check for spelling mistakes. Words are compared with a dictionary database, and unrecognised words are highlighted.

This chapter begins with the **spell** utility before going on to describe some of the supporting routines and files.

SYNTAX

```
spell [-v] [-b] [-x] [-l] [-i] [+local_file]
[files ...]

/usr/lib/spell/hashmake

/usr/lib/spell/spellin n

/usr/lib/spell/hashcheck spelling_list
```

DESCRIPTION

The **spell** utility can be used with or without arguments. If no arguments are given, individual words or lists of words can be typed in for checking. Each word in the list is terminated using the **CR** key, and the list is terminated using the **CTRL-d** sequence. As soon as the list is complete, **spell** checks each word entered, and unrecognisable words are displayed. Note that the words displayed by **spell** are not necessarily misspelled: they are also listed if not known to the system, and not

derivable from recognised words by the application of inflections, prefixes, and/or suffixes.

The arguments to **spell** are as follows:

-v causes all words not *literally* in the spelling list to be output, along with plausible derivations.

-b uses British spelling, that is, prefers spellings such as *colour*, *centre*, *programme*, and endings in *-ise* as opposed to *-ize*.

-x causes all words in the input file to be output, preceded by an equals sign (=). Unrecognised words are then listed a second time, this time, without the equals sign.

-l checks *all* files linked to the file with the **troff** macros **.so** or **.nx**. See the *troff*(1) entry in the *Utilities Reference Manual* for details. The default operation is to check all linked files *except* those whose filenames begin with */usr/lib*.

-i ignores all chains of included files.

+*local_file* *local_file* is a supplementary list of user-supplied words that are to be recognised by **spell**. This allows commonly used technical terms, real names, and abbreviations to be added to the **spell** database.

files identifies the input text files.

EXAMPLES

This section covers the commonly used **spell** operations. Throughout the examples given, the symbol **>** in bold face represents the system prompt, while **CR** indicates that the carriage return key should be pressed in order to enter the command line. The symbol **CTRL-d** means that the key marked **CTRL** or **CONTROL** should be held down while the letter **d** key is typed.

In the first example, **spell** is used to check a list of words entered in the form of a list. Only one entry is misspelled. This is returned by the system at the end.

```
>spell CR
displayed CR
library CR
dictionary CR
msitake CR
output CR
CTRL-d
msitake

>
```

In the next example, a text file is used as input. The input file is called *text.txt*. The first command line in the example uses the **cat** utility to examine its contents.

```
>cat text.txt CR
What is gesamtkunstwerk?
The concept of a total integration in artistic
preformance, so that the elemnets of music, drama
and spectacle are interdependent to the extent
that none dominates.
```

```
>spell text.txt CR
gesamtkunstwerk
preformance
elemnets

>spell text.txt > errors CR

>
```

Three words were found which **spell** did not recognise. In this case, two are genuine typing errors, while one is correctly spelled, but too specialised to have been in the original **spell** library database. The last command line redirects the output from **spell** into a file for later use.

The next example uses the **-v** option. This checks the words in the input file, and highlights their derivation from words literally present in the library database. The incorrectly spelled words are also listed.

```
>spell -v text.txt CR
elemnets
gesamtkunstwerk
+ist+ic     artistic
+s          dominates
-e+ion      integration
+inter      interdependent
+pre-t+ce   preformance

>
```

From the above, it can be seen that the dictionary contains the words *art*, *dominate*, *integrate*, *dependent* and *formant*, and adds or subtracts certain prefixes and suffixes in order to derive the words in the input file. Note that although *preformance* was a typing error, **spell** is able to recreate it, by coincidence, from another word

and a collection of prefixes and suffixes.

In the next example, the **-b** option is used to specify that British spelling should be used. The first command line uses **cat** to display the contents of a file called *newfile*. This file is then run through **spell**, and the British spellings are listed as errors. When **spell** is re-run, using the **-b** option, no errors are listed.

```
>cat newfile CR
Most television programmes look better in colour.

>spell newfile CR
programmes
colour

>spell -b newfile CR

>
```

The next example uses the **-x** option. Each recognisable word is listed, and where the word is inflected (that is, there is a suffix or prefix added to a root), the root is also listed.

```
>spell -x text.txt CR
=and
=are
=art
=artist
=artistic
=concept
=dependent
=dominate
=dominates
.
.
=What
```

```
=what
elemnets
gesamtkunstwerk
```

Note that the word *artistic* is listed along with its two roots, *art* and *artist*. Words appearing with a capital letter in the input file are listed with the first letter in upper and lower case. Finally, the unrecognisable words are listed.

The next example uses the **-l** option to check on linked files. The first two command lines display the contents of two files. The first is called *textfile*. It contains a single line of text, and a pointer to a second file called *linkfile*. This second file also contains a single line of text.

```
>cat textfile CR
.so linkfile
A sample line of txt

>cat linkfile CR
Anothr sample line of text

>spell textfile CR
txt

>spell -l textfile CR
txt
Anothr

>
```

The first **spell** command line used no options, and so only the one spelling mistake was listed. Note that **spell** will ignore **troff**, **tbl** and **eqn** constructions.

When the -l option was used, **spell** checked *textfile*, listed the spelling mistake found there, then followed the **.so** pointer to *linkfile*, and listed its spelling mistake.

On at least one occasion in these examples, a word that is correctly spelled is listed as being unrecognisable. The *+local_file* option allows the user to set up a supplementary library file that contains words and abbreviations used frequently, but not present in the standard **spell** library database.

Note that when creating a *local_file*, words appearing totally in upper case must appear at the beginning of the file.

In the following example, the file *text.txt* is again used. The first command line is a reminder of this file's contents. The second runs **spell** on the file, generating a list of three unrecognised words. Two are genuine spelling mistakes, while the third is simply not known to **spell**.

```
>cat text.txt CR
What is gesamtkunstwerk?
The concept of a total integration in artistic
preformance, so that the elemnets of music, drama
and spectacle are interdependent to the extent
that none dominates.

>spell text.txt CR
gesamtkunstwerk
preformance
elemnets

>cat update.lib CR
De Stijl
acmeism
gesamtkunstwerk
suprematism
```

```
rayonism

>spell +update.lib text.txt CR
preformance
elemnets

>
```

The third command line listed the contents of a file called *update.lib*, created by the user, and containing a number of specialist terms. When run using the library update file, **spell** listed only the genuine spelling mistakes.

The next section of this chapter describes the working of some of the **spell** administrative files. The process of *hashing* is that of coding entries in the **spell** dictionary. Each word is represented by a unique 9-digit *hashcode*. The advantage of coding entries in this way is that **spell** can operate more quickly.

/usr/lib/spell/hashmake generates a unique hashcode for each word.

/usr/lib/spell/spellin n reads *n* hashcodes, as generated by *hashmake*, from the standard input, and compresses them into a spelling list on the standard output. This program is used to create a new spell list, or to extend an existing one.

/usr/lib/spell/hashcheck reads the compressed hashcodes, as generated by *hashmake* and compressed by *spellin*, and decompresses them.

SPLIT: splits a file

INTRODUCTION

This is a tutorial introduction to the **split** utility, which reads a file and uses it to create a series of smaller files, each of a specified number of lines. The default is 1000 lines per file. The name of the output file is specified on the command line, up to a maximum of 12 characters in length. To this filename, **split** assigns a sequence of extensions beginning *aa*, *ab*, and ending at *zz*. In this way, the maximum number of files that can be created using **split** is 676. The default output filename is *x*.

SYNTAX

```
split [-n] [filein [fileout]]
```

DESCRIPTION

The arguments and options to **split** are as follows:

-n the number of lines to be written to the output file (default 1000).

filein the name of the input file to be split. The input file is not altered. If no filename is given, or if *filein* is specified as -, the standard input is used.

fileout the name of the output files (default *x*)

EXAMPLES

The example given here uses **split** to divide a source file called *clathrate.txt* into three short files, each of two lines. These are assigned the default output filename *x*. The first command in the sequence displays the contents of the input file. For details of the **cat** command, see the **cat** tutorial in this manual. The next command is the **split** operation that produces the three two-line output files. These are listed using the **ls -l** command. Finally, the contents of the first output file is displayed. Remember that the **>** symbol in bold represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>cat clathrate.txt CR
What is a clathrate?
A clathrate is a compound in which the first molecular
component is held in place by the cage-like structure
of the second. The behaviour of clathrates is intermediate
between that of normal compounds and that of simple
mixtures.

>split -2 clathrate.txt CR

>ls -l CR
total 5
-rw-r--r--  1 spike  GRP2  253  July 21 14:43  clathrate.txt
-rw-r--r--  1 spike  GRP2   76  July 21 14:55  xaa
-rw-r--r--  1 spike  GRP2  114  July 21 14:55  xab
-rw-r--r--  1 spike  GRP2   63  July 21 14:55  xac

>cat xaa CR
What is a clathrate?
A clathrate is a compound in which the first molecular

>
```

TAIL: print the last part of a file

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **tail** utility, which prints out the end of a file, from a point specified by the user.

SYNTAX

```
tail [+-[number] [lbc[f]]] [file]
```

DESCRIPTION

The options and arguments available are as follows:

number copying begins *+number* lines after the beginning of the file, or *-number* lines before the end of the file. If no *number* is given the default value of 10 is assumed.

l specifies that *number* is to be measured in lines.

b specifies that *number* is to be measured in blocks.

c specifies that *number* is to be measured in characters.

If none of these is specified, **tail** assumes lines.

f where the input is not a pipe (see the shell tutorials in the *Shell / C Shell X/OS Command*

Language User Guide for details), **tail** enters an endless loop consisting of a display operation followed by a short sleep period. In this way, **tail** can be used to monitor any command that periodically appends material to a file.

file specifies the file to be displayed. If no *file* is specified, **tail** uses the standard input, that is, in most cases, input from the keyboard.

Note that the **f** option will not work if the input takes the form of a pipe operation. Note also that, as with other X/OS utilities, the *file* argument can be either a filename or a full pathname.

EXAMPLES

In the following examples, **tail** is used to display various portions of a file called *phones.lst*. Remember that the **>** symbol in bold face represents the system prompt, and that **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

The first example begins by using the **cat** command to display the full contents of *phones.lst*. The second command line uses **tail** to display the last five lines.

```
>cat phones.lst CR
spike  377622 ext 235
sue    644298 ext 521
anne   238809 ext 42
freya  448022
peter  388837 ext 210
chris  377622 ext 239
anna   775140
tom    225346
```

```
>tail -5 phones.lst CR
freya  448022
peter  388837 ext 210
chris  377622 ext 239
anna   775140
tom    225346

>tail +24c phones.lst CR
ue    644298 ext 521
anne   238809 ext 42
freya  448022
peter  388837 ext 210
chris  377622 ext 239
anna   775140
tom    225346

>
```

The last command line in the example used **tail** to display the file from a point 24 characters from the beginning of the file. Note that spaces and new line characters are included in the character count.

TEE: pipe fitting utility

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **tee** utility, which transcribes the standard input to the standard output, and creates copies in files.

SYNTAX

```
tee [-i] [-a] [file ...]
```

DESCRIPTION

The **tee** utility acts as a pipe, but is able to make a copy of the data passing through it in a named file, rather than acting only as a passage from one process to another. To use **tee**, data must be piped into it and out of it.

The options and arguments available to **tee** are as follows:

-i causes **tee** to ignore interrupts.

-a causes output from **tee** to be appended to the named files rather than over-writing them.

file identifies the files to contain the output from **tee**.

EXAMPLES

The following example sets up a **tee** string, where the input comes from **tr** utility. The example begins with the **cat** command used to display the contents of a file called *chl*. The **tr** utility is used to translate all the lower-case characters in the text file into upper case characters. The output from this operation is then piped into **spell**, which checks the text file for typing errors and spelling mistakes. These are printed, along with the derivation of words not literally occurring in the **spell** dictionary. However, the output from **tr** passes through an intermediate stage, that of the **tee** mechanism.

The **tee** channels the output from **tr** into **spell**, but makes a copy first, calling it *chl.out*. At the end of the command line, the **cat** command displays the contents of the new file created by **tee**.

```
>cat chl CR
What is geophyscs?
The inter-disciplinary sicence that applies
the techniques and theories of physics to the
study of the atmosphere, surfface and interior
of the erth.

>tr "[a-z]" "[A-Z]" < chl | tee chl.out | spell -v; cat chl.out CR
ERTH
GEOPHYSCS
SICENCE
SURFFACE
-y+ies  APPLIES
+s      PHYSICS
+s      TECHNIQUES
-y+ies  THEORIES
WHAT IS GEOPHYSCS?
THE INTER-DISCIPLINARY SICENCE THAT APPLIES
THE TECHNIQUES AND THEORIES OF PHYSICS TO THE
STUDY OF THE ATMOSPHERE, SURFFACE AND INTERIOR
OF THE ERTH.
```

Full details of the **cat**, **tr** and **spell** utilities are available in this manual.

TEST: expression evaluation utility

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **test** utility, which evaluates an expression, and returns a status code of either true or false. The expressions evaluated by **test** relate to the status of files and strings, and the equality of integers.

SYNTAX

```
test expr
[ expr ]
```

DESCRIPTION

After it has evaluated an expression, **test** passes an exit code of 0 if the expression is true, and non-zero if the expression is false. A non-zero code is also returned if **test** has no arguments. This is useful when **test** is executed from within a shell script, especially useful when it is the first entry in a command list following an **if** or **while** statement. These are explained in the shell tutorials in the *Shell / C Shell X/OS Command Language User Guide*.

The expressions are constructed from the following components:

-r *file* true if *file* exists, and is readable

-w *file* true if *file* exists, and is writable

-x *file*	true if *file* exists, and is executable
-f *file*	true if *file* exists, and is a regular file
-d *file*	true if *file* exists, and is a directory
-c *file*	true if *file* exists, and is a character special file
-b *file*	true if *file* exists, and is a block special file
-p *file*	true if *file* exists, and is a named pipe
-u *file*	true if *file* exists, and its set-user-ID bit is set
-g *file*	true if *file* exists, and its set-group-ID bit is set
-k *file*	true if *file* exists, and its sticky bit is set
-s *file*	true if *file* exists, and has a size greater than zero
-t *[fildes]*	true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device
-z *s1*	true if the length of string *s1* is zero
-n *s1*	true if the length of string *s1* is non-zero
s1=*s2*	true if strings *s1* and *s2* are identical
s1!=*s2*	true if strings *s1* and *s2* are not identical
s1	true if string *s1* is not the null string

n1 op n2 true if the integers *n1* and *n2* show the relationship described by *op*. Any of the following may be used:

-eq equal to

-ne not equal to

-gt greater than

-ge greater than or equal to

-lt less than

-le less than or equal to

These primary components may be combined using the operators

! unary negation operator (that is, reverses the effect of the following components)

-a binary AND operator (that is, combines two components)

-o binary OR operator (that is, the two components are exclusive)

( ) used for grouping components. These are meaningful to the shell, and must be escaped. Details of the shell's escape sequences are available in the *Shell / C Shell X/OS Command Language User Guide*.

When the second version of **test** is used, that is, where the expression is enclosed in square brackets, the spaces between the brackets and the expression are compulsory. Note that

[*expression*]

and

test *expression*

have the same effect.

EXAMPLES

In the first example, a procedure called *checkfile* is used. The first command line displays the contents of *checkfile*, using the **cat** command, which has its own tutorial in this manual.

Remember that the > symbol in bold face represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>cat checkfile CR
if test -d $1
  then echo $1 is a directory
elif test -f $1
  then echo $1 is a file
else echo $1 does not exist
fi

>chmod u+x checkfile CR

>checkfile test.txt CR
test.txt is a file

>
```

The first two lines of *checkfile* state "accept a filename in place of **$1**. If it is a directory, print a message saying so". The next two lines say "if, on the other

hand, it is an ordinary file, print a message saying this instead". The last two lines conclude by saying "if it is neither a file nor a directory, print a message telling that it does not exist". Remember that **test -d** checks to see whether a name represents a directory (true or false), while **test -f** checks to see whether it represents an ordinary file (true or false).

The **$1** is one of the shell's ways of accepting information that is not already known. In this way, any filename can be entered. The second command line then uses the **chmod** utility to make *checkfile* executable. The **chmod** utility is described in its own tutorial in this manual. The third command line executes the program, and substitutes the filename **test.txt** for **$1**. In this case, *test.txt* was an ordinary file, so the **-d** returned the value for *false*, and **-f** returned *true*.

TR: translates characters

INTRODUCTION

This is a tutorial introduction to the **tr** utility which filters characters according to options entered on the command line. Characters can be mapped into other characters or deleted. One or more strings can be entered to define the input and output, and options may be selected that transform the input string according to set rules.

Strings and transformations may be specified according to the following:

- repeated occurrences of a character can be reduced to a singe occurrence.

- characters may be identified as ASCII values or octal codes. Octal codes are identified by a preceding backslash (\).

- character ranges may be specified according to the following metasyntax:

    - ranges of characters are identified by enclosing them in square brackets. For example, *[x-z]* specifies the letters *x*, *y* and *z*. The entire range of ASCII alphabetical characters can be specified using *"[a-z]""[A-Z]"*.

    - multiple occurrences of the same character can be specified using the notation *[x*n]* where *x* is the character to be identified, and *n* is the number of occurrences. The number *n* is considered to be an octal value if it begins with a 0, and decimal if it begins with any digit other than a 0.

SYNTAX

```
tr [option] [string1 [string2]]
```

DESCRIPTION

The arguments and options to **tr** are as listed below. In general, where *string1* is present, it identifies the characters that will provide the input to **tr** . Where *string2* also exists, the characters in *string1* are translated into the corresponding characters in *string2* .

string1 identifies the input characters to be operated on by the command line.

string2 when this option is present, the characters in *string1* are mapped to the corresponding characters in *string2.*

option this argument should consist of one or more of the following, which may be specified in any order:

-c the normal operation of **tr** is to map the characters of *string1* into those of *string2.* The *-c* option reverses this operation, and states that the characters of *string1* are *not* to be translated into *string2.* Instead, they pass unaltered to the output.

-d specifies that the characters contained in *string1* are to be deleted from the output. The *string2* argument is not used with this option. If it exists, it will be ignored.

-s causes multiple occurrences of a character to be translated into a single occurrence.

Note that no provision is made within the syntax of the command for the specification of filenames. Where it necessary to supply input in the form of a file, or to direct output into a file, the direction indicators (< for input and > for output) can be used. Use of these indicators is illustrated in the examples, below.

EXAMPLES

The first example uses the simplest form of the command to translate lower case letters into upper case. This is done by specifying the full range of lower case characters as *string1*, and the full range of upper case characters as *string2*. The input is taken from a file called *gamma.txt* and the output is directed into a file called *GAMMA.TXT*. The command sequence begins by using the **cat** command to display the contents of *gamma.txt*.

The symbol > in bold is used to represent the system prompt, and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>cat gamma.txt CR
Gamma rays are a form of electromagnetic rays
emitted as photons during radioactive decay.

>tr "[a-z]" "[A-Z]" < gamma.txt > GAMMA.TXT CR

>cat GAMMA.TXT CR
GAMMA RAYS ARE A FORM OF ELECTROMAGNETIC RAYS
EMITTED AS PHOTONS DURING RADIOACTIVE DECAY.

>
```

The next example uses the **-s** option to translate multiple occurrences of the space character into a single space. The command uses input from a file called *redshift.txt* and directs the output into a file called *redshift.out*.

```
>cat redshift.txt
What  is the    Red  Shift?
A displacement   towards  red, found in the    spectral lines
of  distant   galaxies and stars.  Interpreted as   a Doppler
Effect, implying the     expansion of the          Universe.

>tr -s " " < redshift.txt > redshift.out CR

>cat redshift.out CR
What is the Red Shift?
A displacement towards red, found in the spectral lines
of distant galaxies and stars. Interpreted as a Doppler
Effect, implying the expansion of the Universe.

>
```

The last example shows how **tr** can be used to split a file into lines, each consisting of a single word. This sort of operation is often useful before a file is sorted, for example when compiling a dictionary from text files. The newline character is inserted between each word.

```
>cat gamma.txt CR
Gamma rays are a form of electromagnetic rays
emitted as photons during radioactive decay.

>tr -cs "[A-z]" "[\012*]" < gamma.txt > gamma.out CR
```

```
>cat gamma.out CR
Gamma
rays
are
a
form
of
electromagnetic
.
.
```

UNIQ: searches for repeated lines

INTRODUCTION

This is a tutorial introduction to the **uniq** utility , which reads an input file, and compares adjacent lines. Duplicates are removed, and the processed text is written to the output. Note that duplicated lines must be adjacent to be located. The input and output files must have different names.

SYNTAX

```
uniq [options [-n] [+n]] [filein [fileout]]
```

DESCRIPTION

The arguments and options to **uniq** are as follows:

options may take the form of one or more of the following:

-u lines that are not duplicated in the input are written to the output.

-d only one copy of the duplicated line is written to the output.

-c writes all lines to the output, and generates a count of how often a line occurs. Note that this option supercedes the **-u** and **-d** options.

-n ignores *n* fields in a line before beginning comparison. A field is here

defined as a string of non-space, non-tab characters separated by spaces or tabs from its neighbours. For example, this option would allow a line count field occurring at the beginning of each line to be ignored, so that following text only could be checked for duplication.

+*n* ignores *n* characters in a line before beginning comparison. Note that fields are skipped before characters.

filein specifies the input file.

fileout specifies the output file.

EXAMPLES

The first example illustrates a likely sequence of commands, beginning with the **cat** command, which is used to display the contents of a file called *objects*. This file contains a list of miscellaneous items in random order. The **sort** command is then used to order the list items into a file called *objects.abc*. The **cat** command is then used to display the list as it appears in alphabetical order. **Uniq** is then run on *objects.abc* to strip out duplicated items, and the resulting file, called *objects.out*, is viewed, again using **cat**.

Remember that the > symbol in bold represents the system prompt, and that **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

```
>cat objects CR
horse shoes
brick bats
lemon trees
```

```
koala bears
lemon trees
gear boxes
fish
horse shoes

>sort objects > objects.abc CR

>cat objects.abc CR
brick bats
fish
gear boxes
horse shoes
horse shoes
koala bears
lemon trees
lemon trees

>uniq -c objects.abc objects.out CR

>cat objects.out CR
1 brick bats
1 fish
1 gear boxes
2 horse shoes
1 koala bears
2 lemon trees

>
```

The second example runs **uniq** on a file called *things*. This file contains a list of objects with a line count field built in. The **-n** option is used to strip out this numbering before comparing the lines.

```
>cat things CR
001 lemon trees
002 brick bats
003 brick bats
004 gear boxes
005 koala bears
006 koala bears
007 horse shoes

>uniq -1 things thing.out CR

>cat things.out CR
001 lemon trees
002 brick bats
004 gear boxes
005 koala bears
007 horse shoes

>
```

Note that the **-1** option forced **uniq** to skip the first *field*, that is the **001** to **007** element of the line count. Once this element had been skipped, **uniq** executed using its default operation.

Note that tutorials on both the **cat** and **sort** commands can be found in this manual.

WC: count lines, words, and characters

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **wc** utility which assesses the size of a file in terms of words, lines and characters.

SYNTAX

```
wc [-lwc] [file ...]
```

DESCRIPTION

The default operation performed by **wc** is to return a count of lines, words and characters found in a file, or on the standard input. This behaviour can be modified according to the following options and arguments:

-l provides a count of lines only.

-w provides a count of words only.

-c provides a count of characters only.

Note that these can be combined to specify any sub-set of the three counts.

file specifies the file to be processed. More than one filename may be given, in which case, the required counts are identified by the name of each file. If no *file* parameter is given **wc** performs a count on the standard input.

EXAMPLES

In the following example, a small text file is checked. Remember that the **>** symbol in bold type represents the system prompt, and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

The example begins by displaying the contents of a file called *file1.txt*. It then executes **wc** to generate a count of lines, words and characters. The second **wc** command line uses options to generate a count of only the lines and words.

```
>cat file1.txt CR
This is a simple text file consisting
of three short lines. It will be used
to illustrate the wc utility.

>wc file1.txt CR
   3    20    106 file1.txt

>wc -lw file1.txt CR
    3    20 file1.txt

>
```

DATA DISPLAY
COMMANDS

INTRODUCTION

This third chapter contains five tutorials, covering a miscellaneous range of commands which can be used to display various items of information on the screen, for example the date and time, a calendar, or a reminder service.

The commands covered are as follows:

BANNER the poster utility

CAL prints a calendar

CALENDAR the reminder service utility

DATE prints and sets the date and time

ECHO echoes text on the screen

These tutorials are arranged in alphabetical order.

BANNER: poster utility

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **banner** utility. It accepts a string of characters from the user, and prints them in large letters on the standard output, that is, the screen.

SYNTAX

```
banner strings
```

DESCRIPTION

The *strings* argument to **banner** is the text that is to printed in large letters. Each string may consist of up to 10 characters, not counting spaces. Individual words are printed on separate lines unless joined together using double quotes.

EXAMPLES

In the following example, the words *Good Morning to you, Sydney* are printed. The **>** symbol in bold face represents the system prompt, and **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

In the first example, the three words are printed on separate lines. The first line of the output is shown.

```
>banner Good Morning to you, Sydney CR

 ######                                #
#     #                                #
#          ######    ######    ########
#  ####   #      #  #      #  #      #
#     #   #      #  #      #  #      #
 ######    ######    ######    ########
```

The second example shows the command line that would print the words *to you* on the same line.

```
>banner Good Morning "to you," Sydney CR
```

The third example uses the greater than sign (**>**) to redirect the banner output into a file called *greetings*.

```
>banner Good Morning "to you," Sydney > greetings CR

>
```

CAL: print calendar

INTRODUCTION

This short chapter is a tutorial introduction to the **cal** calendar command. A calendar for the specified year is printed. Individual months can be stipulated.

SYNTAX

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

DESCRIPTION

The arguments to the **cal** command are as follows:

month optionally specifies the month to be printed. It must take the form of a number ranging from 1 to 12.

year optionally specifies the year to be printed. It must take the form of a number ranging from 1 to 9999.

If no arguments are given, **cal** prints the calendar of the current month.

EXAMPLES

This example shows how to obtain a calendar for October 1917. Remember that the **>** in bold represents the system prompt, and that the **CR** symbol indicates that the carriage return key is to be pressed in order to enter the command line.

```
>cal 10 1917 CR
     October 1917
  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

>
```

CALENDAR: reminder service

INTRODUCTION

This section is a tutorial introduction to the **calendar** on-line reminder service. The command accesses a file held in the current directory called *calendar*, and prints all lines containing today's and tomorrow's date. The date should be available in some reasonable form. Note that the month must appear first.

It is possible to execute **calendar** automatically, every day, via the **cron** command. This command is described in the *cron*(1M) entry in the *System Administration Utilities Reference Manual*. When executed automatically, **calendar** checks all *calendar* files on the system, and advises the appropriate user via **mail**. In order for this to occur, users must keep their *calendar* files in their login directory.

SYNTAX

```
calendar [-]
```

DESCRIPTION

The argument to **calendar** is as follows:

- an optional argument that tells **calendar** to execute for each user possessing a *calendar* file. This is the form of the command that is executed by the **cron** command, for all users.

When entered without the - argument, **calendar** checks only the *calendar* file of the user invoking the command.

EXAMPLES

In this example, it is assumed that the user's *calendar* file has the following contents. Note that the **cat** command has been executed to display the file's contents. Details of this command can be found in the **cat** tutorial in this manual. Remember that the **>** character in bold represents the system prompt, and that the **CR** symbol signifies that the carriage return key is to be pressed in order to enter the command line.

```
>cat calendar CR
Remember - dentist 10:15, 16 June
Collect suit from cleaners 16 June (p.m.)
Dinner with Sue June 25 8.00 - usual place ***
Confirm air tickets - June 17 1987 !
Meeting re: Jamaican conference June 24 1987

>
```

The following example illustrates the output from **calendar**, assuming that today's date is June 16. Note that for the command line shown here to work correctly, the *calendar* file would have to be in the current directory.

```
>calendar CR
Remember - dentist 10:15, 16 June
Collect suit from cleaners 16 June (p.m.)
Confirm air tickets - June 17 1987 !

>
```

Note that to provide an automatic reminder system, an entry must be made in the *crontab* file for **calendar**, for example

```
09 *** /usr/lib/calendar -
```

This entry would automatically execute a search on all *calendar* files every morning at 9:00, so long as the system is up and running. Note that the - argument was used here. For full details of the **crontab** system, see the tutorial in the *Advanced Utilities User Guide*, and the *crontab*(1) entry in the *Utilities Reference Manual*.

DATE: print and set the date

INTRODUCTION

This is a short tutorial-style introduction to the **date** utility. The date can be set, or it can be printed on the standard output.

SYNTAX

```
date [MMDDhhmm[YY]] [+format]
```

DESCRIPTION

The arguments to **date** are as follows:

MM identifies the month by number

DD identifies the date

hh identifes the hour, using the 24 hour clock system

mm identifies the minute

YY optionally identifies the year, using the last two digits. The first date allowed is 1970.

+format generates output of date and time, according to the *format* specified by the user in the command line. Fields are separated by the % character, and are of fixed size, zero-padded if necessary.

The following field descriptors are available:

n	insert a newline character
t	insert a tab character
m	month of the year (01-12)
d	day of the month (01-31)
y	last two digits of the year (00-99)
D	date as *mm/dd/yy*
H	hour (00-23)
M	minute (00-59)
S	second (00-59)
T	time as *HH:MM:SS*
j	day of the year (001-366)
w	day of the week (0-6, Sunday=0)
a	abbreviated weekday (Sun-Sat)
h	abbreviated month (Jan-Dec)
r	time in AM/PM notation

Note that only the super-user can change the date and time, and that it is unwise to change either value when the system is running in multi-user mode.

EXAMPLES

The first example simply asks the system for the date and time, and gives an example of the output produced. Remember that the > in bold face represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>date CR
Fri Aug 14 17:12:51 MET DST 1987

>
```

The **MET** stands for *Middle European Time*, and **DST** stands for *Daylight Saving Time*.

The second example uses the *+format* option to determine the pattern of the output. Remember that the **%** acts as the field separator.

```
>date '+DATE: %m/%d/%y%nTIME: %H:%M:%S' CR
DATE: 14/08/87
TIME: 17:12:51

>
```

ECHO: echo arguments

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **echo** utility which accepts arguments and echoes them on the standard output. It is also useful when data whose values are known are to be piped into another process.

SYNTAX

```
echo [arg ... [arg]]
```

DESCRIPTION

The *arg* argument to **echo** takes the form of a literal string or a special character. Special characters may be escaped using the backslash (\), as follows:

\b backspace

\c prints a line without a newline character

\f formfeed

\n newline

\r carriage return

\t tab

**** backslash

*****x* the 8 bit character whose ASCII code is the 1, 2 or 3 digit octal number, *x*, which must begin with a 0

Special characters recognised by the shell may be escaped using single quotes or backslash. Examples of these in use are given in the next section.

EXAMPLES

In this section, a number of examples are given of the **echo** utility in use. Remember that the **>** character in bold face represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

In the first example, **echo** is used to print the word *hello*.

```
>echo hello CR
hello

>
```

The second example shows an attempt to echo a single asterisk character. In fact, this shows how an unescaped special character can produce an effect that may not be expected. In the second command line, the correct form of the command is given.

```
>echo * CR
chapt1.abc      chapt2.abc      chapt3.abc      appendixA
appendixB

>echo '*' CR
*

>
```

The asterisk was interpreted as a shell metacharacter, and **echo** listed all the files held in the current working directory. The single quotes were used to indicate to **echo** that a literal asterisk was required.

In the next example, a problem with the double quotes method is highlighted. It is required to echo a single quote mark:

```
>echo ' CR
Unmatched '.

>
```

Here, it was assumed that the first quote was part of a pair, used to quote a string, and that the second was not entered by mistake. The next example attempts to deal with this by enclosing the single quotes in quotes of its own. Note what happens. Because there are now three single quotes, **echo** still assumes that one quote is missing. To deal with this, the backslash must be used.

```
>echo ''' CR
Unmatched '.

>echo \' CR
'

>
```

EDITORS

INTRODUCTION

This fifth chapter of the manual provides expansive coverage of the the two basic X/OS editors. Others are described in the *Advanced User Guide*. The editors covered here are as follows:

ED the line editor

VI the screen editor

These tutorials are arranged in alphabetical order.

ED - a line editor

INTRODUCTION

This chapter is a tutorial on the line editor, **ed**. **Ed** is versatile and requires little computer time to perform editing tasks. It can be used on any type of terminal. The examples of command lines and system responses in this chapter will apply to your terminal, whether it is a video display terminal or a paper printing terminal. The **ed** commands can be typed in at your terminal or they can be used in a shell program

Ed is a line editor; during editing sessions it is always pointing at a single line in the file called the current line. When you access an existing file, **ed** makes the last line the current line so you can start appending text easily. Unless you specify the number of a different line or range of lines, **ed** will perform each command you issue on the current line. In addition to letting you change, delete, or add text on one or more lines, **ed** allows you to add text from another file to the buffer.

During an editing session with **ed**, you are altering the contents of a file in a temporary buffer, where you work until you have finished creating or correcting your text. When you edit an existing file, a copy of that file is placed in the buffer and your changes are made to this copy. The changes have no effect on the original file until you instruct **ed**, by using the write command, to move the contents of the buffer into the file.

After you have read through this tutorial and tried the examples and exercises, you will have a good working knowledge of **ed**. The following basics are included:

- entering the line editor **ed**, creating text, writing the text to file, and quitting **ed**

- addressing particular lines of the file and displaying lines of text
- deleting text
- substituting new text for old text
- using special characters as shortcuts in search and substitute patterns
- moving text around in the file, as well as other useful commands and information

GETTING STARTED

The best way to learn **ed** is to log in to the X/OS system and try the examples as you read this tutorial. Do the exercises; do not be afraid to experiment. As you experiment and try out **ed** commands, you will learn a fast and versatile method of text editing.

In this section you will learn the commands used to:

- enter **ed**
- append text
- move up or down in the file to display a line of text
- delete a line of text
- write the buffer to a file
- quit **ed**

HOW TO ENTER ED

To enter the line editor, type **ed** and a file name:

ed *filename* **CR**

Choose a name that reflects the contents of the file. If you are creating a new file, the system responds with a question mark and the file name:

```
>ed new-file CR
?new-file
```

If you going to edit an existing file, **ed** responds with the number of characters in the file:

```
>ed old-file CR
235
```

HOW TO CREATE TEXT

The editor receives two types of input, editing commands and text, from your terminal. To avoid confusing them, **ed** recognizes two modes of editing work: command mode and text input mode. When you work in command mode, any characters you type are interpreted as commands. In input mode, any characters you type are interpreted as text to be added to a file.

Whenever you enter **ed** you are put into command mode. To create text in your file, change to input mode by typing **a** (for append), on a line by itself, and pressing the **CR** key:

```
a CR
```

Now you are in input mode; any characters you type from this point wil be added to your file as text. Be sure to type **a** on a line by itself; if you do not, the editor will not execute your command.

After you have finished entering text, type a period on a line by itself. This takes you out of the text input mode and returns you to the command mode. Now you can give **ed** other commands.

The following example shows how to enter **ed**, create text in a new file called *try-me*, and quit text input mode with a period.

```
>ed try-me CR
? try-me
a CR
This is the first line of text. CR
This is the second line, CR
and this is the third line. CR
. CR
```

Notice that **ed** does not give a response to the period; it just waits for a new command. If **ed** does not respond to a command, you may have forgotten to type a period after entering text and may still be in text input mode. Type a period and press the **CR** key at the beginning of a line to return to command mode. Now you can execute editing commands. For example, if you have added some unwanted characters or lines to your text, you can delete them once you have returned to command mode.

HOW TO DISPLAY TEXT

To display a line of a file, type **p** (for print) on a line by itself. The **p** command prints the current line, that is, the last line on which you worked. Continue with the previous example. You have just typed a period to exit input mode. Now type the **p** command to see the current line.

```
>ed try-me CR
? try-me
a CR
This is the first line of text. CR
This is the second line, CR
and this is the third line. CR
. CR
p CR
and this is the third line.
```

You can print any line of text by specifying its line number (also known as the address of the line). The address of the first line is 1; of the second, 2; and so on. For example, to print the second line in the file *try-me*, type:

```
2p CR
This is the second line,
```

You can also use line addresses to print a span of lines by specifying the addresses of the first and last lines of the section you want to see, separated by a comma. For example, to print the first three lines of a file, type:

```
1,3p CR
```

You can even print the whole file this way. For example, you can display a twenty-line file by typing **1,20p**. If you do not know the address of the last line in your file, you can substitute a **$** sign, **ed** symbol for the address of the last line. (These conventions are discussed in detail in the section called *Line Addressing*, below.)

```
1,$p CR
This is the first line of text.
This is a second line,
and this is the third line.
```

If you forget to quit text input mode with a period, you will add text that you do not want. Try to make this mistake. Add another line of text to your *try-me* file and then try the **p** command without quitting text input mode. Then quit text input mode and print the entire file.

```
p CR
and this is the third line.
a CR
This is the fourth line. CR
p CR
. CR
1,$p CR
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
p
```

What did you get? The next section will explain how to delete the unwanted line.

HOW TO DELETE A LINE OF TEXT

To delete text, you must be in the command mode of **ed.**

Try this command on the last example to remove the unwanted line containing **p**. Display the current line (**p** command), delete it (**d** command), and display the remaining lines in the file (**p** command). Your screen should look like this:

```
p CR
p
d CR
1,$p CR
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
```

Ed does not send you any messages to confirm that you have deleted text. The only way you can verify that the **d** command has succeeded is by printing the contents of your file with the **p** command. To receive verification of your deletion, you can put the **d** and **p** together on one command line. If you repeat the previous example with this command, your screen should look like this:

```
p CR
p
dp CR
This is the fourth line.
```

HOW TO MOVE UP OR DOWN IN A FILE

To display the line below the current line, press the **CR** key while in command mode. If there is no line below the current line, **ed** responds with a **?** and continues to treat the last line of the file as the current line. To display the line above the current line, press the minus key (-).

The following screen provides examples of how both of these commands are used:

```
p CR
This is the fourth line.
- CR
and this is the third line.
- CR
This is a second line,
- CR
This is the first line of text.
CR
This is a second line,
CR
and this is the third line.
```

Notice that by typing - **CR** or **CR**, you can display a line of text without typing the **p** command. These commands are also line addresses. Whenever you type a line address and do not follow it with a command, **ed** assumes that you want to see the line you have specified. Experiment with these commands: create some text, delete a line, and display your file.

HOW TO SAVE THE BUFFER CONTENTS IN A FILE

As we discussed earlier, during an editing session, the system holds your text in a temporary storage area called a buffer. When you have finished editing, you can save your work by writing it from the temporary buffer to a permanent file in the computer's memory. By writing to a file, you are simply putting a copy of the contents of the buffer into the file. The text in the buffer is not disturbed, and you can make further changes to it.

Note that it is a good idea to write the buffer text into your file frequently. If an interrupt occurs (such as an accidental loss of power to your terminal), you may lose the material in the buffer, but you will not lose the copy written to your file.

To write your text to a file, enter the **w** command. You do not need to specify a file name; simply type **w** and press the **CR** key. If you have just created new text, **ed** creates a file for it with the name you specified when you entered the editor. If you have edited an existing file, the **w** command writes the contents of the buffer to that file by default.

If you prefer, you can specify a new name for your file as an argument on the **w** command line. Be careful not to use the name of a file that already exists unless you want to replace its contents with the contents of the current buffer. **ed** will not warn you about an existing file; it will simply overwrite that file with your buffer contents.

For example, if you decide you would prefer the *try-me* file to be called *stuff*, you can rename it:

```
>ed try-me CR
? try-me
a CR
This is the first line of text. CR
```

```
This is the second line, CR
and this is the third line. CR
.
w stuff CR
85
```

Notice the last line of the screen. This is the number of characters in your text. When the editor reports the number of characters in this way, the write command has succeeded.

HOW TO QUIT THE EDITOR

When you have completed editing your text, write it from the buffer into a file with the **w** command. Then leave the editor and return to the shell by typing **q** (for quit).

```
w CR
85
q CR

>
```

The system responds with a shell prompt. At this point the editing buffer vanishes. If you have not executed the write command, your text in the buffer has also vanished. If you did not make any changes to the text during your editing session, no harm is done. However, if you did make changes, you could lose your work in this way. Therefore, if you type **q** after changing the file without writing it, **ed** warns you with a **?**. You then have a chance to write and quit.

```
q CR
?
```

```
w CR
85
q CR

>
```

If, instead of writing, you insist on typing **q** a second time, **ed** assumes you do not want to write the buffer's contents to your file and returns you to the shell. Your file is left unchanged and the contents of the buffer are wiped out.

You now know the basic commands needed to create and edit a file using **ed**.

The editor commands are summarized in the table below.

Ed Editor Commands

ed *file*	Enter **ed** to edit *file*
a	Append text after the current line
.	Quit text input mode and return to **ed** command mode
p	Print text on your terminal
d	Delete text
CR	Display the next line in the buffer. Literally, carriage return
+	Display the next line in the buffer
-	Display the previous line in the buffer
w	Write the contents of the buffer to the file
q	Quit **ed** and return to the shell

EXERCISE 1

Answers for all the exercises in this chapter are found at the end of the chapter. However, they are not necessarily the only possible correct answers. Any method that enables you to perform a task specified in an exercise is correct, even if it does not match the answer given.

1. Enter **ed** with a file named *junk*. Create line of text containing *Hello World*, write it to the file and quit **ed**.

   Now use **ed** to create a file called *stuff*. Create a line of text containing two words, *Goodbye world*, write this text to the file, and quit **ed**.

2. Enter **ed** again with the file named *junk*. What was the editor's response? Was the character count for it the same as the character count reported by the **w** command in exercise 1?

   Display the contents of the file. Is that your file *junk*?

   How can you return to the shell? Try **q** without writing the file. Why do you think the editor allowed you to quit without writing to the buffer?

3. Enter **ed** with the file *junk*. Add a line:

   ```
   Wendy's horse came through the window.
   ```

   Since you did not specify a line address, where do you think the line was added to the buffer? Display the contents of the buffer. Try quitting the buffer without writing to the file. Try writing the buffer to a different file called *stuff*. Notice that **ed** does not warn you that a file called *stuff* already

exists. You have erased the contents of *stuff* and replaced them with new text.

GENERAL FORMAT OF ED COMMANDS

ed commands have a simple and regular format:

[*address1*[,*address2*]]*command*[*argument*]

The brackets around *address1*, *address2*, and *argument* show that these are optional. The brackets are not part of the command line.

address1,*address2* — The addresses give the position of lines in the buffer. *Address1* through *address2* gives you a range of lines that will be affected by the *command*. If *address2* is omitted, the command will affect only the line specified by *address1*.

command — The *command* is one character and tells the editor what task to perform.

argument — The *arguments* to a *command* are those parts of the text that will be modified, or a file name, or another line address.

This format will become clearer to you when you begin to experiment with the **ed** commands.

LINE ADDRESSING

A line address is a character or group of characters that identifies a line of text. Before **ed** can execute commands that add, delete, move, or change text, it must know the line address of the affected text. Type the line address before the command:

[*address1*],[*address2*]*command*

Both *address1* and *address2* are optional. Specify *address1* alone to request action on a single line of text; both *address1* and *address2* to request a span of lines. If you do not specify any *address*, **ed** assumes that the line address is the current line.

The most common ways to specify a line address in **ed** are:

- by entering line numbers (assuming that the lines of the files are consecutively numbered from 1 to *n*, beginning with the first line of the file)
- by entering special symbols for the current line, last line, or a span of lines
- by adding or subtracting lines from the current line
- by searching for a character string or word on the desired line

You can access one line or a span of lines, or make a global search for all lines containing a specified character string. (A character string is a set of successive characters, such as a word.)

Numerical Addresses

ed gives a numerical address to each line in the buffer. The first line of the buffer is 1, the second line is 2, and so on, for each line in the buffer. Any line can be accessed by **ed** with its line address number. To see how line numbers address a line, enter **ed** with the file *try-me* and type a number.

```
>ed try-me CR
110
1 CR
This is the first line of text.
3 CR
and this is the third line.
```

Remember that **p** is the default command for a line address specified without a command. Because you gave a line address, **ed** assumes you want that line displayed on your terminal.

Numerical line addresses frequently change in the course of an editing session. Later in this chapter you will create lines, delete lines, or move a line to a different position. This will change the line address numbers of some lines. The number of a specific line is always the current position of that line in the editing buffer. For example, if you add five lines of text between line 5 and 6, line 6 becomes line 11. If you delete line 5, line 6 becomes line 5.

SYMBOLIC ADDRESSES

Symbolic Address of the Current Line

The current line is the line most recently acted on by any **ed** command. If you have just entered **ed** with an existing file, the current line is the last line of the buffer. The symbol for the address of the current line is a period. Therefore you can display the current line simply by typing a period (.) and pressing the **CR** key.

Try this command in the file *try-me*:

```
>ed try-me CR
110
. CR
This is the fourth line.
```

The . is the address. Because a command is not specified after the period, **ed** executes the default command **p** and displays the line found at this address.

To get the line number of the current line, type the following command:

```
.= CR
```

Ed responds with the line number. For example, in the *try-me* file, the current line is 4.

```
. CR
This is the fourth line.
.= CR
4
```

Symbolic Address of the Last Line

The symbolic address for the last line of a file is the **$** sign. To verify that the **$** sign accesses the last line, access the *try-me* file with **ed** and specify this address on a line by itself. (Keep in mind that when you first access a file, your current line is always the last line of the file.)

```
>ed try-me CR
110
. CR
This is the fourth line.
$ CR
This is the fourth line.
```

Remember that the **$** address within **ed** is not the same as the **$** prompt from the shell.

Symbolic Address of the Set of All Lines

When used as an address, a comma (,) refers to all the lines of a file, from the first through the last line. It is an abbreviated form of the string mentioned earlier that represents all lines in a file, **1,$**. Try this shortcut to print the contents of *try-me*:

```
,p CR
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
```

Symbolic Address of the Current Line through the Last Line

The semi-colon (*try-me*) represents a set of lines beginning with the current line and ending with the last line of a file. It is equivalent to the symbolic address **.,$**. Try it with the file *try-me*:

```
2p CR
This is the second line,
;p CR
This is the second line,
and this is the third line.
This is the fourth line.
```

RELATIVE ADDRESSES

You may often want to address lines with respect to the current line. You can do this by adding or subtracting a number of lines from the current line with a plus (+) or a minus (-) sign. Addresses derived in this way are called relative addresses. To experiment with relative line addresses, add several more lines to your file *try-me*, as shown in the following screen. Also, write the buffer contents to the file so your additions will be saved:

```
>ed try-me CR
110
. CR
This is the fourth line.
a CR
five CR
six CR
```

```
seven CR
eight CR
nine CR
ten CR
. CR
w CR
140
```

Now try adding and subtracting line numbers from the current line.

```
4 CR
This is the fourth line.
+3 CR
seven
-5 CR
This is a second line,
```

What happens if you ask for a line address that is greater than the last line, or if you try to subtract a number greater than the current line number?

```
5 CR
five
-6 CR
?
.= CR
5
+7 CR
?
```

Notice that the current line remains at line 5 of the buffer. The current line changes only if you give **ed** a correct address. The **?** response means there is an error. The section entitled *Other Useful Commands and Information*, at the end of this chapter, explains how to

get a help message that describes the error.

Character String Addresses

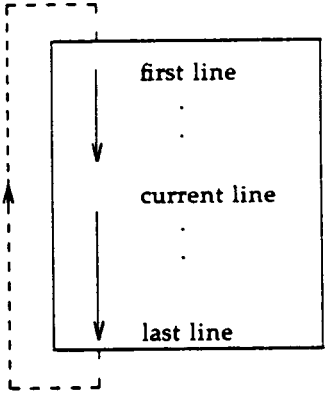
You can search forward or backward in the file for a line containing a particular character string. To do so, specify a string, preceded by a delimiter.

Delimiters mark the boundaries of character strings; they tell **ed** where a string starts and ends. The most common delimiter is / (slash), used in the following format:

/pattern

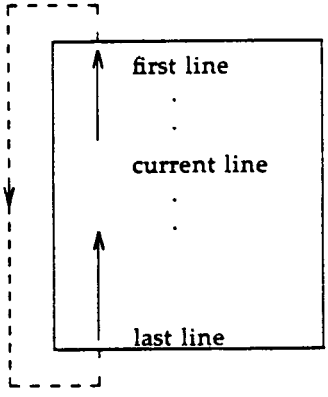
When you specify a pattern preceded by a / (slash), **ed** begins at the current line and searches forward (down through subsequent lines in the buffer) for the next line containing the *pattern*. When the search reaches the last line of the buffer, **ed** wraps around to the beginning of the file and continues its search from line 1.

The following rectangle represents the editing buffer. The path of the arrows shows the search initiated by a /:



Another useful delimiter is **?**. If you specify a pattern preceded by a **?**, (*?pattern*), **ed** begins at the current line and searches backward (up through previous lines in the buffer) for the next line containing the *pattern*. If the search reaches the first line of the file, it will wrap around and continue searching upward from the last line of the file.

The following rectangle represents the editing buffer. The path of the arrows shows the search initiated by a **?**



Experiment with these two methods of requesting address searches on the file *try-me*. What happens if **ed** does not find the specified character string?

```
>ed try-me CR
140
. CR
ten
?first CR
This is the first line of text.
/fourth CR
This is the fourth line.
/junk CR
?
```

In this example, **ed** found the specified strings *first* and *fourth*. Then, because no command was given with the address, it executed the **p** command by default, displaying the lines it had found. When **ed** cannot find a specified string (such as *junk*), it responds with a **?**.

You can also use the **/** (slash) to search for multiple occurrences of a pattern without typing it more than once. First, specify the pattern by typing */pattern*, as usual. After **ed** has printed the first occurrence, it waits for another command. Type **/** and press the **CR** key; **ed** will continue to search forward through the file for the last *pattern* specified. Try this command by searching for the word *line* in the file *try-me*:

```
. CR
This is the first line of text.
/line CR
This is the second line,
/ CR
and this is the third line.
/ CR
This is the fourth line.
/ CR
This is the first line of text.
```

Notice that after **ed** has found all occurrences of the *pattern* between the line where you requested a search and the end of the file, it wraps around to the beginning of the file and continues searching.

SPECIFYING A RANGE OF LINES

There are two ways to request a group of lines. You can specify a range of lines, such as *address1* through *address2*, or you can specify a global search for all lines containing a specified pattern.

The simplest way to specify a range of lines is to use the line numbers of the first and last lines of the range, separated by a comma. Place this address before the command. For example, if you want to display lines 2 through 7 of the editing buffer, give *address1* as 2 and *address2* as 7 in the following format:

```
2,7p CR
```

Try this on the file *try-me*:

```
2,7p CR
This is the second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

Did you try typing **2,7** without the **p**? What happened? If you do not add the **p** command, **ed** prints only *address2*, the last line of the range of addresses.

Relative line addresses can also be used to request a range of lines. Be sure that *address1* precedes *address2* in the buffer. Relative addresses are calculated from the current line, as the following example shows:

```
4 CR
This is the fourth line
-2,+3p CR
This is the second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

SPECIFYING A GLOBAL SEARCH

There are two commands that do not follow the general format of **ed** commands: **g** and **v**. These are global search commands that specify addresses with a character string (*pattern*). The **g** command searches for all lines containing the string *pattern* and performs the *command* on those lines. The **v** command searches for all lines that do not contain the *pattern* and performs the *command* on those lines.

The general format for these commands is:

g/*pattern*/*command*

v/*pattern*/*command*

Try these commands by using them to search for the word *line* in *try-me*:

```
g/line/p CR
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line
```

```
v/line/p CR
five
six
seven
eight
nine
ten
```

Notice the function of the **v** command: it finds all the lines that do not contain the word specified in the command line (*line*).

Once again, the default command for the lines addressed by **g** or **v** is **p**; you do not need to include a **p** as the last delimiter on your command line.

```
g/line CR
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line
```

However, if you are giving line addresses to be used by other **ed** commands, you need to include beginning and ending delimiters. You can use any of the methods discussed in this section to specify line addresses for **ed** commands. The symbols and commands available for addressing lines are summarized in the table below.

Ed Line Addressing Commands

n...	The number of a line in the buffer
.	The current line (the line most recently acted upon by an **ed** command
.=	The command used to request the line number of the current line
$	The last line of the file
,	The set of lines from line 1 through the last line
;	The set of lines from the current line through the last line
+*n*	The line that is located *n* lines after the current line
-*n*	The line that is located *n* lines before the current line
/*abc*	The command used to search forward in the buffer for the first line that contains the pattern *abc*
?*abc*	The command used to search backward in the buffer for the first line that contains the pattern *abc*
g/*abc*	The set of all lines that contain the pattern *abc*
v/*abc*	The set of all lines that do not contain the pattern *abc*

EXERCISE 2

1. Create a file called *towns* with the following lines:

```
My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
I lost my heart in
San Francisco
I lost $$ in
```

```
Las Vegas
```

2. Display line 3.

3. If you specify a range of lines with the relative address **-2,+3p**, what lines are displayed ?

4. What is the current line number? Display the current line.

5. What does the last line say?

6. What line is displayed by the following request for a search?

```
?town CR
```

After **ed** responds, type this command alone on a line:

```
? CR
```

What happened?

7. Search for all lines that contain the pattern *in*. Then search for all lines that do NOT contain the pattern *in*.

DISPLAYING TEXT IN A FILE

Ed provides two commands for displaying lines of text in the editing buffer: **p** and **n**.

Displaying Text Alone

You have already used the **p** command in several examples. You are probably now familiar with its general format:

[*address1*,*address2*]**p**

The **p** command does not take arguments. However, it can be combined with a substitution command line. This will be discussed later in this chapter.

Experiment with the line addresses shown in the table below. Use a file in your home directory. Try the **p** command with each address and see if **ed** responds as described in the figure.

Sample Adresses

1,$p	**Ed** should display the entire file on your terminal
-5p	**Ed** should move backwards five lines from the current line and display the line found there
+2p	**Ed** should move forward two lines from the current line and display the line found there
1,/*x*/p	**Ed** displays the set of lines from line 1 through the first line after the current line that contains the character *x*. It is important to enclose the letter *x* between slashes so that ed can distinguish between the search pattern address (*x*) and the ed command (p)

Displaying Text With Line Addresses

The **n** command displays text and precedes each line with its numerical line address. It is helpful when you are deleting, creating, or changing lines. The general command line format for **n** is the same as that for **p**.

[*address1*,*address2*]**n**

Like **p**, **n** does not take arguments, but it can be combined with the substitute command.

Try running **n** on the *try-me* file:

```
>ed try-me CR
140
1,$n CR
1       This is the first line of text.
2       This is the second line,
3       and this is the third line.
4       This is the fourth line.
5       five
6       six
7       seven
8       eight
9       nine
10      ten
```

The **ed** commands for displaying text are summarized in the table below.

Ed Text Display Commands

p	Displays specified lines of the text in the editing buffer on your terminal
n	Displays specified lines of the text in the editing buffer with their numerical line addresses on your terminal

CREATING TEXT

Ed has three basic commands for creating new lines of text:

a append text

i insert text

c change text

Appending Text

The append command, **a**, allows you to add text AFTER the current line or a specified address in the file. You have already used this command in the *Getting Started* section of this chapter. The general format for the append command line is:

[*address1*]**a**

Specifying an address is optional. The default value of *address1* is the current line.

In previous exercises, you used this command with the default address. Now try using different line numbers for *address1*. In the following example, a new file called *new-file* is created. In the first append command line, the default address is the current line. In the second append command line, line 1 is specified as *address1*. The lines are displayed with **n** so that you can see their numerical line addresses. Remember, the append mode is ended by typing a period (.) on a line by itself.

```
>ed new-file CR
?new-file
a CR
Create some lines
of text in
this file.
. CR
1,$n CR
1       Create some lines
2       of text in
3       this file.
1a CR
This will be line 2 CR
This will be line 3 CR
. CR
1,$n CR
1       Create some lines
2       This will be line 2
3       This will be line 3
4       of text in
5       this file.
```

Notice that after you append the two new lines, the line that was originally line 2 (*of text in*) becomes line 4.

You can take shortcuts to places in the file where you want to append text by combining the append command with symbolic addresses. The following three command lines allow you to move through and add to the text quickly in

this way.

.a appends text after the current line

$a appends text after the last line of the file

0a appends text before the first line of the file (at a symbolic address called line 0)

To try using these addresses, create a one-line file called *lines* and type the examples shown in the following screens. (The examples appear in separate screens for easy reference only; it is not necessary to access the *lines* file three times to try each append symbol. You can access *lines* once and try all three consecutively.)

Example 1:

```
>ed lines CR
?lines
a CR
This is the current line. CR
. CR
p CR
This is the current line.
.a CR
This line is after the current line. CR
. CR
-1,.p CR
This is the current line.
This line is after the current line.
```

Example 2:

```
a CR
This is the last line now. CR
. CR
$ CR
```

```
This is the last line now.
```

Example 3:

```
0a CR
This is the first line now. CR
This is the second line now. CR
The line numbers change CR
as lines are added. CR
. CR
1,4n CR
1       This is the first line now.
2       This is the second line now.
3       The line numbers change
4       as lines are added.
```

Because the append command creates text after a specified address, the last example refers to the line before line 1 as the line after line 0. To avoid such circuitous references, use another command provided by the editor: the insert command, **i**.

Inserting Text

The insert command (**i**), allows you to add text BEFORE a specified line in the editing buffer. The general command line format for **i** is the same as that for **a**.

```
[address1]i
```

As with the append command, you can insert one or more lines of text. To quit input mode, you must type a period (.) alone on a line.

Create a file called *insert* in which you can try the insert command (**i**):

```
>ed insert CR
?insert
a CR
Line 1 CR
Line 2 CR
Line 3 CR
Line 4 CR
. CR
w CR
69
```

Now insert one line of text above line 2 and another above line 1. Use the **n** command to display all the lines in the buffer:

```
2i CR
This is the new line 2. CR
. CR
1,$n CR
1       Line 1
2       This is the new line 2.
3       Line 2
4       Line 3
5       Line 4
1i CR
This is the beginning. CR
. CR
1,$n CR
1       In the beginning
2       Line 1
3       Now this is line 2
4       Line 2
5       Line 3
6       Line 4
```

Experiment with the insert command by combining it with symbolic line addresses, as follows:

- **.i**

- **$i**

Changing Text

The change text command (**c**) erases all specified lines and allows you to create one or more lines of text in their place. Because **c** can erase a range of lines, the general format for the command line includes two addresses.

[*address1*,*address2*]**c**

The change command puts you in text input mode. To leave input mode, type a period alone on a line.

Address1 is the first and *address2* is the last of the range of lines to be replaced by new text. To erase one line of text, specify only *address1*. If no address is specified, **ed** assumes the current line is the line to be changed.

Now create a file called *change* in which you can try this command. After entering the text shown in the screen, change lines one through four by typing **1,4c**:

```
1,5n CR
1       line 1
2       line 2
3       line 3
4       line 4
```

```
5       line 5
1,4c CR
Change line 1 CR
and lines 2 through 4 CR
. CR
1,$n CR
1       change line 1
2       and lines 2 through 4
3       line 5
```

Now experiment with **c** and try to change the current line:

```
. CR
line 5
c CR
This is the new line 5.
. CR
. CR
This is the new line 5.
```

If you are not sure whether you have left text input mode, it is a good idea to type another period. If the current line is displayed, you know you are in the command mode of **ed**.

The **ed** commands for creating text are summarized in the table below.

Ed Text Creation Commands

a	Append text after the specified line in the buffer
i	Insert text before the specified line in the buffer
c	Change the text on the specified line(s) to new text
.	Quit text input mode and return to **ed** command mode

EXERCISE 3

1. Create a new file called *ex3*. Instead of using the append command to create new text in the empty buffer, try the insert command. What happens?

2. Enter **ed** with the file *towns*. What is the current line?

   Insert above the third line:

   ```
   Illinois CR
   ```

   Insert above the current line:

   ```
   or CR
   Naperville CR
   ```

   Insert before the last line:

   ```
   hotels in CR
   ```

   Display the text in the buffer preceded by line numbers.

3. In the file *towns*, display lines 1 through 5 and replace lines 2 through 5 with:

   ```
   London CR
   ```

   Display lines 1 through 3.

4. After you have completed number 3, what is the current line?

   Find the line of text containing:

   ```
   Toledo
   ```

   Replace

   ```
   Toledo
   ```

   with

   ```
   Peoria
   ```

   Display the current line.

5. With one command line search for and replace:

   ```
   New York
   ```

   with:

   ```
   Iron City
   ```

DELETING TEXT

This section discusses two types of commands for deleting text in **ed**. One type is to be used when you are working in command mode: **d** deletes a line and **u** undoes the last command. The other type of command is to be used in text input mode: **#** (the hash sign) deletes a character and **@** (the at sign) kills a line. The delete keys that are used in input mode are the same keys you use to delete text that you enter after a shell prompt.

Deleting Lines

You have already deleted lines of text with the delete command (**d**) in the *Getting Started* section of this chapter.

The general format for the **d** command line is:

[*address1*,*address2*]**d**

You can delete a range of lines (*address1* through *address2*) or you can delete one line only (*address1*). If no address is specified, **ed** deletes the current line.

The next example displays lines one through five and then deletes lines two through four:

```
1,5n CR
1       1 horse
2       2 chickens
3       3 ham tacos
4       4 cans of mustard
5       5 bails of hay
2,4d CR
1,$n CR
1       1 horse
```

```
2       5 bails of hay
```

How can you delete only the last line of a file? Using a symbolic line address makes this easy:

```
$d CR
```

How can you delete the current line? One of the most common errors in **ed** is forgetting to type a period to leave text input mode. When this happens, unwanted text may be added to the buffer. In the next example, a line containing a print command (**1,$p**) is accidentally added to the text before the user leaves input mode. Because this line was the last one added to the text, it becomes the current line. The symbolic address . is used to delete it.

```
a CR
Last line of text CR
1,$p CR
. CR
p CR
1,$p
.d CR
p CR
Last line of text.
```

Before experimenting with the delete command, you may first want to learn about the undo command, **u**.

Undoing the Previous Command

The command **u** (short for undo) nullifies the last command and restores any text changed or deleted by that command. It takes no addresses or arguments.

One purpose for which the **u** command is useful is to restore text you have mistakenly deleted. If you delete all the lines in a file and then type **p**, **ed** will respond with a **?** since there are no more lines in the file. Use the **u** command to restore them.

```
1,$p CR
This is the first line.
This is the middle line.
This is the last line.
1,$d CR
p CR
?
u CR
p CR
This is the last line.
```

Now experiment with **u**: use it to undo the append command.

```
. CR
This is the only line of text
a CR
Add this line CR
. CR
1,$p CR
This is the only line of text
Add this line
u CR
1,$p CR
This is the only line of text
```

The **u** command cannot be used to undo the write command (**w**) or the quit command (**q**). However, **u** can undo an undo command (**u**).

Deleting Text In Input Mode

While in text input mode, you can correct the current line of input with the same keys you use to correct a shell command line. By default, there are two keys available to correct text. The **@** sign key kills the current line. The **#** sign key backs up over one character on the current line so you can retype it, thus effectively erasing the original character.

Note that you can reassign the line kill and character erase functions to other keys if you prefer. If you have reassigned these functions, you must use the keys you chose while working in **ed**; the default keys (**@** and **#**) will no longer work.

Escaping the Delete Function

You may want to include an **@** sign or a **#** sign as a character of text. To avoid having these characters interpreted as delete commands, you must precede them with a \ (backslash), as shown in the following example.

```
a CR
leave San Francisco \@ 20:15 on flight \#347 CR
. CR
p CR
leave San Francisco @ 20:15 on flight #347
```

The **ed** and shell commands for deleting text in **ed** are summarized in the table below.

Ed Text Deletion Commands

COMMAND MODE:	
d	Delete one or more lines of text
u	Undo the previous command
@	Delete the current command line
TEXT INPUT MODE:	
@	Delete the current line
# or <BACKSPACE>	Delete the last character typed in

SUBSTITUTING TEXT

You can modify your text with a substitute command. This command replaces the first occurrence of a string of characters with new text. The general command line format is

[*address1*,*address2*]**s**/*old_text*/*new_text*/[*command*]

Each component of the command line is described below.

address1,*address2* — The range of lines being addressed by **s**. The address can be one line, (*address1*), a range of lines (*address1* through *address2*), or a global search address. If no address is given, **ed** makes the substitution on the current line.

s — The substitute command

/old_text — The argument specifying the text to be replaced is usually delimited by slashes, but can be delimited by other characters such as a ? or a

period. It consists of the words or characters to be replaced. The command will replace the first occurrence of these characters that it finds in the text.

fInew_text — The argument specifying the text to replace *old_text*. It is delimited by slashes or the same delimiters used to specify the *old_text*. It consists of the words or characters that are to replace the *old_text*.

command — Any one of the following four commands:

g Change all occurrences of *old_text* on the specified lines.

l Display the last line of substituted text, including nonprinting characters. (See the last section of this chapter, *Other Useful Commands and Information*.)

n Display the last line of the substituted text preceded by its numerical line address.

p Display the last line of substituted text.

Substituting on the Current Line

The simplest example of the substitute command is making a change to the current line. You do not need to give a line address for the current line.

s/*old_text/new_text/*

The next example contains a typing error. While the line that contains it is still the current line, you make a substitution to correct it. The old text is the *ai* of *airor* and the new text is *er*.

```
a CR
In the beginning, I made an airor.
. CR
.p CR
In the beginning, I made an airor.
s/ai/er CR
```

Notice that **ed** gives no response to the substitute command. To verify that the command has succeeded in this case, you either have to display the line with **p** or **n**, or include **p** or **n** as part of the substitute command line. In the following example, **n** is used to verify that the word *file* has been substituted for the word *toad*.

```
.p CR
This is a test toad
s/toad/file/n CR
1       This is a test file
```

However, **ed** allows you one shortcut: it prints the results of the command automatically, if you omit the last delimiter after the *new_text* argument:

```
.p CR
This is a test file
s/file/frog CR
This is a test frog
```

Substituting on One Line

To substitute text on a line that is not the current line, include an address in the command line, as follows:

[*address1*]**s**/*old_text*/*new_text*/

For example, in the following screen the command line includes an address for the line to be changed (line 1) because the current line is line 3:

```
1,3p CR
This is a pest toad
testing testing
come in toad
. CR
come in toad
1s/pest/test CR
This is a test toad
```

As you can see, **ed** printed the new line automatically after the change was made, because the last delimiter was omitted.

Substituting on a Range of Lines

You can make a substitution on a range of lines by specifying the first address (*address1*) through the last address (*address2*).

[*address1*,*address2*]**s**/*old_text*/*new_text*/

If **ed** does not find the pattern to be replaced on a line, no changes are made to that line.

In the following example, all the lines in the file are addressed for the substitute command. However, only the lines that contain the string *es* (the *old_text* argument) are changed.

```
1,$p CR
This is a test toad
testing testing
come in toad
testing 1, 2, 3
1,$s/es/ES/n CR
4       tESting 1, 2, 3
```

When you specify a range of lines and include **p** or **n** at the end of the substitute line, only the last line changed is printed.

To display all the lines in which text was changed, use the **n** or **p** command with the address **1,$**.

```
1,$n CR
1       This is a tESt toad
2       tESting testing
3       come in toad
4       tESting 1, 2, 3
```

Notice that only the first occurrence of *es* (on line 2) has been changed. To change every occurrence of a pattern, use the **g** command, described in the next section.

Global Substitution

One of the most versatile tools in **ed** is global substitution. By placing the **g** command after the last delimiter on the substitute command line, you can change every occurrence of a pattern on the specified lines. Try changing every occurrence of the string *es* in the last example. If you are following along, doing the examples as you read this, remember you can use **u** to undo the last substitute command.

```
u CR
1,$p CR
This is a test toad
testing, testing
come in toad
testing 1, 2, 3
1,$s/es/ES/g CR
1,$p CR
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Another method is to use a global search pattern as an address instead of the range of lines specified by **1,$**.

```
1,$p CR
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/test/s/es/ES/g CR
1,$p CR
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

If the global search pattern is unique and matches the argument *old_text* (text to be replaced), you can use an **ed** shortcut: specify the pattern once as the global search address and do not repeat it as an *old_text* argument. **ed** will remember the pattern from the search address and use it again as the pattern to be replaced.

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

Note that whenever you use this shortcut, be sure to include two slashes (//) after the **s**.

```
1,$p CR
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/es/s//ES/g CR
1,$p CR
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Experiment with other search pattern addresses:

/pattern **CR** *?pattern* **CR** **v/***pattern* **CR**

See what they do when combined with the substitute command. In the following example, the **v/***pattern* search format is used to locate lines that do not contain the pattern *testing*. Then the substitute command (**s**) is used to replace the existing pattern (*in*) with a new pattern (*out*) on those lines.

```
v/testing/s/in/out CR
This is a test toad
come out toad
```

Notice that the line *This is a test toad* was also printed, even though no substitution was made on it. When the last delimiter is omitted, all lines found with the search address are printed, regardless of whether or not substitutions have been made on them.

Now search for lines that do contain the pattern *testing* with the **g** command.

```
g/testing/s//jumping CR
jumping testing
jumping 1, 2, 3
```

Notice that this command makes substitutions only for the first occurrence of the *pattern* (*testing*) in each line. Once again, the lines are displayed on your terminal because the last delimiter has been omitted.

EXERCISE 4

1. In your file *towns* change *town* to *city* on all lines but the line with *little town* on it.

   The file should read:

```
My kind of city is
London
Like being no where at all in
Peoria
I lost those little town blues in
Iron City
```

```
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas
```

2. Try using ? as a delimiter. Change the current line

```
Las Vegas
```

to

```
Toledo
```

Because you are changing the whole line, you can also do this by using the change command, **c**.

3. Try searching backward in the file for the word

```
lost
```

and substitute

```
found
```

using the ? as the delimiter.
Did it work?

4. Search forward in the file for

```
no
```

and substitute

NO

for it. What happens if you try to use ? as a delimiter?

Experiment with the various command combinations available for addressing a range of lines and doing global searches.

What happens if you try to substitute something for the **$$** ? Try to substitute **Big $** for **$** on line 9 of your file. Type:

```
9s/$/Big $ CR
```

What happened?

SPECIAL CHARACTERS

If you try to substitute the **$** sign in the line

```
I lost my $ in Las Vegas
```

you will find that instead of replacing the **$**, the new text is placed at the end of the line. The **$** is a special character in **ed** that is symbolic for the end of the line.

Ed has several special characters that give you a shorthand for search patterns and substitution patterns. The characters act as wild cards. If you have tried to type in any of these characters, the result was probably

different than what you had expected.

The special characters are:

.	Match any one character.
*	Match zero or more occurrences of the preceding character.
.*	Match zero or more occurrences of any character following the period.
^	Match the beginning of the line.
$	Match the end of the line.
\	Take away the special meaning of the special character that follows.
&	Repeat the old text to be replaced in the new text of the replacement pattern.
[...]	Match the first occurrence of a character in the brackets.
[^...]	Match the first occurrence of a character that is NOT in the brackets.

In the following example, **ed** searches for any three-character sequence ending in the pattern at.

```
1,$p CR
rat
cat
turtle
cow
goat
g/.at CR
rat
cat
```

```
goat
```

Notice that the word goat is included because the string oat matches the string .Iat.

The asterisk (*) represents zero or more occurrences of a specified character in a search or substitute pattern. This can be useful in deleting repeated occurrences of a character that have been inserted by mistake. For example, suppose you hold down the <R> key too long while typing the word broke. You can use the * to delete every unnecessary R with one substitution command.

```
p CR
brrroke
s/br*/br CR
broke
```

Notice that the substitution pattern includes the *b* before the first *r*. If the b were not included in the search pattern, the * would interpret it, during the search, as a zero occurrence of *r*, make the substitution on it, and quit. (Remember, only the first occurrence of a pattern is changed in a substitution, unless you request a global search with **g**.) The following screen shows how the substitution would be made if you did not specify both the b and the r before the *.

```
p CR
brrroke
s/r*/r CR
rbrrroke
```

If you combine the period and the *, the combination will match all characters. With this combination you can replace all characters in the last part of a line:

```
p CR
Toads are slimy, cold creatures
s/are.*/are wonderful and warm CR
Toads are wonderful and warm
```

The .* can also replace all characters between two patterns.

```
p CR
Toads are slimy, cold creatures
s/are.*cre/are wonderful and warm cre CR
Toads are wonderful and warm creatures
```

If you want to insert a word at the beginning of a line, use the ^ (circumflex) for the old text to be substituted. This is very helpful when you want to insert the same pattern in the front of several lines. The next example places the word *all* at the beginning of each line:

```
1,$p CR
creatures great and small
things wise and wonderful
things bright and beautiful
1,$s/^/all / CR
1,$p CR
all creatures great and small
all things wise and wonderful
all things bright and beautiful
```

The **$** sign is useful for adding characters at the end of a line or a range of lines:

```
1,$p CR
```

```
I love
I need
I use
The IRS wants my
1,$s/$/ money. CR
1,$p CR
I love money.
I need money.
I use money.
The IRS wants my money.
```

In these examples, you must remember to put a space after the word *all* or before the word *money* because **ed** adds the specified characters to the very beginning or the very end of the sentence. If you forget to leave a space before the word *money*, your file will look like this:

```
1,$s/$/money/ CR
1,$p CR
I lovemoney
I needmoney
I usemoney
The IRS wants mymoney
```

The **$** sign also provides a handy way to add punctuation to the end of a line:

```
1,$p CR
I love money
I need money
I use money
The IRS wants my money
1,$s/$/./ CR
1,$p/ CR
I love money.
I need money.
I use money.
```

```
The IRS wants my money.
```

Because . is not matching a character (old text), but replacing a character (new text), it does not have a special meaning. To change a period in the middle of a line, you must take away the special meaning of the period in the old text. To do this, simply precede the period with a backslash (\). This is how you take away the special meaning of some special characters that you want to treat as normal text characters in search or substitute arguments. For example, the following screen shows how to take away the special meaning of the period:

```
p CR
Way to go.  Wow!
s/\./! CR
Way to go!  Wow!
```

The same method can be used with the backslash character itself. If you want to treat a \ as a normal text character, be sure to precede it with another \. For example, if you want to replace the \ symbol with the word backslash, use the substitute command line shown in the following screen:

```
1,2p CR
This chapter explains
how to use the \.
s/\\/backslash CR
how to use the backslash.
```

If you want to add text without changing the rest of the line, the & (ampersand) provides a useful shortcut. The & repeats the old text in the replacement pattern, so you do not have to type the pattern twice. For example:

```
p CR
The neanderthal skeletal remains
s/thal/& man's/ CR
p CR
The neanderthal man's skeletal remains
```

Ed automatically remembers the last string of characters in a search pattern or the old text in a substitution. However, you must prompt **ed** to repeat the replacement characters in a substitution with the **%** sign. The **%** sign allows you to make the same substitution on multiple lines without requesting a global substitution. For example, to change the word **money** to the word **gold**, repeat the last substitution from line 1 on line 3, but not on line 4.

```
1,$n CR
 1      I love money
 2      I need food
 3      I use money
 4      The IRS wants my money
 1s/money/gold CR
 I love gold
 3s//% CR
 I use gold
 1,$n CR
 1      I love gold
 2      I need food
 3      I use gold
 4      The IRS wants my money
```

Ed automatically remembers the word *money* (the old text to be replaced), so that string does not have to be repeated between the first two delimiters. The **%** sign tells **ed** to use the last replacement pattern, *gold*.

Ed tries to match the first occurrence of one of the characters enclosed in brackets and substitute the specified old text with new text. The brackets can be at any position in the pattern to be replaced.

In the following example, **ed** changes the first occurrence of the numbers **6**, **7**, **8**, or **9** to **4** on each line in which it finds one of those numbers:

```
1,$p CR
Monday      33,000
Tuesday     75,000
Wednesday   88,000
Thursday    62,000
1,$s/[6789]/4 CR
Monday      33,000
Tuesday     45,000
Wednesday   48,000
Thursday    42,000
```

The next example deletes the *Mr* or *Ms* from a list of names:

```
1,$p CR
Mr Arthur Middleton
Mr Matt Lewis
Ms Anna Kelley
Ms M. L. Hodel
1,$s/M[rs] // CR
1,$p CR
Arthur Middleton
Matt Lewis
Anna Kelley
M. L. Hodel
```

If a ^ (circumflex) is the first character in brackets, **ed** interprets it as an instruction to match characters

that are NOT within the brackets. However, if the circumflex is in any other position within the brackets, **ed** interprets it literally, as a circumflex.

```
1,$p CR
grade  A   Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  D   Jogging
grade  C   Tennis
1,$s/grade [^AB]/grade A CR
1,$p CR
grade  A   Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  A   Jogging
grade  A   Tennis
```

Whenever you use special characters as wild cards in the text to be changed, remember to use a unique pattern of characters. In the above example, if you had used only

```
1,$s/[^AB]/A CR
```

you would have changed the *g* in the word *grade* to *A*. Try it.

Experiment with these special characters. Find out what happens (or does not happen) if you use them in different combinations.

The special characters used for search or substitute patterns are summarized in the table below.

Ed Special Characters

.	Match any one character in a search or substitute pattern
*	Match zero or more occurences of the preceding character in a search or substitute pattern
.*	Match zero or more occurences of any characters following the full stop or period (.)
^	Match the beginning of the line in the substitute pattern to be replaced, or in a search pattern
$	Match the end of the line in the substitute pattern to be replaced
\	Take away the special meaning of the special character that follows in the substitute or search pattern
&	Repeat the old text to be replaced in the new text replacement pattern
%	Match the last replacement pattern
[...]	Match the first occurence of the character in the brackets
[^...]	Match the first occurence of the character that is NOT in the brackets

EXERCISE 5

1. Create a file that contains the following lines of text.

```
A       Computer Science
D       Jogging
C       Tennis
```

What happens if you try this command line:

```
1,$s/[^AB]/A/ CR
```

Undo the above command. How can you make the *C* and *D* unique? (Hint: they are at the beginning of the line, in the position shown by the ^.) Do not be afraid to experiment!

2. Insert the following line above line 2:

```
These are not really my grades.
```

Using brackets and the ^ character, create a search pattern that you can use to locate the line you inserted. There are several ways to address a line. When you edit text, use the way that is quickest and easiest for you.

3. Add the following lines to your file:

```
I love money
I need money
The IRS wants my money
```

Now use one command to change them to:

```
It's my money
It's my money
The IRS wants my money
```

Using two command lines, do the following: change the word on the first line from *money* to *gold*, and change the last two lines from *money* to *gold* without using the words *money* or *gold* themselves.

4. How can you change the line

   1020231020

   to

   10202031020

   without repeating the old digits in the replacement pattern?

5. Create a line of text containing the following characters.

   * . \ & % ^ *

   Substitute a letter for each character. Do you need to use a backslash for every substitution?

MOVING TEXT

You have now learned to address lines, create and delete text, and make substitutions. **Ed** has one more set of versatile and important commands. You can move, copy, or join lines of text in the editing buffer. You can also read in text from a file that is not in the editing buffer, or write lines of the file in the buffer to another file in the current directory. The commands that move text are:

m move lines of text

t copy lines of text

j join contiguous lines of text

w write lines of text to a file

r read in the contents of a file

Moving Lines of Text

The **m** command allows you to move blocks of text to another place in the file. The general format is:

[*address1*,*address2*]**m**[*address3*]

The components of this command line include:

address1,*address2*	The range of lines to be moved. If only one line is moved, only *address1* is given. If no address is given, the current line is moved.
m	The move command.
address3	Place the text after this line.

Try the following example to see how the command works. Create a file that contains these three lines of text:

```
I want to move this line.
I want the first line
below this line.
```

Type:

```
1m3 CR
```

Ed will move line 1 below line 3.

```
I want to move this line.
```

```
I want the first line
below this line
I want to move this line.
```

The next screen shows how this will appear on your terminal:

```
1,$p CR
I want to move this line.
I want the first line
below this line.
1m3 CR
1,$p CR
I want the first line
below this line.
I want to move this line.
```

If you want to move a paragraph of text, have *address1* and *address2* define the range of lines of the paragraph.

In the following example, a block of text (lines 8 through 12) is moved below line 65. Notice the **n** command that prints the line numbers of the file:

```
8,12n CR
8       This is line 8.
9       It is the beginning of a
10      very short paragraph.
```

```
11      This paragraph ends
12      on this line.
64,65n CR
64      Move the block of text
65      below this line.
8,12m65 CR
59,65n CR
59      Move the block of text
60      below this line.
61      This is line 8.
62      It is the beginning of a
63      very short paragraph.
64      This paragraph ends
65      on this line.
```

How can you move lines above the first line of the file? Try the following command.

```
3,4m0 CR
```

When *address3* is 0, the lines are placed at the beginning of the file.

Copying Lines of Text

The copy command **t** (transfer) acts like the **m** command except that the block of text is not deleted at the original address of the line. A copy of that block of text is placed after a specified line of text. The general format of the command line is also similar.

The general format of the **t** command also looks like the **m** command.

[*address1*,*address2*]**t**[*address3*]

address1,address2 The range of lines to be copied. If only one line is copied, only *address1* is given. If no address is given, the current line is copied.

t The copy command.

address3 Place the copy of the text after this line.

The next example shows how to copy three lines of text below the last line.

```
    Safety procedures:

If there is a fire in the building:
Close the door of the room to seal off the fire


  Break glass of nearest alarm.
  Pull lever.
  Locate and use fire extinguisher.

    .
    .
    .
A chemical fire in the lab requires that you:


  Break glass of nearest alarm.
  Pull lever.
  Locate and use fire extinguisher.
```

The commands and **ed**'s responses to them are displayed in the next screen. Again, the **n** command displays the line numbers:

```
5,8n CR
5       Close the door of the room, to seal off the fire.
6       Break glass of nearest alarm.
7       Pull lever.
8       Locate and use fire extinguisher.
30n CR
30      A chemical fire in the lab requires that you:
6,8t30 CR
30,$n CR
30      A chemical fire in the lab requires that you:
31      Break glass of nearest alarm
32      Pull lever
33      Locate and use fire extinguisher
6,8n CR
6       Break glass of nearest alarm
7       Pull lever
8       Locate and use fire extinguisher
```

The text in lines 6 through 8 remains in place. A copy of those three lines is placed after line 50.

Experiment with **m** and **t** on one of your files.

Joining Contiguous Lines

The **j** command joins the current line with the following line. The general format is:

[*address1,address1*]**j**

The next example shows how to join several lines together. An easy way of doing this is to display the lines you want to join using **p** or **n**.

```
1,2p CR
```

```
Now is the time to join
the team.
p CR
the team.
lp CR
Now is the time to join
j CR
p CR
Now is the time to jointhe team.
```

Notice that there is no space between the last word (*join*) and the first word of the next line (*the*), and the last word (*play*). You must place a space between them by using the **s** command.

Writing Lines of Text to a File

The **w** command writes text from the buffer into a file. The general format is:

```
[address1,address2]w [filename]
```

address1,address2	The range of lines to be placed in another file. If you do not use *address1* or *address2*, the entire file is written into a new file.
w	The write command.
filename	The name of the new file that contains a copy of the block of text.

In the following example the body of a letter is saved in a file called *memo*, so that it can be sent to other people.

```
1,$n CR
1       March 17, 1986
2       Dear Kelly,
3       There will be a meeting in the
4       green room at 4:30 P.M. today.
5       Refreshments will be served.
3,6w memo CR
87
```

The **w** command places a copy of lines three through six into a new file called *memo*. **Ed** responds with the number of characters in the new file.

Problems

The **w** command overwrites preexisting files; it erases the current file and puts the new block of text in the file without warning you. If, in our example, a file called *memo* had existed before we wrote our new file to that name, the original file would have been erased.

In the section called *Special Commands*, later in this chapter, you will learn how to execute shell commands from **ed**. Then you can list the file names in the directory to make sure that you are not overwriting a file.

Another potential problem is that you cannot write other lines to the file *memo*. If you try to add lines 13 through 16, the existing lines (3 through 6) will be erased and the file will contain only the new lines (13 through 16).

Reading in the Contents of a File

The **r** command can be used to append text from a file to the buffer. The general format for the read command is:

[*address1*]**r** *filename*

address1 The text will be placed after the line *address1*. If *address1* is not given, the file is added to the end of the buffer.

r The read command.

filename The name of the file that will be copied into the editing buffer.

Using the example from the write command, the next screen shows a file being edited and new text being read into it.

```
1,$n CR
1       March 17, 1986
2       Dear Michael,
3       Are you free later today?
4       Hope to see you there.
3r memo CR
87
3,$n CR
3       Are you free later today?
4       There is a meeting in the
5       green room at 4:30 P.M. today.
6       Refreshments will be served.
7       Hope to see you there.
```

Ed responds to the read command with the number of characters in the file being added to the buffer (in the example, *memo*).

It is a good idea to display new or changed lines of text to be sure that they are correct.

The **ed** commands for moving text are summarized in the table below.

Ed Text Moving Commands

m	Move lines of text
t	Copy lines of text
j	Join contiguous lines
w	Write text into a new file
r	Read in text from another file

EXERCISE 6

1. There are two ways to copy lines of text in the buffer: by issuing the copy command; or by using the write and read commands to first write text to a file and then read the file into the buffer.

   Writing to a file and then reading the file into the buffer is a longer process. Can you think of an example where this method would be more practical?

   What commands can you use to copy lines 10 through 17 of file *exer* into the file *exer6* at line 7?

2. Lines 33 through 46 give an example that you want placed after line 3, and not after line 32. What command performs this task?

3. Say you are on line 10 of a file and you want to join lines 13 and 14. What commands can you issue to do this?

OTHER USEFUL COMMANDS AND INFORMATION

There are four other commands and a special file that will be useful to you during editing sessions.

h,H access the help commands, which provide error messages

l display characters that are not normally displayed

f display the current file name

! temporarily escape **ed** to execute a shell command

ed.hup When a system interrupt occurs, the **ed** buffer is saved in a special file named *ed.hup*.

Help Commands

You may have noticed when you were editing a file that **ed** responds to some of your commands with a **?**. The **?** is a diagnostic message issued by **ed** when it has found an error. The help commands give you a short message to explain the reason for the most recent diagnostic.

There are two help commands:

h Display a short error message that explains the reason for the most recent **?**.

H Place **ed** into help mode so that a short error message is displayed every time the **?** appears. (To cancel this request, type **H**.)

You know that if you try to quit **ed** without writing the changes in the buffer to a file, you will get a **?**. Do this now. When the **?** appears, type **h**:

```
q CR
?
h CR
warning: expecting `w'
```

The **?** is also displayed when you specify a new file name on the **ed** command line. Give **ed** a new file name. When the **?** appears, type **h** to find out what the error message means.

```
>ed newfile CR ? newfile h CR cannot open input file
```

This message means one of two things: either there is no file called *newfile* or there is such a file but **ed** is not allowed to read it.

As explained earlier, the **H** command responds to the **?** and then turns on the help mode of **ed**, so that **ed** gives you a diagnostic explanation every time the **?** is displayed subsequently. To turn off help mode, type **H** again. The next screen shows **H** being used to turn on help mode. Sample error messages are also displayed in response to some common mistakes:

```
>ed newfile CR
e newfile CR
? newfile
H CR
cannot open input file
/hello CR
?
illegal suffix
1,22p CR
?
line out of range
a CR
I am appending this line to the buffer.
. CR
```

```
s/$ tea party CR
?
illegal or missing delimiter
,$s/$/ tea party CR
?
unknown command
H CR
q CR
?
h CR
warning: expecting `w'
```

These are some of the most common error messages that you may encounter during editing sessions:

illegal suffix — **Ed** cannot find an occurrence of the search pattern *hello* because the buffer is empty.

line out of range — **Ed** cannot print any lines because the buffer is empty or the line specified is not in the buffer.

A line of text is appended to the buffer to show you some error messages associated with the **s** command.

illegal or missing delimiter — The delimiter between the old text to be replaced and the new text is missing.

unknown command — *addressl* was not typed in before the comma; **ed** does not recognize **,$**.

Help mode is then turned off and **h** is used to determine the meaning of the last **?** . While you are learning **ed**, you may want to leave help mode turned on. If so, use the **H** command. However, once you become adept at using **ed**, you will only need to see error messages

occasionally. Then you can use the **h** command.

Display Nonprinting Characters

If you are typing a **TAB** character, the terminal will normally display up to eight spaces (covering the space up to the next **TAB** setting. (Your **TAB** setting may be more or less than eight spaces.

If you want to see how many tabs you have inserted into your text, use the **l** (list) command. The general format for the **l** command is the same as for **n** and **p**.

[*address1*,*address2*]**l**

The components of this command line are:

address1,*address2*	The range of lines to be displayed. If no address is given, the current line will be displayed. If only *address1* is given, only that line will be displayed.
l	The command that displays the nonprinting characters along with the text.

The **l** command denotes tabs with a **>** (greater than) character. To type control characters, hold down the CTRL key and press the appropriate alphabetic key. The key that sounds the bell is **CTRL-g**. It is displayed as \07 which is the octal representation (the computer's code) for **CTRL-g**.

Type in two lines of text that contain a **CTRL-g** and a tab. Then use the **l** command to display the lines of text on your terminal.

```
a CR
Add a CTRL-g to this line. CR
Add a TAB to this line. CR
. CR
1,21 CR
Add a \07 CTRL-g to this line. CR
Add a > (TAB) to this line. CR
```

Did the bell sound when you typed **CTRL-g**?

The Current File Name

In a long editing session, you may forget the file name The **f** command will remind you which file is currently in the buffer. Or, you may want to preserve the original file that you entered into the editing buffer and write the contents of the buffer to a new file. In a long editing session, you may forget, and accidentally overwrite the original file with the customary **w** and **q** command sequence. You can prevent this by telling the editor to associate the contents of the buffer with a new file name while you are in the middle of the editing session. This is done with the **f** command and a new file name.

The format for displaying the current file name is **f** alone on a line:

```
f CR
```

To see how **f** works, enter **ed** with a file. For example, if your file is called *oldfile*, **ed** will respond as shown in the following screen:

```
>ed oldfile CR
```

```
323
f CR
oldfile
```

To associate the contents of the editing buffer with a new file name use this general format:

```
f newfile CR
```

If no file name is specified with the write command, ed remembers the file name given at the beginning of the editing session and writes to that file. If you do not want to overwrite the original file, you must either use a new file name with the write command, or change the current file name using the f command followed by the new file name. Because you can use f at any point in an editing session, you can change the file name immediately. You can then continue with the editing session without worrying about overwriting the original file.

The next screen shows the commands for entering the editor with *oldfile* and then changing its name to *newfile*. A line of text is added to the buffer and then the write and quit commands are issued.

>ed oldfile **CR** 323 f **CR** oldfile f newfile **CR** newfile a **CR**
Add a line of text. **CR** . **CR** w **CR** 343 q **CR**

Once you have returned to the shell, you can list your files and verify the existence of the new file, *newfile*. *newfile* should contain a copy of the contents of *oldfile* plus the new line of text.

Escape to the Shell

How can you make sure you are not overwriting an existing file when you write the contents of the editor to a new file name? You need to return to the shell to list your files. The ! allows you to temporarily return to the shell, execute a shell command, and then return to the current line of the editor.

The general format for the escape sequence is:

```
!shell command line
shell response to the command line
!
```

When you type the ! as the first character on a line, the shell command must follow on that same line. The program's response to your command will appear as the command is running. When the command has finished executing, the ! will be appear alone on a line. This means that that you are back in the editor at the current line.

For example, if you want to return to the shell to find out the correct date, type ! and the shell command **date**.

```
p CR
This is the current line
! date CR
Tue  Apr 1  14:24:22. EST  1986
!
p CR
This is the current line.
```

The screen first displays the current line. Then the command is given to temporarily leave the editor and display the date. After the date is displayed, you are

returned to the current line of the editor.

If you want to execute more than one command on the shell command line, see the discussion on ; in either of the shell tutorials in this manual.

Recovering From System Interrupts

What happens if you are creating text in **ed** and there is an interrupt to the system, you are accidentally hung up on the system, or your terminal is unplugged? When an interrupt occurs, the X/OS system tries to save the contents of the editing buffer in a special file named *ed.hup*. Later you can retrieve your text from this file in one of two ways. First, you can use a shell command to move *ed.hup* to another file name, such as the name the file had while you were editing it (before the interrupt). Second, you can enter **ed** and use the **f** command to rename the contents of the buffer. An example of the second method is shown in the following screen:

```
>ed ed.hup CR
928
f myfile CR
myfile
```

If you use the second method to recover the contents of the buffer, be sure to remove the *ed.hup* file afterward.

Conclusion

You now are familiar with many useful commands in **ed**. The commands that were not discussed in this tutorial, such as **G**, **P**, **Q** and the use of **( )** and **{ }**, are discussed on the *ed*(1) entry of the *Utilities Reference Manual*. You can experiment with these commands and try them to see what tasks they perform.

The functions of the commands introduced in this section are summarized in the table below.

Other Useful Ed Commands

h	Display a short error message for the preceding diagnostic ?
H	Turn on Help mode. An error message will be given with each diagnostic ?. The second **H** turns off Help mode
l	Display non printing characters in the text
f	Display the current filename
f *newfile*	Change the current filename associated with the editing buffer to *newfile*
!*cmd*	Temporarily escape to the shell to execute the shell command *cmd*
ed.hup	The editing buffer is saved in *ed.hup* if the terminal is hung up before a write command

EXERCISE 7

1. Create a new file called *newfile1*. Access **ed** and change the file's name to *current1*. Then create some text and write and quit **ed**. Run the **ls** shell command to verify that there is not a file called *newfile1* in your directory. If you do the shell command **ls**, you will see the directory does not contain a file called *newfile1*.

2. Create a file named *file1*. Append some lines of text to the file. Leave append mode but do not write the file. Turn off your terminal. Then turn on your terminal and log in again. Issue the **ls** command in the shell. Is there a new file called *ed.hup*? Place *ed.hup* in **ed**. How can you change the current file name to *file1*? Display the contents of the file

Are the lines the same lines you created before you turned off your terminal?

3. While you are in **ed**, temporarily escape to the shell and send a mail message to yourself.

ANSWERS TO EXERCISES

EXERCISE 1

1-1.

```
>ed junk CR
? junk
a CR
Hello world. CR
. CR
w CR
12
q CR

>
```

1-2.

```
>ed junk CR
12
1,$p CR
Hello world. CR
q CR

>
```

The system did not respond with the warning question mark because you did not make any changes to the buffer.

1-3.

```
>ed junk CR
12
a CR
Wendy's horse came through the window. CR
. CR
```

```
1,$p CR
Hello world.
Wendy's horse came through the window.
q CR
?
w stuff CR
60
q CR

>
```

EXERCISE 2

2-1.

```
>ed towns CR
? towns
a CR
My kind of town is CR
Chicago CR
Like being no where at all in CR
Toledo CR
I lost those little town blues in CR
New York CR
I lost my heart in CR
San Francisco CR
I lost $$ in CR
Las Vegas CR
. CR
w CR
164
```

2-2.

```
3 CR
Like being no where at all in
```

2-3.

```
-2,+3p CR
My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
```

2-4.

```
.= CR
6
6 CR
New York
```

2-5.

```
$ CR
Las Vegas
```

2-6.

```
?town CR
I lost those little town blues in
? CR
My kind of town is
```

2-7.

```
g/in CR
My kind of town is
Like being no where at all in
I lost those little town blues in
I lost my heart in
I lost $$ in
```

```
v/in CR
Chicago
Toledo
New York
San Francisco
Las Vegas
```

EXERCISE 3

3-1.

```
>ed ex3 CR
?ex3
i CR
?
q CR
```

The ? after the i means there is an error in the command. There is no current line before which text can be inserted.

3-2.

```
>ed towns CR
164
.n CR
10 Las Vegas
3i CR
Illinois CR
. CR
.i CR
or CR
Naperville CR
. CR
$i CR
hotels in CR
1,$n CR
 1  my kind of town is
```

```
 2  Chicago
 3  or
 4  Naperville
 5  Illinois
 6  Like being no where at all in
 7  Toledo
 8  I lost those little town blues in
 9  New York
10  I lost my heart in
11  San Francisco
12  I lost $$ in
13  hotels in
14  Las Vegas
```

3-3.

```
1,5n CR
1  My kind of town is
2  Chicago
3  or
4  Naperville
5  Illinois
2,5c CR
London CR
. CR
1,3n CR
1  My kind of town is
2  London
3  Like being no where at all
```

3-4.

```
. CR
Like being no where at all
/Tol CR
Toledo
c CR
Peoria CR
. CR
```

```
. CR
Peoria
```

3-5.

```
. CR
/New Y/c CR
Iron City CR
. CR
. CR
Iron City
```

Your search string need not be the entire word or line. It only needs to be unique.

EXERCISE 4

4-1.

```
v/little town/s/town/city CR
My kind of city is
London
Like being no where at all in
Peoria
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas
```

The line

```
I lost those little town blues in
```

was not printed because it was NOT addressed by the **v** command.

4-2.

```
. CR
Las Vegas
s?Las Vegas?Toledo CR
Toledo
```

4-3.

```
?lost?s??found CR
I found $$ in
```

4-4.

```
/no?s??NO CR
?
/no/s//NO CR
Like being NO where at all in
```

You cannot mix delimiters such as / and ? in a command line.

The substitution command on line 9 produced this output:

```
I found $$ inBig $
```

It did not work correctly because the $ sign is a special character in **ed**.

EXERCISE 5

5-1.

```
>ed filel CR
? filel
a CR
A  Computer Science CR
D  Jogging CR
C  Tennis CR
. CR
1,$s/[^AB]/A/ CR
1,$p CR
A  Computer Science
A  Jogging
A  Tennis
u CR
```

```
1,$s/^[^AB]/A CR
1,$p CR
A  Computer Science
A  Jogging
A  Tennis
```

5-2.

```
2i CR
These are not really my grades. CR
. CR
1,$p CR
A  Computer Science
These are not really my grades.
A  Tennis
A  Jogging
/^[^A] CR
These are not really my grades
?^[T] CR
```

```
These are not really my grades
```

5-3.

```
1,$p CR
I love money
I need money
The IRS wants my money
g/^I/s/I.*m /It's my m CR
It's my money
It's my money



/s/money/gold CR
It's my gold
2,$s//% CR
The IRS wants my gold
```

5-4.

```
s/10202/&0 CR
10202031020
```

5-5.

```
a CR
* . \ & % ^ * CR
. CR
s/*/a CR
a . \ & % ^ *
s/*/b CR
a . \ & % ^ b
```

Because there were no preceding characters, * substituted for itself.

```
s/\./c CR
a c \ & % ^ b
s/\\/d CR
a c d & % ^ b
s/&/e CR
a c d e % ^ b
s/%/f CR
a c d e f ^ b
```

The **&** and **%** are only special characters in the replacement text.

```
s/\^/g CR
a c d e f g b
```

EXERCISE 6

6-1. Any time you have lines of text that you may want to have repeated several times, it may be easier to write those lines to a file and read in the file at those points in the text.

If you want to copy the lines into another file you must write them to a file and then read that file into the buffer containing the other file.

```
ed exer CR
725
10,17 w temp CR
210
q CR
ed exer6 CR
305
7r temp CR
210
```

The file *temp* can be called any file name.

6-2.

```
33,46m3 CR
```

6-3.

```
.= CR
10
13p CR
This is line 13.
j CR
.p CR
This is line 13.and line 14.
```

Remember that .= gives you the current line.

EXERCISE 7

7-1.

```
>ed newfile1 CR
? newfile1
f currentl CR
currentl
a CR
This is a line of text CR
Will it go into newfile1 CR
or into currentl CR
. CR
w CR
66
q CR

>ls CR
```

```
bin
current1
```

7-2.

```
ed file1 CR
? file1
a CR
I am adding text to this file. CR
Will it show up in ed.hup? CR
. CR
```

Turn off your terminal.

Log in again.

```
>ed ed.hup CR
58
f file1 CR
file1
1,$p CR
I am adding text to this file.
Will it show up in ed.hup?
```

7-3.

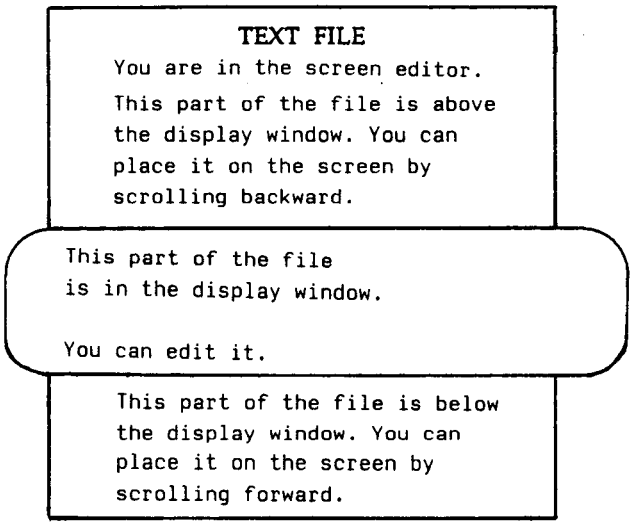
```
>ed file1 CR
58
! mail mylogin CR
You will get mail when CR
you are done editing! CR
. CR
!
```

VI - a screen editor

INTRODUCTION

This chapter is a tutorial-style introduction to the screen editor, **vi** (short for visual editor). The **vi** editor is a powerful and sophisticated tool for creating and editing files. It is designed for use with a video display terminal which is used as a window through which you can view the text of a file. A few simple commands allow you to make changes to the text that are quickly reflected on the screen.

The **vi** editor displays from one to many lines of text. It allows you to move the cursor to any point on the screen or in the file (by specifying places such as the beginning or end of a word, line, sentence, paragraph, or file) and create, change, or delete text from that point. You can also use some line editor commands, such as the powerful global commands that allow you to change multiple occurrences of the same character string by issuing one command. To move through the file, you can scroll the text forward or backward, revealing the lines below or above the current window, as shown in the figure below.



Note that not all terminals have text scrolling capability; whether or not you can take advantage of **vi**'s scrolling feature depends on what type of terminal you have.

There are more than 100 commands within **vi**. This chapter covers the basic commands that will enable you to use **vi** simply but effectively. Specifically, it explains how to do the following tasks:

- set up your terminal so that **vi** is accessible
- enter **vi**, create text, delete mistakes, write the text to a file, and quit
- move text within a file
- electronically cut and paste text
- use special commands and shortcuts
- temporarily escape to the shell to execute shell commands
- use line editing commands available within **vi**
- edit several files in the same session
- recover a file lost by an interruption to an editing session
- change your shell environment to set your terminal configuration and an automatic carriage return

SYNTAX

---

vi [-rfile] [-l] [-wn] [-R] [+command] name ...

view [-rfile] [-l] [-wn] [-R] [+command] name ...

vedit [-rfile] [-l] [-wn] [-R] [+command] name ...

DESCRIPTION

Vi supports the following options and arguments:

-r*file* recovers a specified file after a system or editor crash. If no *file* is specified, a list of recovered files is given.

-l activates LISP mode, to give appropriate indentation for LISP code. Note that the **( )**, **{ }**, **[[** and **]]** commands in **vi** and **open** are modified when this option is used.

-w*n* sets the default window size to *n* lines.

-R sets **vi** to read-only mode, preventing an accidental overwriting of the file. See also **view**, below.

+*command* the specified **ex** command is interpreted before editing begins.

name specifies the file to edited.

Note that other versions of **vi** can be used: **vedit** is intended for beginners, while **view** is a read-only version.

EXAMPLES

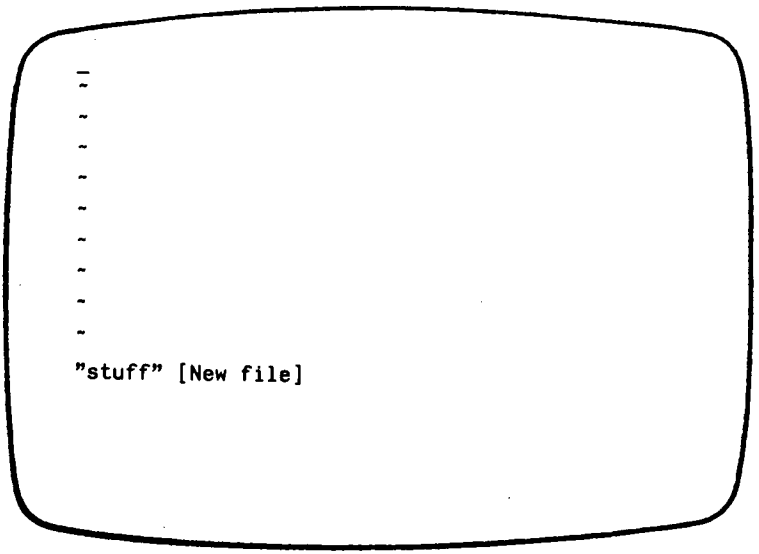
This section of the tutorial covers the facilities offered by the **vi** editor. Throughout, these features are illustrated with examples. The first stage is the creation of a file.

Remember that the symbol > represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter a line. Where key sequences such as **CTRL-d** occur, the **CTRL** or **CONTROL** key should be held down while the second key, in this case **d**, is pressed.

First, enter the editor; type **vi** and the name of the file you want to create or edit, as follows.

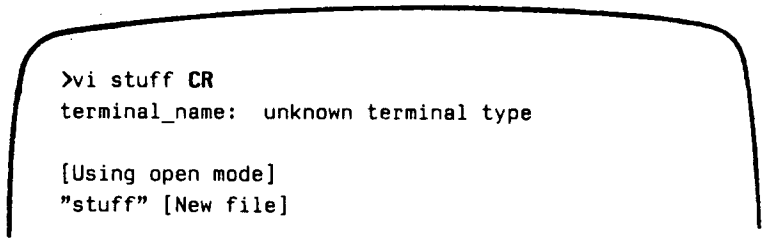
vi *filename* **CR**

For example, say you want to create a file called *stuff*. When you type the **vi** command with the file name *stuff*, **vi** clears the screen and displays a window in which you can enter and edit text.



The underscore (_) on the top line shows the cursor waiting for you to enter a command there. (On video display terminals the cursor may be a blinking underscore or a reverse color block.) Every other line is marked with a ~ (tilde), the symbol for an empty line.

If, before entering **vi**, you have forgotten to set your terminal configuration or have set it to the wrong type of terminal, you will see an error message instead.



You cannot set the terminal configuration while you are in the editor; you must be in the shell. Leave the editor by typing

```
:q CR
```

Then set the correct terminal configuration.

HOW TO CREATE TEXT: THE APPEND MODE

If you have successfully entered **vi**, you are in *command mode* and **vi** is waiting for your commands. How do you create text?

1. Press the **a** key to enter the append mode of **vi**. (Do not press the **CR** key.) You can now add text to the file. (Note that the **A** is not printed on the screen.)

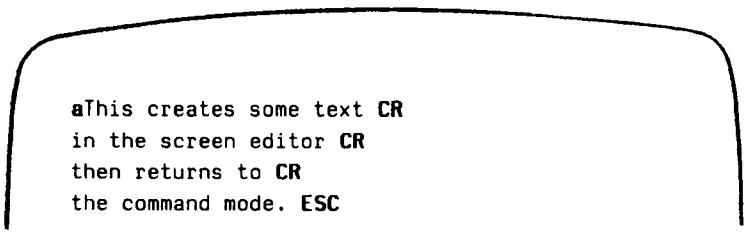
2. Type in some text. Note that as you approach the right margin, a bell sounds to remind you to press

the **CR** key. Terminals that do not have a bell may warn you in another way, such as by flashing the screen.

3. Remember, to begin a new line, press the **CR** key.

HOW TO LEAVE APPEND MODE

When you finish creating text, press the **ESC** key to leave append mode and return to command mode. Then you can edit any text you have created or write the text in the buffer to a file.



If you press the **ESC** key and a bell sounds, you are already in command mode. The text in the file is not affected by this, even if you press the **ESC** key several times.

EDITING TEXT: THE COMMAND MODE

To edit an existing file you must be able to add, change, and delete text. However, before you can perform those tasks you must be able to move to the part of the file you want to edit. **Vi** offers an array of commands for moving from page to page, between lines, and between specified points inside a line. These commands, along with commands for deleting and adding text, are introduced in this section.

HOW TO MOVE THE CURSOR

To edit your text, you need to move the cursor to the point on the screen where you will begin the correction. This is easily done with four keys that are grouped together on the keyboard: pressing

h moves the cursor one character to the left

j moves the cursor down one line

k moves the cursor up one line

l moves the cursor one character to the right

The **j** and **k** commands maintain the column position of the cursor. For example, if the cursor is on the seventh character from the left, when you type **j** or **k** it goes to the seventh character on the new line. If there is no seventh character on the new line, the cursor moves to the last character.

Many people who use **vi** find it helpful to mark these four keys with arrows showing the direction in which each key moves the cursor.

Some terminals have special cursor control keys that are marked with arrows. Use them in the same way you use the **h**, **j**, **k**, and **l** commands.

Watch the cursor on the screen while you press the keys **h**, **j**, **k**, and **l**. Instead of pressing a motion command key a number of times to move the cursor a corresponding number of spaces or lines, you can precede the command with the desired number. For example, to move two spaces to the right, you can press **l** twice or press the key sequence **2l**. To move up four lines, press **k** four times or enter **4k**. If you cannot go any farther in the direction you have requested, **vi** will sound a bell.

Now experiment with the **j** and **k** motion commands. First, move the cursor up seven lines. Type **7k**, and the cursor will move up seven lines above the current line. If there are less than seven lines above the current line, a bell will sound and the cursor will remain on the current line.

Now move the cursor down thirty-five lines. Type **35j** and **vi** will clear and redraw the screen. The cursor will be on the thirty-fifth line below the current line, appearing in the middle of the new window. If there are less than thirty-five lines below the current line, the bell will sound and the cursor will remain on the current line. Watch what happens when you type the next command. Type **35k**. Like most **vi** commands, the **h**, **j**, **k**, and **l** motion commands are silent; they do not appear on the screen as you enter them. The only time you should see characters on the screen is when you are in append mode and are adding text to your file. If the motion command letters appear on the screen, you are still in append mode. Press the **ESC** key to return to command mode and try the commands again.

Moving the Cursor to the Right or Left

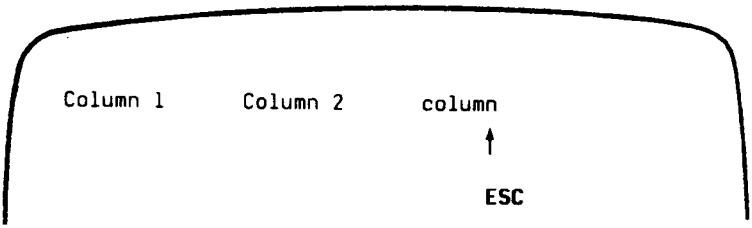
In addition to the motion command keys **h** and **l**, the **space bar** and the **BACKSPACE** key can be used to move the cursor right or left to a character on the current line.

space bar	move the cursor one character to the right
***n*space bar**	move the cursor *n* characters to the right
BACKSPACE	move the cursor one character to the left
***n*BACKSPACE**	move the cursor *n* characters to the left

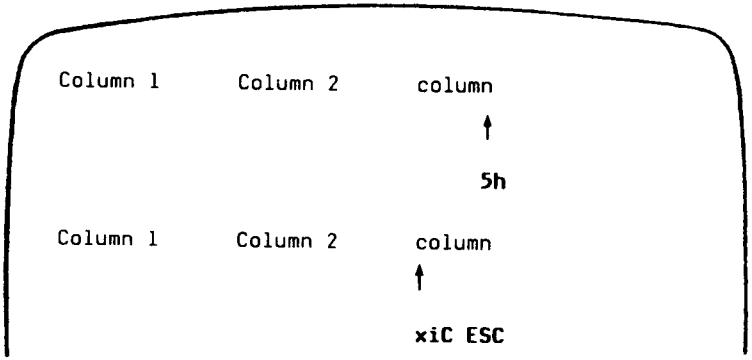
Try typing in a number before the command key. Notice that the cursor moves the specified number of characters to the left or right. In the example below, the cursor

movement is shown by the arrows.

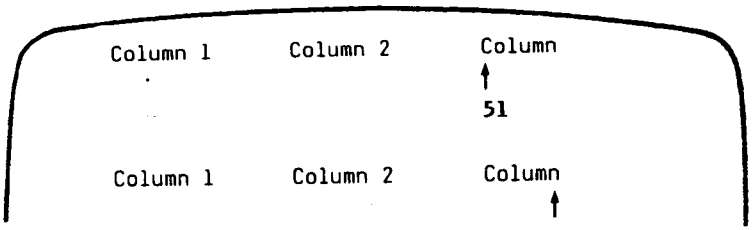
To move the cursor quickly to the right or left, prefix a number to the command. For example, suppose you want to create four columns in your screen. After you've finished typing the headings for the first three columns, you notice a typing mistake.



You want to correct your mistake before continuing. Exit insert mode and return to command mode by pressing the **ESC** key; the cursor will move to the letter **n**. Then use the **h** command to move back five spaces.



Erase the letter **c** by typing **x**. Then change to insert mode by pressing **i**, enter a letter **C**, followed by pressing the **ESC** key. Use the **l** motion command to return to your earlier position.



By now you may have discovered that you can move the cursor back and forth on a line by using the **space bar** and the **BACKSPACE** key. Here's a reminder:

space bar move the cursor one character to the right

*n***space bar** move the cursor *n* characters to the right

BACKSPACE move the cursor one character to the left

*n***BACKSPACE** move the cursor *n* characters to the left

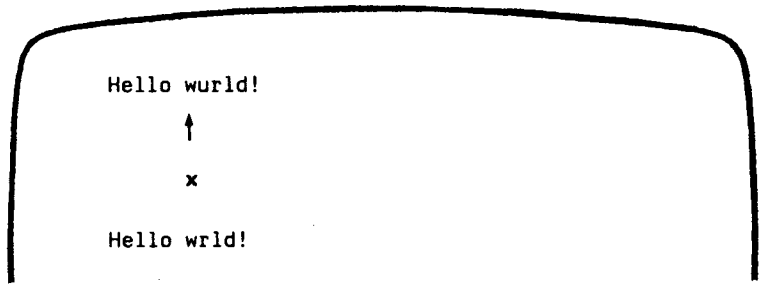
Again, you can specify a multiple space movement by typing a number before pressing the **space bar** or **BACKSPACE** key. The cursor will move the number of characters you request to the left or right.

HOW TO DELETE TEXT

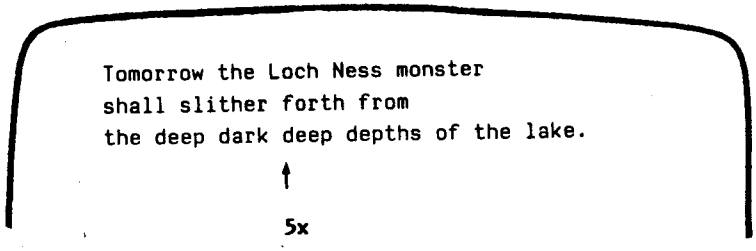
If you want to delete a character, move the cursor to that character and press the **x**. Watch the screen as you do so; the character will disappear and the line will readjust to the change. To erase three characters in a row, press **x** three times. In the following example, the arrows under the letters show the positions of the cursor.

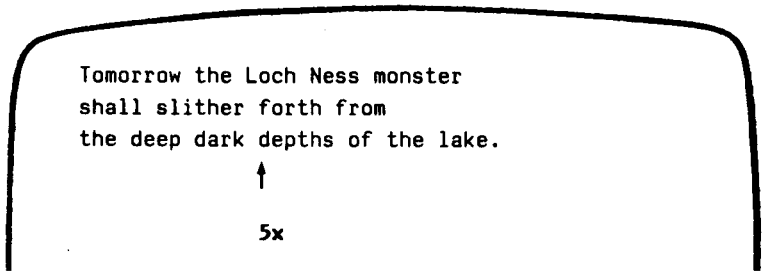
x delete one character

*n***x** delete *n* characters, where *n* is the number of characters you want to delete



Now try preceding **x** with the number of characters you want to delete. For example, delete the second occurrence of the word **deep** from the text shown in the following screen. Put the cursor on the first letter of the string you want to delete, and delete five characters (for the four letters of **deep** plus an extra space).

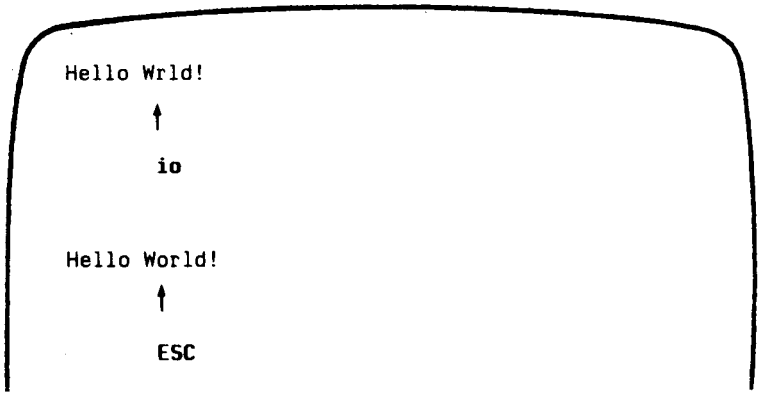




Notice that **vi** adjusts the text so that no gap appears in place of the deleted string. If, as in this case, the string you want to delete happens to be a word, you can also use the **vi** command for deleting a word. This command is described later in the section called *Word Positioning*.

HOW TO ADD TEXT

There are two basic commands for adding text: the insert command, **i**, and the append command, **a**. To add text with the insert command at a point in your file that is visible on the screen, move the cursor to that point by using **h**, **j**, **k**, and **l**. Then press **i** and start entering text. As you type, the new text will appear on the screen to the left of the character on which you put the cursor. That character and all characters to the right of the cursor will move right to make room for your new text. The **vi** editor will continue to accept the characters you type until you press the **ESC** key. If necessary, the original characters will even wrap around onto the next line.



You can use the append command in the same way. The only difference is that the new text will appear to the right of the character on which you put the cursor.

Later in this tutorial you will learn how to move around on the screen or scroll through a file to add or delete characters, words, or lines.

QUITTING VI

When you have finished your text, you will want to write the buffer contents to a file and return to the shell. To do this, hold down the **SHIFT** key and press **Z** twice (**ZZ**). The editor remembers the file name you specified with the **vi** command at the beginning of the editing session, and moves the buffer text to the file of that name. A notice at the bottom of the screen gives the file name and the number of lines and characters in the file. Then the shell gives you a prompt.

```
aThis is a test file. CR
I am adding text to CR
a temporary buffer and CR
now it is perfect. CR
I want to write this file, CR
and return to the shell. ESC ZZ
~
~
~
~
"stuff" [New file] 7 lines, 151 characters

>
```

You can also use the **:w** and **:q** commands of the line editor for writing and quitting a file. (Line editor commands begin with a colon and appear on the bottom line of the screen.) The **:w** command writes the buffer to a file. The **:q** command leaves the editor and returns you to the shell. You can type these commands separately or combine them into the single command **:wq**. It is easier to combine them.

```
aThis is a test file. CR
I am adding text to CR
a temporary buffer and CR
now it is perfect. CR
I want to write this file, CR
and return to the shell. ESC
~
~
~
~
~
:wq CR
```

The table below summarizes the basic commands you need to enter and use **vi**.

Command	Function
vi *filename*	enter **vi** to edit the file called *filename*
a	add text after the cursor
h	move one character to the left
j	move down one line
k	move up one line
l	move one character to the right
x	delete a character
CR	carriage return
ESC	leave append mode, and return to **vi** command mode
:w	write to a file
:q	quit **vi**
:wq	write to a file and quit **vi**
ZZ	write to a file and quit **vi**

EXERCISE 1

Answers to the exercises are given at the end of this chapter. However, keep in mind that there is often more than one way to perform a task in **vi**. If your method works, it is correct.

As you give commands in the following exercises, watch the screen to see how it changes or how the cursor moves.

1. If you have not logged in yet, do so now.

2. Enter **vi** and append the following five lines of text to a new file called *exer1*.

```
This is an exercise!
Up, down,
left, right,
build your terminal's
muscles bit by bit
```

3. Move the cursor to the first line of the file and the seventh character from the right. Notice that as you move up the file, the cursor moves in to the last letter of the file, but it does not move out to the last letter of the next line.

4. Delete the seventh and eighth characters from the right.

5. Move the cursor to the last character on the last line of the text.

6. Append the following new line of text:

```
and byte by byte
```

7. Write the buffer to a file and quit **vi**.

8. Reenter **vi** and append two more lines of text to the file *exer1*. What does the notice at the bottom of the screen say once you have reentered **vi** to edit *exer1*?

MOVING THE CURSOR AROUND THE SCREEN

Until now you have been moving the cursor with the **h**, **j**, **k**, **l**, **BACKSPACE** key, and the **space bar**. There are several other commands that can help you move the cursor quickly around the screen. This section explains how to position the cursor in the following ways:

1. by characters on a line

2. by lines

3. by text objects

   - words

   - sentences

   - paragraphs

4. in the window

There are also commands that position the cursor within parts of the **vi** editing buffer that are not visible on the screen. These commands will be discussed in the next section, *Positioning the Cursor in Undisplayed Text*.

To follow this section of the tutorial, you should enter **vi** with a file that contains at least forty lines. If you do not have a file of that length, create one now. Remember, to execute the commands described here, you must be in command mode of **vi**. Press the **ESC** key to make sure that you are in command mode rather than append mode.

Positioning the Cursor on a Character

There are three ways to position the cursor on a character in a line.

1. by moving the cursor right or left to a character

2. by specifying the character at either end of the line

3. by searching for a character on a line

The first method was discussed earlier in this chapter under *Moving the Cursor to the Right or Left*. The following sections describe the other two methods.

Moving the Cursor to the Beginning or End of a Line

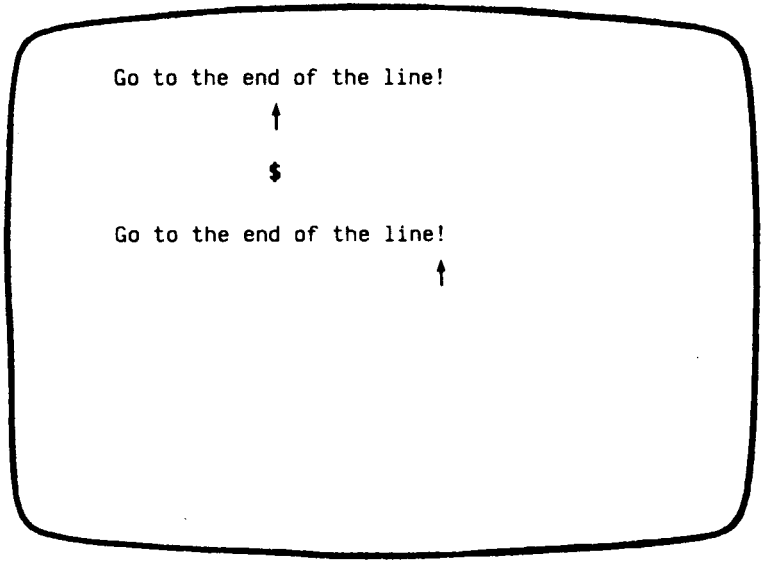
The second method of positioning the cursor on a line is by using one of three commands that put the cursor on the first or last character of a line.

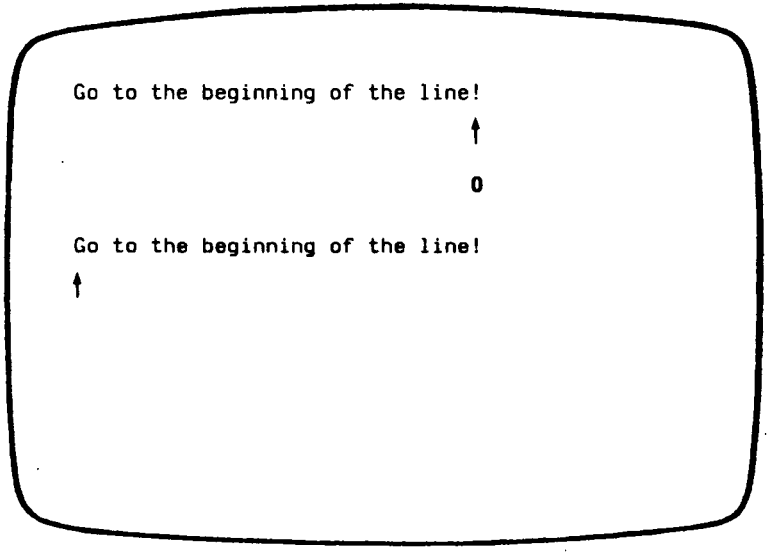
$ the dollar sign puts the cursor on the last character of a line

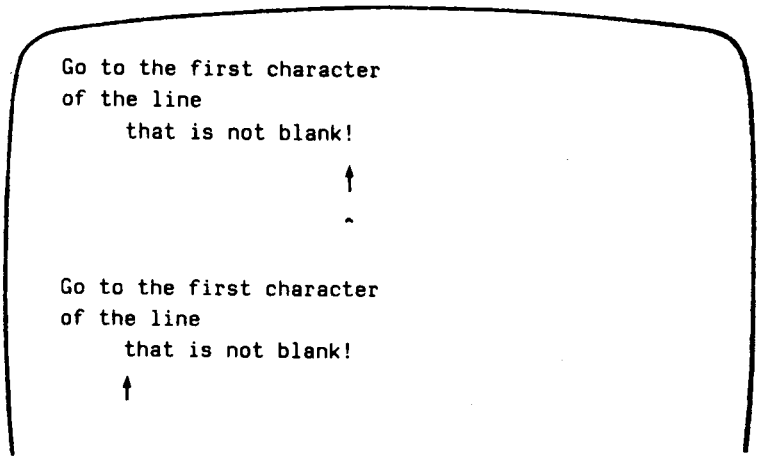
0 zero puts the cursor on the first character of a line

^ the circumflex puts the cursor on the first nonblank character of a line

The following examples show the movement of the cursor produced by each of these three commands.


Go to the end of the line!
$
Go to the end of the line!


Go to the beginning of the line!
0
Go to the beginning of the line!



Searching for a Character on a Line

The third way to position the cursor on a line is to search for a specific character on the current line. If the character is not found on the current line, a bell sounds and the cursor does not move. (There is also a command that searches a file for patterns. This will be discussed in the next section.) There are six commands you can use to search within a line: **f**, **F**, **t**, **T**, **;**, and **,**. You must specify a character after all of them except the **;** and **,** commands.

f*x* Move the cursor to the right to the specified character *x*.

F*x* Move the cursor to the left to the specified character *x*.

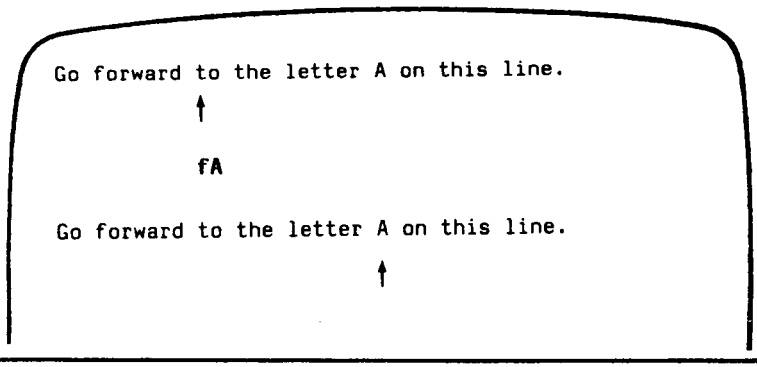
t*x* Move the cursor right to the character just before the specified character *x*.

T*x* Move the cursor left to the character just after the specified character *x*.

; Continue the search specified in the last command, in the same direction. The ; remembers the character and seeks out the next occurrence of that character on the current line.

, Continue the search specified in the last command, in the opposite direction. The , remembers the character and seeks out the previous occurrence of that character on the current line.

For example, in the following screen **vi** searches to the right for the first occurrence of the letter **A** on the current line.



Try the search commands on one of your files.

Line positioning

Besides the **j** and **k** commands that you have already used, the **+**, **-**, and **CR** commands can be used to move the cursor to other lines.

The Minus Sign Motion Command

The - command moves the cursor up a line, positioning it at the first nonblank character on the line. To move more than one line at a time, specify the number of lines you want to move before the - command. For example, to move the cursor up thirteen lines, type **13-**. The cursor will move up thirteen lines. If some of those lines are above the current window, the window will scroll up to reveal them. This is a rapid way to move quickly up a file.

Now try to move up 100 lines. Type **100-**. What happened to the window? If there are less then 100 lines above the current line a bell will sound, telling you that you have made a mistake, and the cursor will remain on the current line.

The Plus Sign Motion Command

The plus sign command, **+**, or the **CR** command moves the cursor down a line. Specify the number of lines you want to move before the + command. For example, to move the cursor down nine lines, type **9+**. The cursor will move down nine lines. If some of those lines are below the current screen, the window will scroll down to reveal them.

Now try to do the same thing by pressing the **CR** key. Were the results the same as when you pressed the **+** key?

Word Positioning

The **vi** editor considers a word to be a string of characters that may include letters, numbers, or underscores. There are six word positioning commands: **w**, **b**, **e**, **W**, **B**, and **E**. The lower case commands (**w**, **b**, and **e**) treat any character other than a letter, digit, or underscore as a delimiter, signifying the beginning or

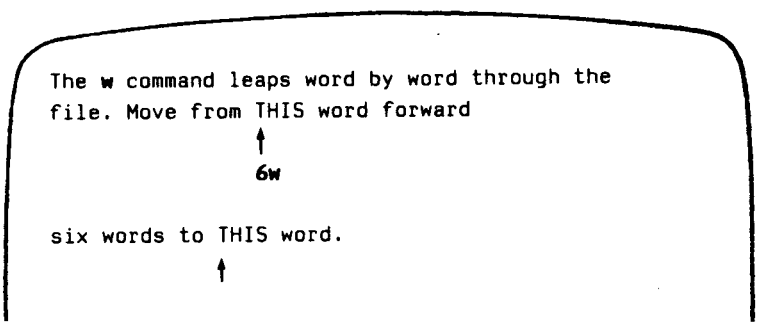
end of a word. Punctuation before or after a blank is considered a word. The beginning or end of a line is also a delimiter.

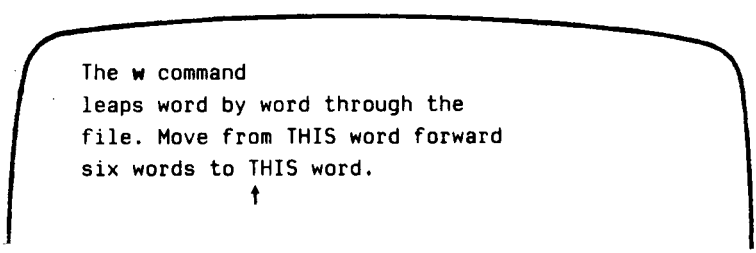
The upper case commands (**W**, **B**, and **E**) treat punctuation as part of the word; words are delimited by blanks and newlines only.

The following is a summary of the word positioning commands.

w Move the cursor forward to the first character in the next word. You may press **w** as many times as you want to reach the word you want, or you can prefix the necessary number to the **w**.

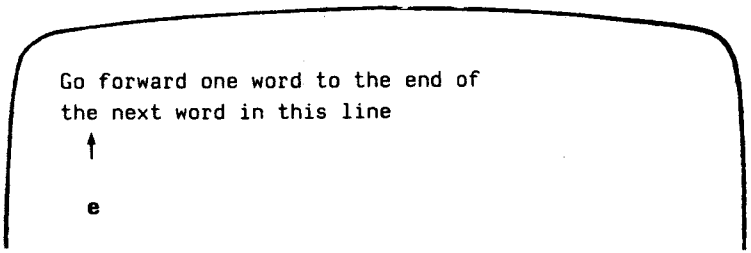
*n***w** Move the cursor forward *n* number of words to the first character of that word. The end of the line does not stop the movement of the cursor; instead, the cursor wraps around and continues counting words from the beginning of the next line.

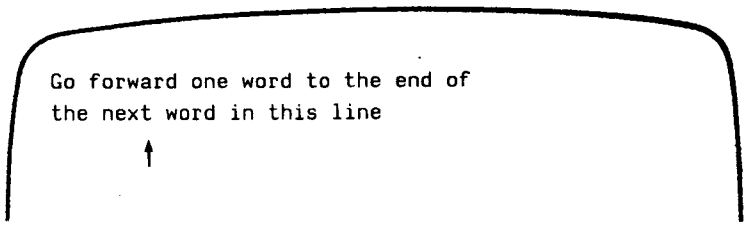


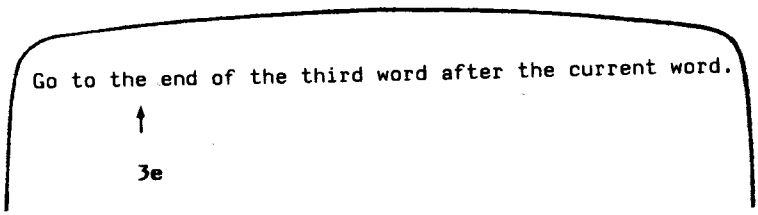


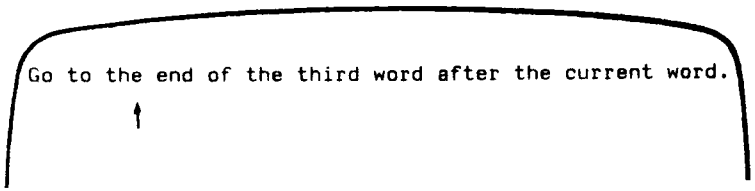
W Ignore all punctuation and move the cursor forward to the word after the next blank.

e Moves the cursor forward in the line to the last character in the next word.







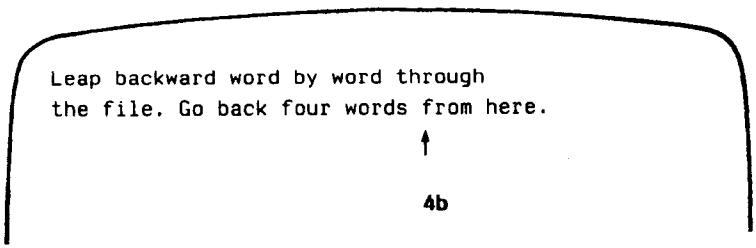


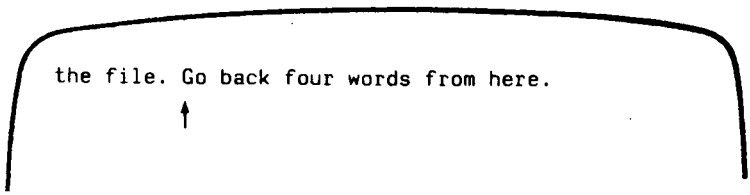
E Ignores all punctuation except blanks, delimiting words only by blanks.

b Move the cursor backward in the line to the first character of the previous word.

***n*b** Move the cursor backward *n* number of words to the first character of the *n*th word. The **b** command does not stop at the beginning of a line, but moves to the end of the line above and continues moving backward.

B Can be used just like the **b** command, except that it delimits the word only by blank spaces and newlines. It treats all other punctuation as letters of a word.





Positioning the Cursor by Sentences

The **vi** editor also recognizes sentences. In **vi** a sentence ends in ., ? or !. If these delimiters appear in the middle of a line, they must be followed by two blanks for **vi** to recognize them. You should get used to the **vi** convention of recognizing two blanks after a period as the end of a sentence, because it is often useful to be able to operate on a sentence as a unit.

You can move the cursor from sentence to sentence in the file with the ((open parenthesis) and) (close parenthesis) commands.

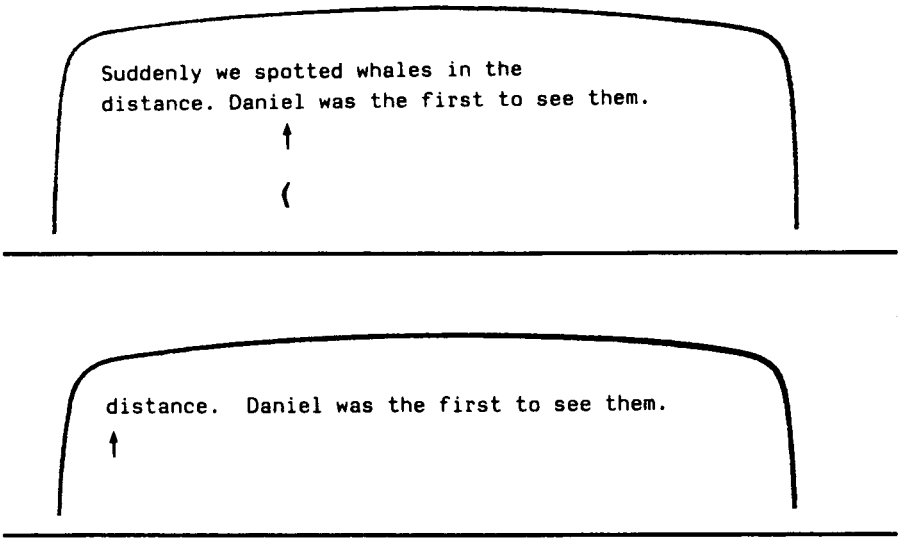
(Move the cursor to the beginning of the current sentence.

n(Move the cursor to the beginning of the *n*th sentence above the current sentence.

) Move the cursor to the beginning of the next sentence.

n) Move the cursor to the beginning of the *n*th sentence below the current sentence.

The example in the following screens shows how the open parenthesis moves the cursor around the screen.



Now repeat the command, preceding it with a number. For example, type 3(or 5).

Did the cursor move the correct number of sentences?

Positioning the Cursor by Paragraphs

Paragraphs are recognized by **vi** if they begin after a blank line. If you want to be able to move the cursor to the beginning of a paragraph (or later in this tutorial, to delete or change a whole paragraph), then make sure each paragraph ends in a blank line.

{ Move the cursor to the beginning of the current paragraph, which is delimited by a blank line above it.

n{ Move the cursor to the beginning of the *n*th paragraph above the current paragraph.

} Move the cursor to the beginning of the next paragraph.

***n*}** Move the cursor to the *n*th paragraph below the current line.

The following two screens show how the cursor can be moved to the beginning of another paragraph.

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
          ↑
          }

"Hey look!  Here come the whales!" he cried excitedly.
```

```
Suddenly, we spotted whales in the



distance.  Daniel was the first to see them.
←
"Hey look!  Here come the whales!" he cried excitedly.
```

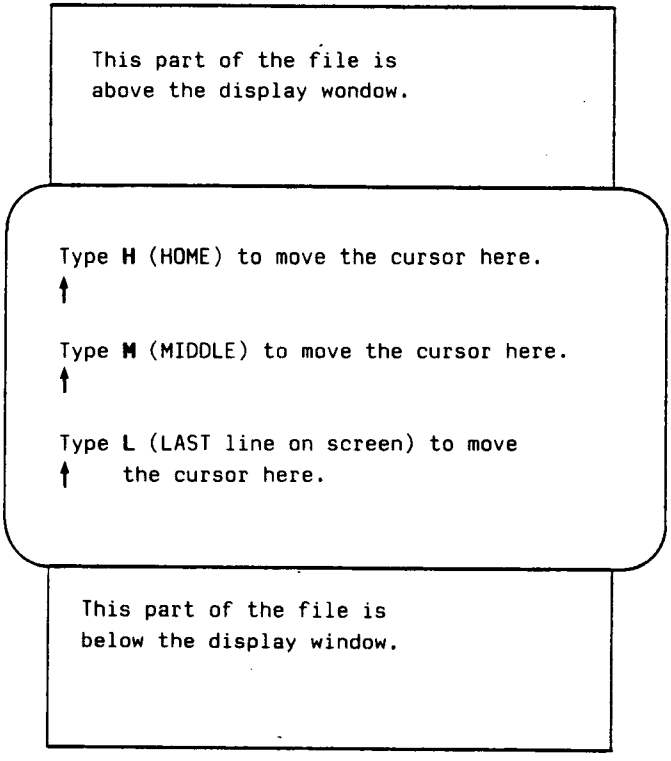
Positioning in the Window

The **vi** editor also provides three commands that help you position yourself in the window. Try out each command. Be sure to type them in upper case.

H Move the cursor to the first line on the screen.

M Move the cursor to the middle line on the screen.

L Move the cursor to the last line on the screen.



The next few pages of tables summarize the **vi** commands for moving the cursor by positioning it on a character, line, word, sentence, paragraph, or position on the screen. (Additional **vi** commands for moving the cursor are summarized in a later table.

Positioning on a Character [table 1/6]

h	Move the cursor one character to the left
l	Move the cursor one character to the right
BACKSPACE	Move the cursor one character to the left
space bar	Move the cursor one character to the right
f*x*	Move the cursor to the right to the specified character *x*
F*x*	Move the cursor to the left to the specified character *x*
t*x*	Move the cursor to the right to the character just before the specified character *x*
T*x*	Move the cursor to the left to the character just before the specified character *x*
;	Continue searching in the same direction on for the last character requested with **f**, **F**, **t**, or **T**. The **;** remembers the character and finds the next occurrence of it on the current line
,	Continue searching in opposite direction on the line for the last character requested with **f**, **F**, **t**, or **T**. The **,** remembers the character and finds the next occurrence of it on the current line

Positioning on a Sentence [table 2/6]

(	Move the cursor to the beginning of the current sentence
)	Move the cursor to the beginning of the next sentence

Positioning on a Line [table 3/6]

k	Move the cursor up to the same column in the previous line (if a character exists in that column
j	Move the cursor odown to the same column in the next line (if a character exists in that column)
-	Move the cursor up to the beginning of the previous line
+	Move the cursor down to the beginning of the next line
CR	Move the cursor down to the beginning of the next line

Positioning on a Word [table 4/6]

w	Move the cursor forward to the first character in the next word
W	Ignore all punctuation and move the cursor forward to the next line delimited only by

	blanks
b	Move the cursor backward one word to the first character of that word
B	Move the cursor to the left one word, which is delimited only by blanks
e	Move the cursor to the end of the current word
E	Delimit the words by blanks only. The cursor is placed on the last character before the next blank space, or end of the line

Positioning on a Paragraph [table 5/6]

{	Move the cursor to the beginning of the current paragraph
}	Move the cursor to the beginning of the next paragraph

Positioning in the Window [table 6/6]

H	Move the cursor to the first line of the screen (the home position)
M	Move the cursor to the middle line of the screen
L	Move the cursor to the last line of the screen

POSITIONING THE CURSOR IN UNDISPLAYED TEXT

How do you move the cursor to text that is not shown in the current editing window? One option is to use the **20j** or **20k** sequences. However, if you are editing a large file, you need to move quickly and accurately to another place in the file. This section covers those commands that can help you move around within the file in the following ways:

1. by scrolling forward or backward in the file

2. by going to a specified line in the file

3. by searching for a pattern in the file

Scrolling the Text

Four commands allow you to scroll the text of a file. The **CTRL-f**, obtained by holding down the key marked **CONTROL** or **CTRL**, and pressing **f**, and **CTRL-d** (**CTRL** and **d**) commands scroll the screen forward. The **CTRL-b** (**CTRL** and **b**) and **CTRL-u** (**CTRL** and **u**) commands scroll the screen backward.

The CTRL-f Command

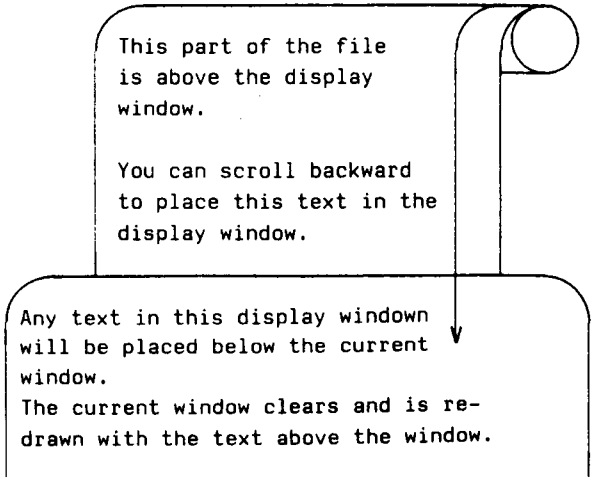
The **CTRL-f** command scrolls the text forward one full window of text below the current window. To do this **vi** clears the screen and redraws the window. The three lines that were at the bottom of the current window are placed at the top of the new window. If there are not enough lines left in the file to fill the window, the screen displays a ~ (tilde) to show that there are empty lines.

The CTRL-d Command

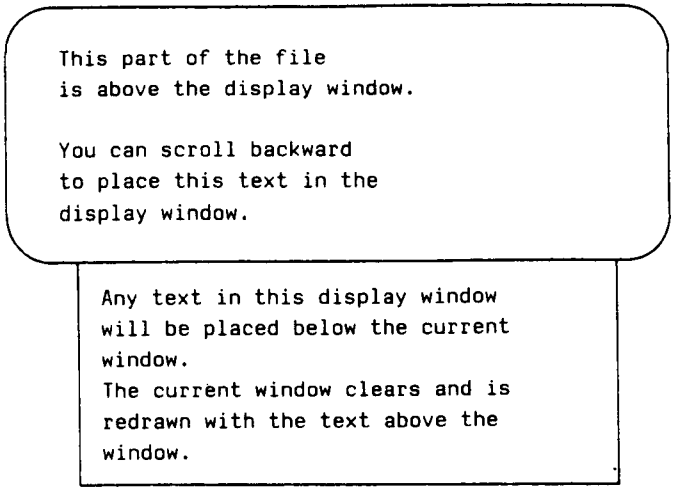
The **CTRL-d** command scrolls down a half screen to reveal text below the window. When you type **CTRL-d**, the text appears to be rolled up at the top and unrolled at the bottom. This allows the lines below the screen to appear on the screen, while the lines at the top of the screen disappear. If there are not enough lines in the file, a bell will sound.

The CTRL-b Command

The **CTRL-b** command scrolls the screen back a full window to reveal the text above the current window. To do this, **vi** clears the screen and redraws the window with the text that is above the current screen. Unlike the **CTRL-f** command, **CTRL-b** does not leave any reference lines from the previous window. If there are not enough lines above the current window to fill a full new window, a bell will sound and the current window will remain on the screen.



Now try scrolling backward. Type **CTRL-b** and **vi** clears the screen then draws a new one.



Any text that was in the display window is placed below the current window.

The CTRL-u Command

The **CTRL-u** command scrolls up a half screen of text to reveal the lines just above the window. The lines at the bottom of the window are erased. Now scroll down in the text, moving the portion below the screen into the window. Type **CTRL-u**. When the cursor reaches the top of the file, a bell sounds to notify you that the file cannot scroll further.

GO TO A SPECIFIED LINE

The **G** command positions the cursor on a specified line in the window; if that line is not currently on the screen, **G** clears the screen and redraws the window around it. If you do not specify a line, **G** goes to the last line of the file.

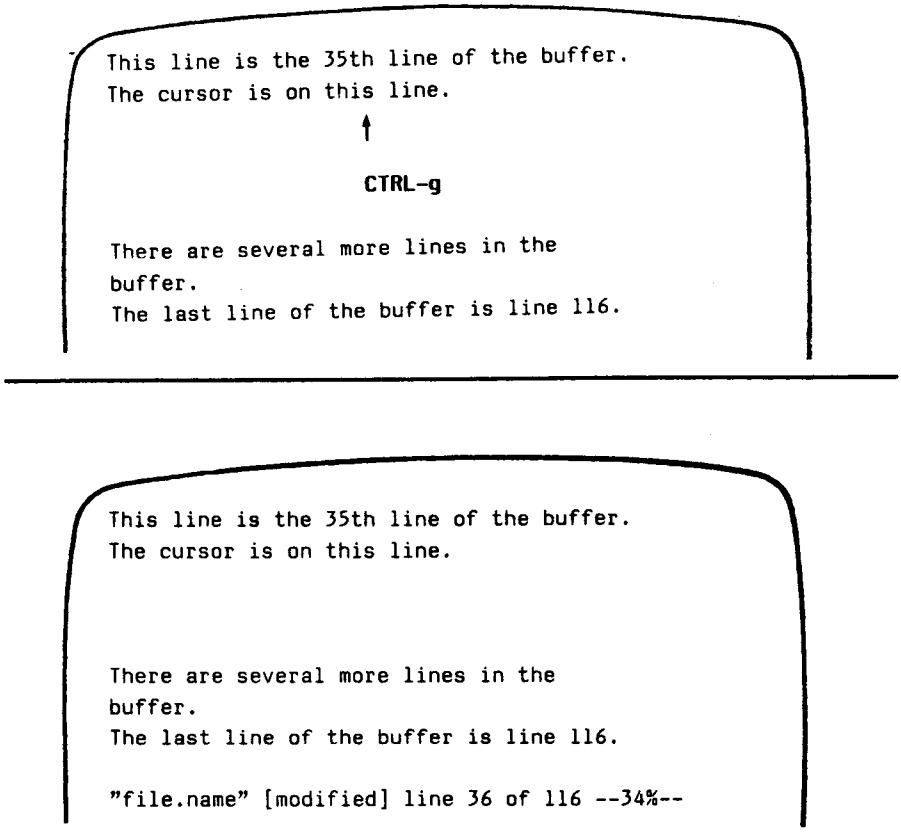
G go to the last line of the file

*n***G** go to the *n*th line of the file

LINE NUMBERS

Each line of the file has a line number corresponding to its position in the buffer. To get the number of a particular line, position the cursor on it and type **CTRL-g**. The **CTRL-g** command gives you a status notice at the bottom of the screen which tells you:

- the name of the file
- if the file has been modified
- the line number on which the cursor rests
- the total number of lines in the buffer
- the percentage of the total lines in the buffer represented by the current line



SEARCHING FOR A PATTERN OF CHARACTERS

The fastest way to reach a specific place in your text is by using one of the search commands: **/**, **?**, **n**, or **N**. These commands allow you to search forward or backward in the buffer for the next occurrence of a specified character pattern. The **/** and **?** commands are not silent; they appear as you type them, along with the search pattern, on the bottom of the screen. The **n** and **N** commands, which allow you to repeat the requests you made for a search with a **/** or **?** command, are silent.

The /, followed by a *pattern*(/*pattern*), searches forward in the buffer for the next occurrence of the characters in *pattern*, and puts the cursor on the first of those characters. For example, the command line

```
/Hello world CR
```

finds the next occurrence in the buffer of the words **Hello world** and puts the cursor under the **H**.

The ?, followed by a *pattern*(?*pattern*), searches backward in the buffer for the first occurrence of the characters in *pattern*, and puts the cursor on the first of those characters. For example, the command line

```
?data set design CR
```

finds the last occurrence in the buffer (before your current position) of the words *data set design* and puts the cursor under the *d* in *data*.

These search commands do not wrap around the end of a line while searching for two words. For example, say you are searching for the words *Hello world*. If *Hello* is at the end of one line and *world* is at the beginning of the next, the search command will not find that occurrence of *Hello World*.

However, they do wrap around the end or the beginning of the buffer to continue a search. For example, if you are near the end of the buffer, and the pattern for which you are searching (with the */pattern* command) is at the top of the buffer, the command will find the pattern.

The **n** and **N**commands allow you to continue searches you have requested with */pattern* or ?*pattern* without retyping them.

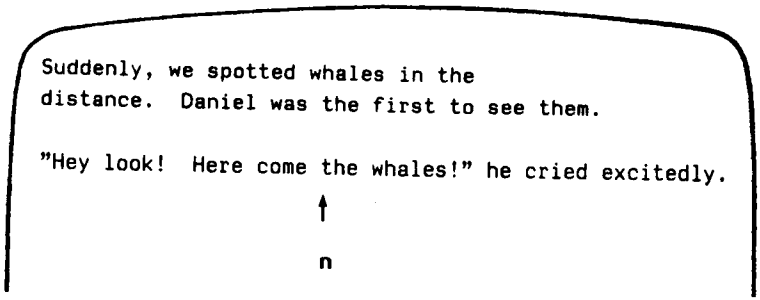
n Repeat the last search command.

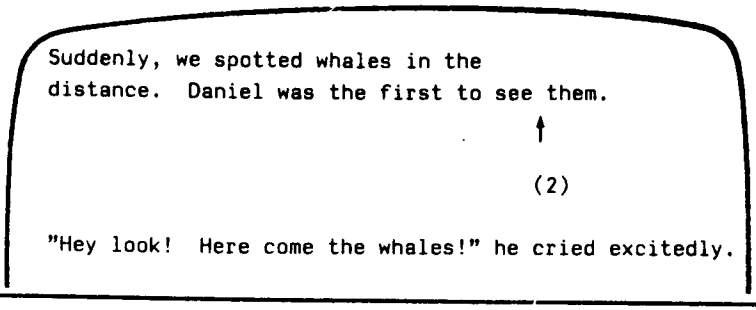
N Repeat the last search command in the opposite direction.

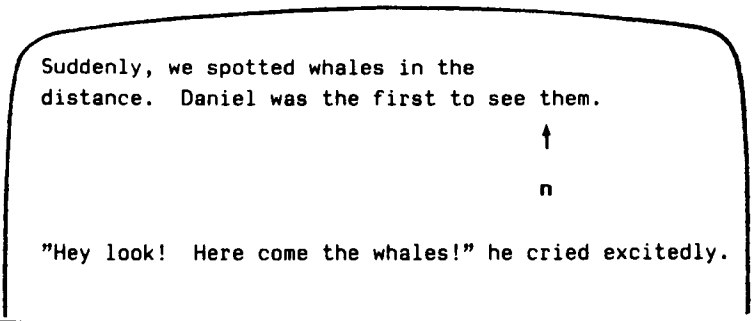
For example, say you want to search backward in the file for the three-letter pattern *the*. Initiate the search with **?***the* and continue it with **n**. The following screens offer a step-by-step illustration of how the **n** searches backward through the file and finds four occurrences of the character string *the*.

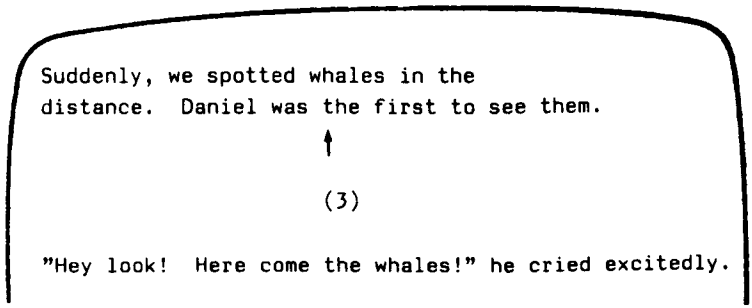
```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.

"Hey look!  Here come the whales!" he cried excitedly.


?the
```

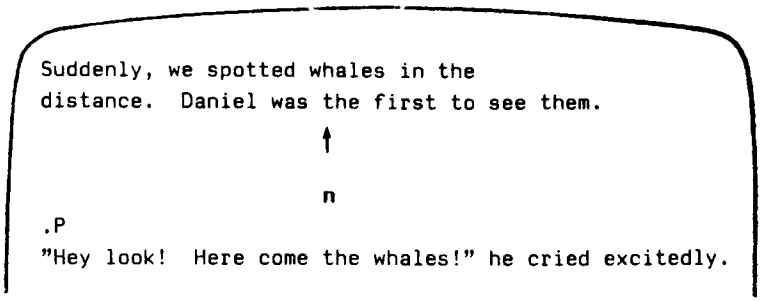
```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.
.P
"Hey look!  Here come the whales!" he cried excitedly.
                      ↑
                     (1)
```

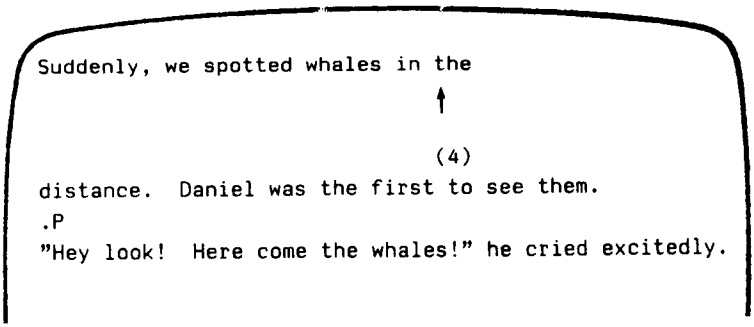

Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
"Hey look! Here come the whales!" he cried excitedly.
n


Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
(2)
"Hey look! Here come the whales!" he cried excitedly.


Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
n
"Hey look! Here come the whales!" he cried excitedly.







The / and ? search commands do not allow you to specify particular occurrences of a *pattern* with numbers. You

cannot, for example, request the third occurrence (after your current position) of a *pattern*.

Scrolling [table 1/3]

CTRL-f	Scroll the screen forward a full window, revealing the window of text below the current window
CTRL-d	Scroll the screen down a half window, revealing lines below the current window
CTRL-b	Scroll the screen back a full window, revealing the window of text above the current window
CTRL-u	Scroll the screen up a half window, revealing the lines of text above the current window

Positioning on a Numbered Line [table 2/3]

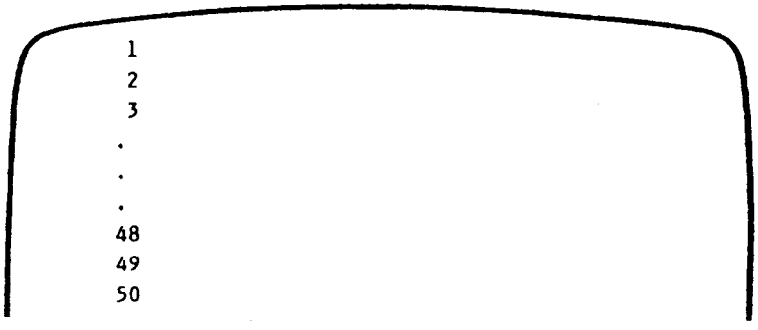
1G	Go ot the first line of the file
G	Go to the last line of the file
CTRL-g	Give the line number and file status

Searching for a Pattern [table 3/3]

/pattern	Search forward in the buffer for the next occurence of the pattern. Position the cursor on the first character of the pattern
?pattern	Search backward in the buffer for the first occurence of the pattern. Position the cursor under the first character of the pattern
n	Repeat the last search command
N	Repeat the search command in the opposite direction

EXERCISE 2

1. Create a file called *exer2*. Type a number on each line, numbering the lines from 1 to 50. Your file should look similar to the following.



2. Try using each of the scroll commands, noticing how many lines scroll through the window. Try the following:

CTRL-f
CTRL-b
CTRL-u
CTRL-d

3. Go to the end of the file. Append the following line of text.

123456789 123456789

What number does the command **7h** place the cursor on? What number does the command **31** place the cursor on?

4. Try the command **$** and the command **0** (number zero).

5. Go to the first character on the line that is not a blank. Move to the first character in the next word. Move back to the first character of the word to the left. Move to the end of the word.

6. Go to the first line of the file. Try the commands that place the cursor in the middle of the window, on the last line of the window, and on the first line of the window.

7. Search for the number 8. Find the next occurrence of the number 8. Find 48.

CREATING TEXT

There are three basic commands for creating text:

a append text

i insert text

o open a new line on which text can be entered

After you finish creating text with any one of these commands, you can return to the command mode of **vi** by pressing the **ESC** key.

Appending Text

There are two commands for appending text to a file:

a append text after the cursor

A append text at the end of the current line

You have already experimented with the **a** command in the section entitled *Creating a File*, above. Make a new file named *junk2*. Append some text using the **a** command. To return to command mode of **vi**, press the **ESC** key. Then compare the **a** command to the **A** command.

Inserting Text

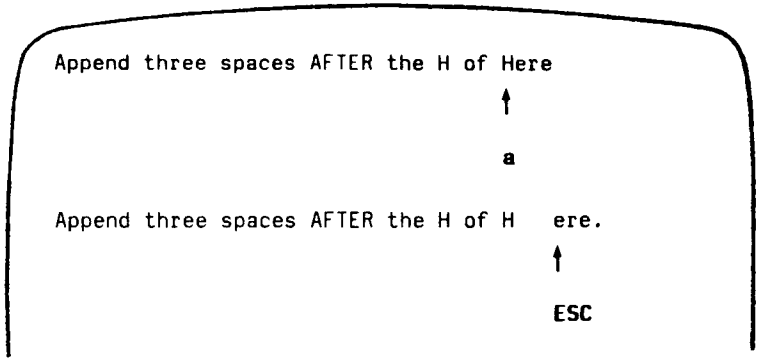
There are two commands for inserting text:

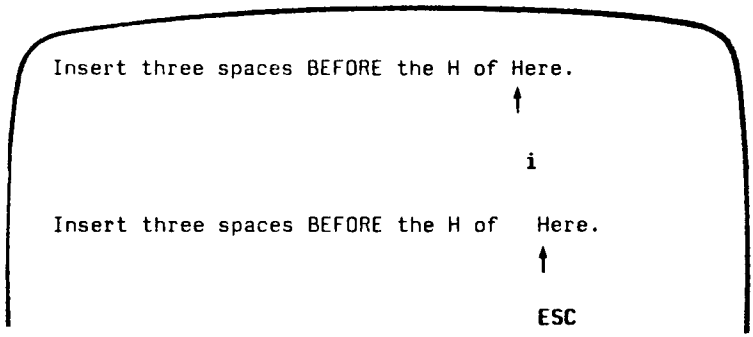
i insert text before the cursor

I insert text at the beginning of the current line before the first character that is not a blank

To return to the command mode of **vi**, press the **ESC** key.

In the following examples you can compare the append and insert commands.





Notice that in both cases, the user has left text input mode by pressing the **ESC** key.

Opening a Line for Text

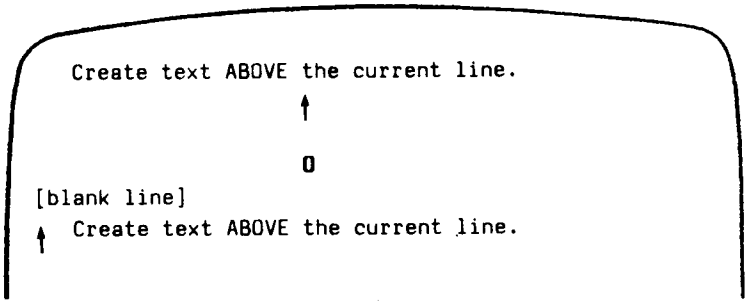
There are two commands for opening a line:

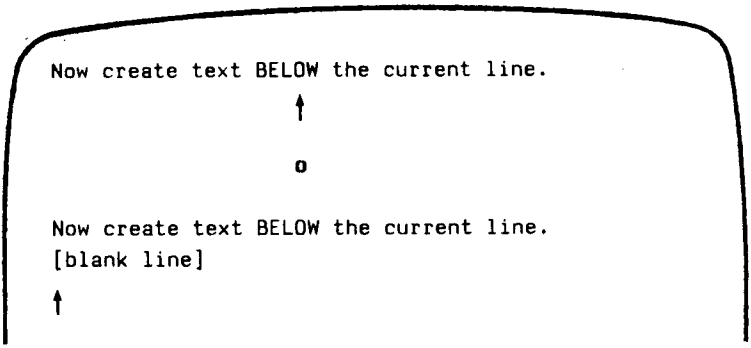
o Create text from the beginning of a new line below the current line. You can issue this command from

any point in the current line.

O Create text from the beginning of a new line above the current line. This command can also be issued from any position in the current line.

The open command creates a directly above or below the current line, and puts you into text input mode. For example, in the following screens the **O** command opens a line above the current line, and the **o** command opens a line below the current line. In both cases, the cursor waits for you to enter text from the beginning of the new line.





The table below summarizes the commands for creating and adding text with the **vi** editor.

Vi Commands for Text Creation

a	Create text after the cursor
A	Create text at the end of the current line
i	Create text in front of the cursor
I	Create text before the first character on the current line that is not a blank
o	Create text at the beginning of a new line below the current line
O	Create text at the beginning of a new line above the current line
ESC	Return **vi** to command mode from any of the above text input modes

EXERCISE 3

1. Create a text file called *exer3*.

2. Insert the following four lines of text.

```
Append text
Insert text
a computer's
job is boring.
```

3. Add the following line of text above the last line:

```
financial statement and
```

4. Using a text insert command, add the following line of text above the third line:

```
Delete text
```

5. Add the following line of text below the current line:

```
byte of the budget
```

6. Using an append command, add the following line of text below the last line:

```
But, it is an exciting machine.
```

7. Move to the first line and add the word some before the word text.

   Now practice using each of the six commands for creating text.

8. Leave **vi** and go on to the next section to find out how to delete any mistakes you made in creating text.

DELETING TEXT

You can delete text with various commands in command mode, and undo the entry of small amounts of text in text input mode. In addition, you can undo entirely the effects of your most recent command.

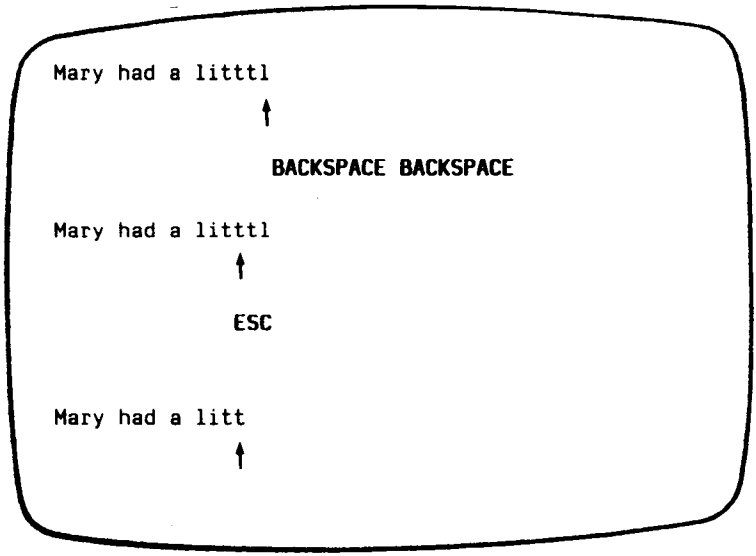
Undoing Entered Text in Text Input Mode

To delete a character at a time when you are in text input mode use the **BACKSPACE** key.

BACKSPACE Delete the current character (the character shown by the cursor).

The **BACKSPACE** key backs up the cursor in text input mode and deletes each character that the cursor backs across. However, the deleted characters are not erased from the screen until you type over them or press the **ESC** key to return to command mode.

In the following example, the arrows represent the cursor.



Notice that the characters are not erased from the screen until you press the **ESC** key.

There are two other keys that delete text in text input mode. Although you may not use them often, you should be aware that they are available. To remove the special meanings of these keys so that they can be typed as text, see the section on special commands.

CTRL-w undo the entry of the current word

@ delete all text entered on current line since text input mode was entered

When you type **CTRL-w**, the cursor backs up over the word last typed and waits on the first character. It does not literally erase the word until you press the **ESC** key or enter new characters over the old ones. The **@** sign behaves in a similar manner except that it removes all text you have typed on the current line since you last entered input mode.

Undo the Last Command

Before you experiment with the delete commands, you should try the **u** command. This command undoes the last command you issued.

u undo the last command

U restore the current line to its state before you changed it

If you delete lines by mistake, type **u**; your lines will reappear on the screen. If you type the wrong command, type **u** and it will be nullified. The **U** command will nullify all changes made to the current line as long as the cursor has not been moved from it.

If you type **u** twice in a row, the second command will undo the first; your undo will be undone! For example, say you delete a line by mistake and restore it by typing **u** Typing **u** a second time will delete the line again. Knowing this command can save you a lot of trouble.

Delete Commands in Command Mode

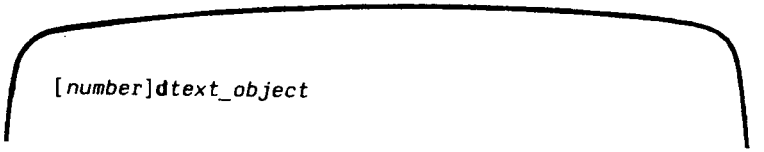
You know that you can precede a command by a number. Many of the commands in **vi**, such as the delete and change commands, also allow you to enter a cursor movement command after another command. The cursor movement command can specify a text object such as a word, line, sentence, or paragraph. The general format of a **vi** command is:

[*number*][*command*]*text_object*

The brackets around some components of the command format show that those components are optional.

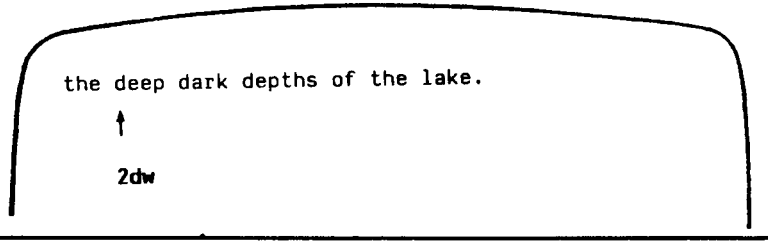
All delete commands issued in command mode immediately remove unwanted text from the screen and redraw the affected part of the screen.

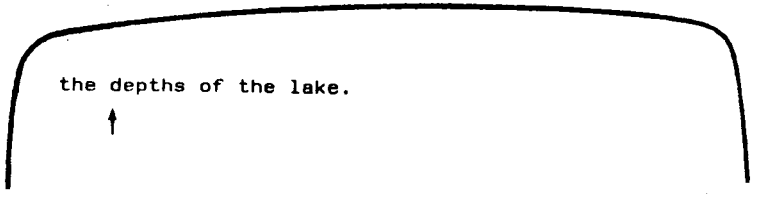
The delete command follows the general format of a **vi** command.



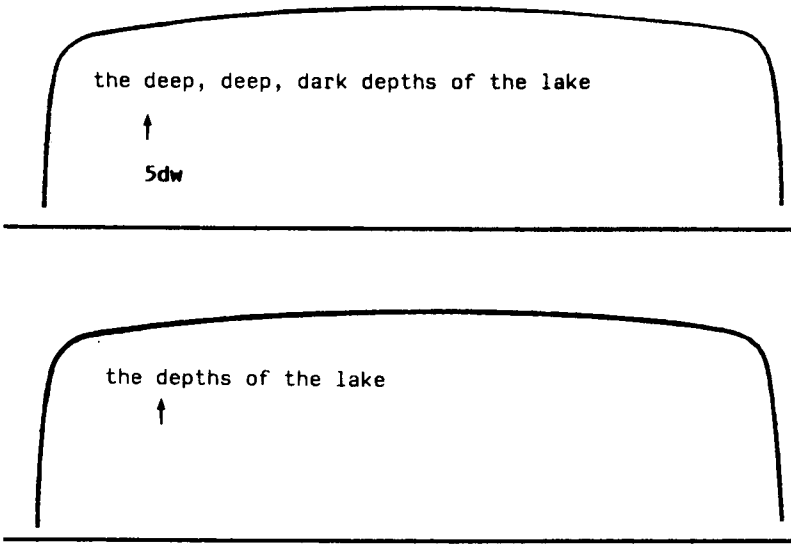
Deleting Words

You can delete a word or part of a word with the **dw** command. Move the cursor to the first character to be deleted and type **dw**. The character under the cursor and all subsequent characters in that word will be erased.





The **dw** command deletes one word or punctuation mark and the space(s) that follow it. You can delete several words or marks at once by specifying a number before the command. For example, to delete three words and two commas, type **5dw**.



Deleting Paragraphs

To delete paragraphs, use the following commands:

d{ and **d}**

Observe what happens to your file. Remember, you can restore the deleted text with **u**.

Deleting Lines

To delete a line, type **dd**. To delete multiple lines, specify a number before the command. For example, typing

10dd

will erase ten lines. If you delete more than a few lines, **vi** will display this notice on the bottom of the screen:

```
10 lines deleted
```

If there are less than ten lines below the current line in the file, a bell will sound and no lines will be deleted.

Deleting Text After the Cursor

To delete all text on a line after the cursor, put the cursor on the first character to be deleted and type

D or **d$**

Neither of these commands allows you to specify a number of lines; they can be used only on the current line.

The **vi** commands for deleting text are sumarized in the table below.

Vi Commands for Text Deletion

INSERT mode:	
BACKSPACE	Delete the current character
CTRL-h	Delete the current character
CTRL-W	Delete the current word
@	Delete the current line of new text or delete

Vi Commands for Text Deletion

	all new text on the current line
COMMAND mode:	
u	Undo the last command
U	Restore current line to its previous state
x	Delete the current character
*n***d***x*	Delete *n* number of text objects of type *x*
dw	Delete the word at the cursor through the next space or to the next punctuation mark
dW	Delete the word and punctuation at the cursor through the next space
dd	Delete the current line
D	Delete the portion of the line to the right of the cursor
d)	Delete the current sentence
d}	Delete the current paragraph

EXERCISE 4

1. Create a file called *exer4* and put the following four lines of text in it:

```
When in the course of human events
there are many repetitive, boring
chores, then one ought to get a
robot to perform those chores.
```

2. Move the cursor to line two and append to the end of that line:

```
tedious and unsavory.
```

Delete the word *unsavory* while you are in append mode.

Delete the word *boring* while you are in command mode.

What is another way you could have deleted the word *boring*?

3. Insert at the beginning of line four:

```
congenial and computerized.
```

Delete the line.

How can you delete the contents of the line without removing the line itself?

Delete all the lines with one command.

4. Leave the screen editor and remove the empty file from your directory.

MODIFYING TEXT

The delete commands and text input commands provide one way for you to modify text. Another way you can change text is by using a command that lets you delete and create text simultaneously. There are three basic change commands: **r**, **s**, and **c**.

Replacing Text

r Replace the current character (the character shown by the cursor). This command does not initiate text input mode, and so does not need to be followed by pressing the **ESC** key.

*n***r** Replace *n* characters with the same letter. This command automatically terminates after the *n*th character is replaced. It does not need to be followed by pressing the **ESC** key.

R Replace only those characters typed over until the **ESC** command is given. If the end of the line is reached, this command will append the input as new text.

The **r** command replaces the current character with the next character that is typed in. For example, suppose you want to change the word acts to ants in the following sentence:

```
The circus has many acts.
```

Place the cursor under the *c* of *acts* and type **r** then the letter *n*. The sentence becomes

```
The circus has many ants.
```

To change *many* to *7777*, place the cursor under the *m* of *many* and type **4r7**.

The **r** command changes the four letters of Imany to four occurrences of the number seven.

```
The circus has 7777 ants.
```

Substituting Text

The substitute command replaces characters, but then allows you to continue to insert text from that point until you press the **ESC** key.

s Delete the character shown by the cursor and append text. End the text input mode by pressing the **ESC** key.

*n***s** Delete *n* characters and append text. End the text input mode by pressing the **ESC** key.

S Replace all the characters in the line.

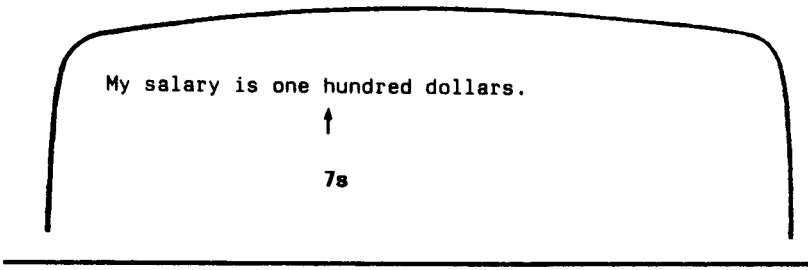
When you enter the **s** command, the last character in the string of characters to be replaced is overwritten by a **$** sign. The characters are not erased from the screen until you type over them, or leave text input mode by pressing the **ESC** key.

Notice that you cannot use an argument with either **r** or **s**. Did you try?

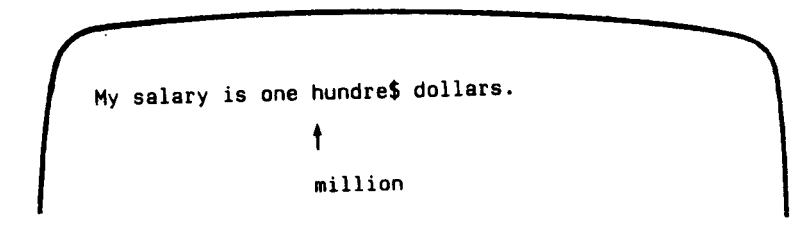
Suppose you want to substitute the word million for the word hundred in the sentence

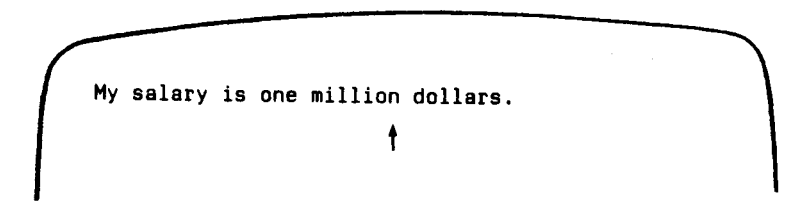
```
My salary is one hundred dollars.
```

Put the cursor under the *h* of *hundred* and type **7s**. Notice where the **$** sign appears.



Then type *million*.





Changing Text

The substitute command replaces characters. The change command replaces text objects, and then continues to append text from that point until you press the **ESC** key. To end the change command, press the **ESC** key.

The change command can take an argument. You can replace a character, word, or an entire line with new text.

*n***c***x* Replace *n* number of text objects of type *x*, such as sentences (shown by **)**) and paragraphs (shown by **}**).

c w Replace a word or the remaining characters in a word with new text. The **vi** editor prints a **$** sign to show the last character to be changed.

*n***cw** Replace *n* words.

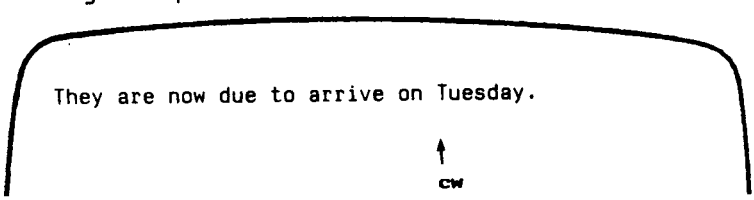
cc Replace all the characters in the line.

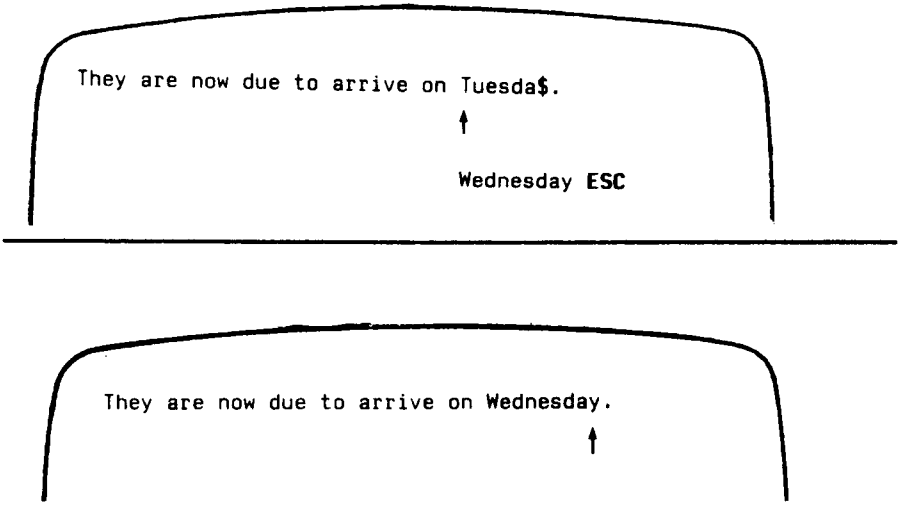
*n***cc** Replace all characters in the current line and up to *n* lines of text.

C Replace the remaining characters in the line, from the cursor to the end of the line.

*n***C** Replace the remaining characters from the cursor in the current line and replace all the lines following the current line up to *n* lines.

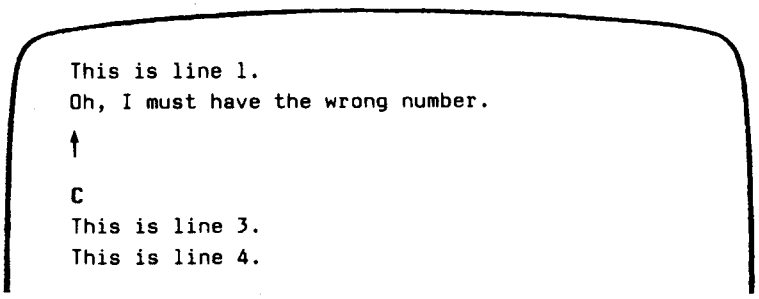
The change commands, **cw** and **C**, use a **$** sign to mark the last letter to be replaced. Notice how this works in the following example:

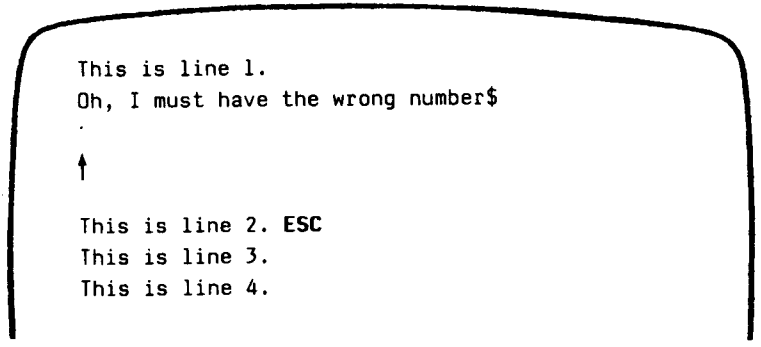


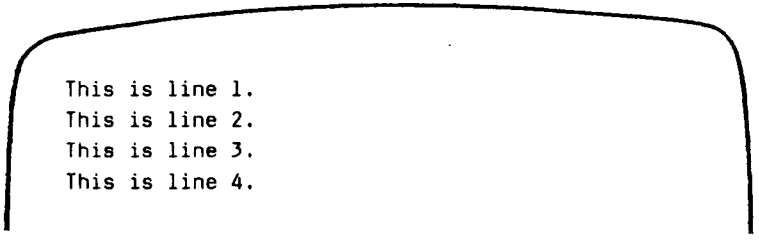


Notice that the new word (*Wednesday*) has more letters than the word it replaced (*Tuesday*). Once you have executed the change command you are in text input mode and can enter as much text as you want. The buffer will accept text until you press the **ESC** key.

The **C** command, when used to change the remaining text on a line, works in the same way. When you enter the command it uses a **$** sign to mark the end of the text that will be deleted, puts you in text input mode, and waits for you to type new text over the old. The following screens offer an example of the **C** command.







Now try combining arguments. For example, type **c{**. Because you know the undo command, do not hesitate to experiment with different arguments or to precede the command with a number. You must press the **ESC** key before using the **u** command, since **c** places you in text input mode.

Compare **S** and **cc**. The two commands should produce the same results.

The **vi** commands for changing text are summarized in the table below..

Vi Text Changing Commands

r	Replace the current character
R	Replace only those characters typed over with new characters until the **ESC** key is pressed
s	Delete the character the cursor is on and append text. End the append mode by pressing the **ESC** key
S	Replace all the characters in the line
cc	Replace all the characters in the line
ncx	Replace *n* number of text objects of type *x*, such as sentences (shown by)) and paragraphs (shown by })

Vi Text Changing Commands

cw	Replace a words or the remaining characters in a word with new text
C	Replace the remaining characters in the line, from the cursor to the end of the line

CUTTING AND PASTING TEXT

Vi provides a set of commands that cut and paste text in a file. Another set of commands copies a portion of text and places it in another section of a file.

Moving Text

You can move text from one place to another in the **vi** buffer by deleting the lines and then placing them at the required point. The last text that was deleted is stored in a temporary buffer. If you move the cursor to that part of the file where you want the deleted lines to be placed and press the **p** key, the deleted lines will be added below the current line.

p Place the contents of the temporary buffer after the cursor.

A partial sentence that was deleted by the **D** command can be placed in the middle of another line. Position the cursor in the space between two words, then press **p**. The partial line is placed after the cursor.

Characters deleted by *n***x** also go into a temporary buffer. Any text object that was just deleted can be placed somewhere else in the text with **p**.

The **p** command should be used right after a delete command since the temporary buffer only stores the results of one command at a time. The **p** command is also used to copy text placed in the temporary buffer by the yank command. The yank command, **y**, is discussed in the section entitled *Copying Text*, below.

Fixing Transposed Letters

A quick way to fix transposed letters is to combine the **x** and the **p** commands as **xp**. **x** deletes the letter. **p** places it after next character.

Notice the error in the next line.

```
A line of tetx
```

This error can be changed quickly by placing the cursor under the *t* in *tx* and then pressing the **x** and **p** keys, in that order. The result is:

```
A line of text
```

Try this. Make a typing error in your file and use the **xp** command to correct it. Why does this command work?

Copying Text

You can yank (copy) one or more lines of text into a temporary buffer, and then put a copy of that text anywhere in the file. To put the text in a new position type **p**; the text will appear on the next line.

The yank command follows the general format of a **vi** command.

[*number*]**y**[*text_object*]

Yanking lines of text does not delete them from their original position in the file. If you want the same text to appear in more than one place, this provides a

convenient way to avoid typing the same text several times. However, if you do not want the same text in multiple places, be sure to delete the original text after you have put the text into its new position.

The yank command is summarized in the table below.

Vi Yank Commands

*n*c*x*	Yank *n* number of text objects of type *x*, such as sentences,), and paragraphs, }
yw	Yank a copy of a word
yy	Yank a copy of the current line
*n***yy**	Yank *n* lines
y)	Yank all text up to the end of a sentence
y}	Yank all text up to the end of the paragraph

Notice that this command allows you to specify the number of text objects to be yanked.

Try the following command lines and see what happens on your screen. (Remember, you can always undo your last command.) Type **5yw**.

Move the cursor to another spot. Type **p**.

Now try yanking a paragraph **y}** and placing it after the current paragraph. Then move to the end of the file **G** and place that same paragraph at the end of the file.

Copying or Moving Text Using Registers

Moving or copying several sections of text to a different part of the file is tedious work. **Vi** provides a shortcut for this: named registers in which you can store text until you want to move it. To store text you can either yank or delete the text you wish to store.

Using registers is useful if a piece of text must appear in many places in the file. The extracted text stays in the specified register until you either end the editing session, or yank or delete another section of text to that register.

The general format of the command is:

[*number*][*"x*]*command*[*text_object*]

The *x* is the name of the register and can be any single letter. It must be preceded by a double quotation mark. For example, place the cursor at the beginning of a line. Type **3"ayy**.

Type in more text and then go to the end of the file. Type **"ap**.

Did the lines you saved in register **a** appear at the end of the file?

The cut and paste commands are summarized in the table below.

Vi Cutting and Pasting Commands

p	Place the contents of the temporary buffer containing the text obtained from the most recent delete or yank command into the text after the cursor

yy	Yank a line of text and place it in a temporary buffer
nyx	Yank a copy of *n* number of text objects of type *x*, and place them in a temporary buffer
"*xyn*	Place a copy of a text object of type *n* in the register named by the letter *x*
"*xp*	Place the contents of register *x* after the cursor

EXERCISE 5

1. Enter **vi** with the file called *exer2*. that you created in Exercise 2.

   Go to line eight and change its contents to *END OF FILE*

2. Yank the first eight lines of the file and place them in register **z**. Put the contents of register **z** after the last line of the file.

3. Go to line eight and change its contents to *eight is great*

4. Go to the last line of the file. Substitute *EXERCISE* for *FILE* Replace *OF* with *TO*

SPECIAL COMMANDS

Here are some special commands that you will find useful.

. repeat the last command

J join two lines together

CTRL-lclear the screen and redraw it

~ change lower case to upper case and vice versa

Repeating the Last Command

The . command repeats the last command to create, delete, or change text in the file. It is often used with the search command.

For example, suppose you forget to capitalize the *S* in United *S*tates. However, you do not want to capitalize the *s* in chemical states. One way to correct this problem is by searching for the word *states*. The first time you find it in the expression *United States*, you can change the *s* to *S*. Then continue your search. When you find another occurrence, you can simply type a period; **vi** will remember your last command and repeat the substitution of *s* for *S*.

Experiment with this command. For example, if you try to add a period at the end of a sentence while in command mode, the last text change will suddenly appear on the screen. Watch the screen to see how the text is affected.

Joining Two Lines

The J command joins lines. To enter this command, place the cursor on the current line, and press the **SHIFT** and **j** keys simultaneously. The current line is joined with the following line.

For example, suppose you have the following two lines of text:

```
Dear Mr.
Smith:
```

To join these two lines into one, place the cursor under any character in the first line and type **J**. You will immediately see the following on your screen:

```
Dear Mr. Smith:
```

Notice that **vi** automatically places a space between the last word on the first line and the first word on the second line.

Clearing and Redrawing the Window

If another X/OS system user sends you a message using the write command while you are editing with **vi**, the message will appear in your current window, over part of the text you are editing. To restore your text after you have read the message, you must be in command mode. (If you are in text input mode, press the **ESC** key to return to command mode.) Then type **CTRL-l**. **Vi** will erase the message and redraw the window exactly as it appeared before the message arrived.

Changing Lower Case to Upper Case and Vice Versa

A quick way to change any lower case letter to upper case, or vice versa, is by putting the cursor on the letter to be changed and typing a ~ (tilde). For example, to change the letter a to A, press ~. You can change several letters by typing ~ several times, but you cannot precede the command with a number to change several letters with one command.

The special commands are summarized in the table below.

Vi Special Commands

.	Repeat the last command
J	Join the line below the current line with the current line
CTRL-l	Clear and redraw the current window
~	Change lower case to upper case and vice versa

USING LINE EDITING COMMANDS IN VI

The **vi** editor has access to many of the commands provided by a line editor called **ex**. (For a complete list of **ex** commands see the *ex*(1) entry in the *Utilities Reference Manual*.) This section discusses some of those most commonly used.

If you are familiar with **ed**, you may want to experiment on a test file to see how many **ed** commands also work in **vi**.

Line editor commands begin with a : (colon). After the colon is typed, the cursor will drop to the bottom of the screen and display the colon. The remainder of the command will also appear at the bottom of the screen as you type it.

Temporarily Returning to the Shell

When you enter **vi**, the contents of the buffer fill your screen, making it impossible to issue any shell commands. However, you may want to do so. For example, you may want to get information from another file to incorporate into your current text. You could get that information by running one of the shell commands that display the text of a file on your screen, such as the **cat** or **pg**

commands. However, quitting and reentering the editor is time consuming and tedious. **Vi** offers two methods of escaping the editor temporarily so that you can issue shell commands (and even edit other files) without having to write your buffer and quit: the **:!** command and the **:sh** command.

The **:!** command allows you to escape the editor and run a shell command on a single command line. From the command mode of **vi**, type **:!**. These characters will be printed at the bottom of your screen. Type a shell command immediately after the **!**. The shell will run your command, give you output, and print the message *[Hit return to continue]*. When you press the **CR** key **vi** will refresh the screen and the cursor will reappear exactly where you left it.

The **ex** command **:sh** allows you to do the same thing, but behaves differently on the screen. From the command mode of **vi** type **:sh** and press the **CR** key. A shell command prompt will appear on the next line. Type your command(s) after the prompt as you would normally do while working in the shell. When you are ready to return to **vi**, type **CTRL-d** or **exit**; your screen will be refreshed with your buffer contents and the cursor will appear where you left it.

Even changing directories while you are temporarily in the shell will not prevent you from returning to the **vi** buffer where you were editing your file when you type **exit** or **CTRL-d**.

Writing Text to a New File

The **:w** (for write) command allows you to create a file by copying lines of text from the file your are currently editing into a file that you specify. To create your new file you must specify a line or range of lines (with their line numbers), along with the name of the new file, on the command line. You can write as many lines as you

like. The general format is:

:*line_number*[,*line_number*]**w** *filename*

For example, to write the third line of the buffer to a line named *three*, type:

:3w three

Vi reports the successful creation of your new file with the following information:

"three" [New file] 1 line, 20 characters

To write your current line to a file, you can use a . (period) as the line address:

:.w junk

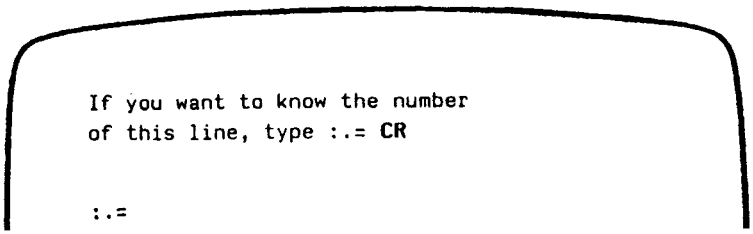
A new file called *junk* will be created. It will contain only the current line in the **vi** buffer.

You can also write a whole section of the buffer to a new file by specifying a range of lines. For example, to write lines 23 through 37 to a file, type the following:

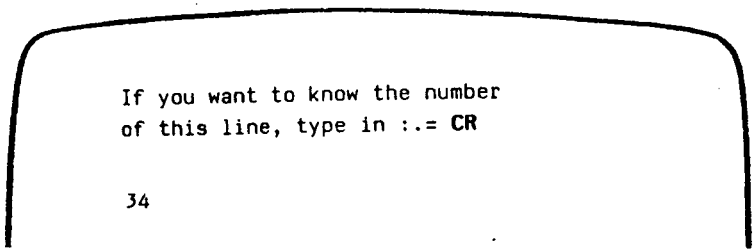
:23,37w *newfile*

Finding the Line Number

To determine the line number of a line, move the cursor to it and type : (colon). The colon will appear at the bottom of the screen. Type .= after it and press the **CR** key.



As soon as you press the **CR** key, your command line will disappear from the bottom line and be replaced by the number of your current line in the buffer.



You can move the cursor to any line in the buffer by typing : and the line number. The command line

:*n* **CR**

means to go to the *n*th line of the buffer.

Deleting the Rest of the Buffer

One of the easiest ways to delete all the lines between the current line and the end of the buffer is by using the line editor command **d** with the special symbols for the current and last lines.

```
:.,$d
```

The . represents the current line; the **$** sign, the last line.

Adding a File to the Buffer

To add text from a file below a specific line in the editing buffer, use the **:r** (read) command. For example, to put the contents of a file called *data* into your current file, place the cursor on the line above the place where you want it to appear. Type

```
:r data
```

You may also specify the line number instead of moving the cursor. For example, to insert the file *data* below line 56 of the buffer, type

```
:56r data
```

Do not be afraid to experiment; you can use the **u** command to undo **ex** commands, too.

Making Global Changes

One of the most powerful commands in **ex** is the global command. The global command is given here to help those users who are familiar with the line editor. Even if you are not familiar with a line editor, you may want to try the command on a test file.

For example, say you have several pages of text about the DNA molecule in which you refer to its structure as a helix. Now you want to change every occurrence of the word helix to double helix. The **ex** editor's global command allows you to do this with one command line. First, you need to understand a series of commands.

:g/*pattern/command* For each line containing *pattern*, execute the **ex** command named *command*. For example, type: **:g/helix CR**. The line editor will print all lines that contain the pattern *helix*.

:s/*pattern/new_words*

This is the substitute command. The line editor searches for the first instance of the characters *pattern* on the current line and changes them to *new_words*.

:s/*pattern/new_words***/g**

If you add the letter **g** after the last delimiter of this command line, **ex** will change every occurrence of *pattern* on the current line. If you do not, **ex** will change only the first occurrence.

:g/*helix***/s//***double helix***/g**

This command line searches for the word *helix*. Each time *helix* is found, the substitute command substitutes two words, *double helix*, for every instance of *helix* on that line. The delimiters after the **s** do not need to have *helix* typed in again. The command remembers the word from the delimiters after the global command **g**. This is a powerful command.

The line editor commands available in **vi** are summarized in the table below.

Line Editor Commands Available to vi	
:	Shows that the commands that follow are line line editor commands
:sh	Temporarily returns you to the shell to perform shell commands
CTRL-d	Escapes the temporary shell and returns you to the current window of **vi** to continue editing
:n	Goes to the *n*th line of the buffer
:x,y **w** data	Writes lines from number *x* through number *y* into a new file called *data*
:$	Goes to the last line of the buffer
:,$d	Deletes all the lines in the buffer from the current line to the end
:r *file*	Inserts the contents of *file* after the current line of the buffer

QUITTING VI

There are five basic command sequences to quit the **vi** editor. Commands that are preceded by a colon are line editor commands.

ZZ or **:wq** — Writes the contents of the **vi** buffer to the X/OS file currently being edited and quit **vi**.

:w *filename* with

:q — Write the temporary buffer to a new file named *filename* and quit **vi**.

:w! *filename* with

:q — Overwrite an existing file called *filename* with the contents of the buffer and quit **vi**.

:q! Quit **vi** without writing the buffer to a file, and discard all changes made to the buffer.

:q Quit **vi** without writing the buffer to a X/OS file. This works only if you have made no changes to the buffer; otherwise **vi** will warn you that you must either save the buffer or use the **:q!** command to terminate.

The **ZZ** command and **:wq** command sequence both write the contents of the buffer to a file, quit **vi**, and return you to the shell. You have tried the **ZZ** command. Now try to exit **vi** with **:wq**. **Vi** remembers the name of the file currently being edited, so you do not have to specify it when you want to write the buffer's contents back into the file. Type **:wq**. The system responds in the same way it does for the **ZZ** command. It tells you the name of the file, and reports the number of lines and characters in the file.

What must you do to give the file a different name? For example, suppose you want to write to a new file called *junk*. Type **:w junk** After you write to the new file, leave **vi**. Type **:q**.

If you try to write to an existing file, you will receive a warning. For example, if you try to write to a file called *johnson*, the system will respond with:

```
"johnson" File exists - use "w! johnson" to overwrite
```

If you want to replace the contents of the existing file with the contents of the buffer, use the **:w!** command to overwrite *johnson*.

```
:w! johnson
```

Your new file will overwrite the existing one.

If you edit a file called *memo*, make some changes to it, and then decide you don't want to keep the changes, or if you accidentally press a key that gives **vi** a command you cannot undo, leave **vi** without writing to the file. Type **:q!**.

The quit commands are summarized in the table below..

Vi Quit Commands

ZZ	Write the file and quit **vi**
:wq	Write the file and quit **vi**
:w *file*	Write the editing buffer to a new file called *file* and quit **vi**
:w! *file*	Overwrite an existing file called *file* with the contents of the editing buffer and quit **vi**
:q!	Quit **vi** without writing the buffer to a file
:q	Quit **vi** without writing the buffer to a file

SPECIAL OPTIONS FOR VI

The **vi** command has some special options. It allows you to:

- recover a file lost by an interrupt to the X/OS system
- place several files in the editing buffer and edit each in sequence, and

- view the file at your own pace by using the **vi** cursor positioning commands

Recovering a File Lost by an Interrupt

If there is an interrupt or disconnect, the system will exit the **vi** command without writing the text in the buffer back to its file. However, the X/OS system will store a copy of the buffer for you. When you log back in to the X/OS system you will be able to restore the file with the **-r** option for the **vi** command. Type

vi -r *filename*

The changes you made to *filename* before the interrupt occurred are now in the **vi** buffer. You can continue editing the file, or you can write the file and quit **vi**. The **vi** editor will remember the file name and write to that file.

Editing Multiple Files

If you want to edit more than one file in the same editing session, issue the **vi** command, specifying each file name. Type

vi *file1 file2*

Vi responds by telling you how many files you are going to edit. For example:

```
2 files to edit
```

After you have edited the first file, write your changes (in the buffer) to the file (*file1*). Type **:w**.

The system response to the **:w** command will be a message at the bottom of the screen giving the name of the file, and the number of lines and characters in that file. Then you can bring the next file into the editing buffer by using the **:n** command. Type **:n**.

The system responds by printing a notice at the bottom of the screen, telling you the name of the next file to be edited and the number of characters and lines in that file.

Select two of the files in your current directory. Then enter **vi** and place the two files in the editing buffer at the same time. Notice the system responses to your commands at the bottom of the screen.

Viewing a File

It is often convenient to be able to inspect a file by using **vi**'s powerful search and scroll capabilities. However, you might want to protect yourself against accidentally changing a file during an editing session. The read-only option prevents you from writing in a file. To avoid accidental changes, you can set this option by invoking the editor as **view** rather than **vi**.

The special options for **vi** are summarized in the table below.

Vi Special Options

vi *file1 file2 file3*	Enter three files into the vi for for editing
:w	Write the current file and
:n	call the next file into the buffer
vi -r *file*	Restore the changes made to file

EXERCISE 6

1. Try to restore a file lost by an interrupt.

   Enter **vi**, create some text in a file called *exer6*. Turn off your terminal without writing to a file or leaving **vi**. Turn your terminal back on, and log in again. Then try to get back into **vi** and edit *exer6*.

2. Place *exer1* and *exer2* in the **vi** buffer to be edited. Write *exer1* and call in the next file in the buffer, *exer2*.

   Write *exer2* to a file called *junk*.

   Quit **vi**.

3. Try out the command:

   vi exer*

What happens? Try to quit all the files as quickly as possible.

4. Look at *exer4* in read-only mode.

   Scroll forward.

   Scroll down.

   Scroll backward.

   Scroll up.

   Quit and return to the shell.

ANSWERS TO EXERCISES

There is often more than one way to perform a task in **vi**. Any method that works is correct. The following are suggested ways of doing the exercises.

EXERCISE 1

1-1. Ask your system administrator for your terminal's system name. Type:

```
TERM=terminal_name
```

where *terminal_name* is the name of the system.

1-2. Enter the **vi** command for a file called *exer1*:

```
vi exer1
```

Then use the append command, **a**, to enter the following text in your file:

```
This is an exercise! CR
Up, down CR
left, right, CR
build your terminal's CR
muscles bit by bit ESC
```

1-3. Use the **k** and **h** commands.

1-4. Use the **x** command.

1-5. Use the **j** and **l** commands.

1-6. Enter **vi** and use the append command, **a**, to enter the following text:

```
and byte by byte ESC
```

Then use **j** and **l** to move to the last line and character of the file. Use the **a** command again to add text. You can create a new line by pressing the **CR** key. To leave text input mode, press the **ESC** key.

1-7. Type **ZZ**.

1-8. Type

```
vi exerl CR
```

Notice the system response:

```
"exerl" 7 lines, 102 characters
```

EXERCISE 2

2-1. Type:

```
vi exer2 CR
al CR
2 CR
3 CR
.
.
.
48 CR
```

```
49 CR
50 ESC
```

2-2. Type:

```
CTRL-f
CTRL-b
CTRL-u
CTRL-d
```

Notice the line numbers as the screen changes.

2-3. Type:

```
G
o
123456789 123456789 ESC
7h
31
```

Typing **7h** puts the cursor on the *2* in the second set of numbers. Typing **31** puts the cursor on the *5* in the second set of numbers.

$ = end of line

0 = first character in the line

2-5. Type:

```
^
w
b
```

e

2-6. Type:

1G
M
L
H

2-7. Type:

/8
n
/48

EXERCISE 3

3-1. Type

vi exer3

3-2. Type:

aAppend text **CR**
Insert text **CR**
a computer's **CR**
job is boring. **ESC**

3-3. Type:

o
financial statement and **ESC**

3-4. Type:

3G
i Delete text **CR ESC**

The text in your file now reads:

Append text
Insert text
Delete text
a computer's
financial statement and
job is boring.

3-5. The current line is

a computer's.

To create a line of text below that line use the **o** command.

3-6. The current line is

byte of the budget

G puts you on the bottom line.

A lets you begin appending at the end of the line.

CR creates the new line.

Add the sentence:

```
But, it is an exciting machine.
```

ESC leaves append mode.

3-7. Type:

```
1G
/text
i some space bar ESC
```

3-8. ZZ will write the buffer to *exer3* and return you to the shell.

EXERCISE 4

4-1. Type:

```
vi exer4 CR
aWhen in the course of human events CR
there are many repetitive, boring CR
chores, then one ought to get a CR
robot to perform those chores. ESC
```

4-2. Type:

```
2G
A tedious and unsavory 8 BACKSPACE CR
ESC
```

Press **h** until you get to the *b* of *boring*. Then type **dw**. (You can also use **6x**.)

4-3. You are at the second line. Type:

2j
I congenial and computerized **ESC**
dd

To delete the line and leave it blank, type in:

0 (zero moves the cursor to the beginning of the line)
D

H
3dd

4-4. Write and quit **vi**.

ZZ

Remove the file.

rm exer4 **CR**

EXERCISE 5

5-1. Type:

```
vi exer2 CR
8G
cc END OF FILE ESC
```

5-2. Type:

```
1G
8"zyy
G
"zp
```

5-3. Type:

```
8G R
cc 8 is great ESC
```

5-4. Type:

```
G
2w
cw
EXERCISE ESC
2b
cw
TO ESC
```

EXERCISE 6

6-1. Type:

```
vi exer6 CR
a (append several lines of text)
ESC
```

Turn off the terminal.

Turn on the terminal.
Log in on your X/OS system. Type:

```
vi -r exer6 CR
:wq CR
```

6-2. Type:

```
vi exer1 exer2 CR
:w CR
:n CR

:w junk CR
ZZ
```

6-3. Type:

```
vi exer* CR
```

The response is:

```
8 files to edit
```

Note that **vi** calls all files with names that begin with *exer*.

ZZ
ZZ

6-4. Type:

view exer4 **CR**
CTRL-f
CTRL-d
CTRL-b
CTRL-u
:q **CR**

INTRODUCTION

This sixth chapter contains tutorial coverage of four commands used to save disk space by storing files in a compressed form. One is an archiving system, while the other three comprise the file packing system. The utilities covered are as follows:

AR the archive and library utility

PACK compresses files

PCAT concatenates and prints packed files

UNPACK expands previously packed files

These tutorials are arranged in alphabetical order.

AR: archive and library utility

INTRODUCTION

This section is a brief tutorial introduction to the X/OS **ar** archive and library maintenance system for portable archives. This utility is used to maintain archive files consisting of more than one original file. Its main function is the creation and updating of library files used by the **ld** link editor, but it can be used for any similar purpose. For details of **ld**, see the *System Interfaces and Libraries Reference Manual.* Archives created using **ar** are supplied with headers that can be interpreted by any X/OS-compatible system. These headers are inserted at the beginning of each archive, and are described in the *ar*(4) and *a.out*(4) entries of the *System Interfaces and Libraries Reference Manual.*

SYNTAX

```
ar [-]key [posname] file name ...
```

DESCRIPTION

The arguments to **ar** are as follows:

[-]*key* This option takes the form of an optional - character, followed by one or more of the following:

d deletes the file or files identified by the *name* parameter(s).

r replaces the file or files identified by the *name* parameter(s), using the following

options:

u replaces only those files whose date of last modification is later than the date held in the archive's *ar_date* variable (see above).

a places new files after the location pointed to by *posname*.

b or **i** places new files before the location pointed to by *posname*.

Note that in both these cases, *posname* must be specified in the **ar** command line for the option to be meaningful. If *posname* is not specified, new files will be added to the end of the archive.

q quickly appends the names files to the end of the archive. Note that the positioning options listed above under option **r** will not work if this option is used. Note also that **ar** does not check whether the named files are already in the archive.

t prints a list of the files contained in the archive. If *name* is specified, only the file or files identified will appear in the table of contents. If *name* is not specified, all files will be listed.

p prints the contents of the file or files listed in *name*.

m moves the file or files identified in *name* to the end of the archive. Note that the positioning options under **r** can be used to insert files into a specific location.

x extracts the file or files identified by *name*. Note that if no *name* is specified, all files in the archive are extracted. Note also that this option does not affect the archive itself.

v **ar** gives a verbose file-by-file description of the process when a new archive is being made from an old archive. When used with the **t** option, a long list of information about each file is given. When used with the **x** option, each file is preceded by its name.

c creates the file identified by the parameter *file*. This file is normally created by **ar** when necessary. The message usually associated with the creation of *file* does not appear when this option is used.

l places temporary files in the local directory. If this option is not used, the temporary files are placed in the default directory, */tmp*.

s re-writes the archive symbol table, even when **ar** was called with an option having no effect on the contents of the archive. This option may be called in order to restore the symbol table after the **strip** command has been used.

Note that the options **v**, **u**, **a**, **i**, **b**, **c**, **l** or **s** must be used with one or more of the options **d**, **r**, **q**, **t**, **p**, **m** or **x**.

posname This option is used to indicate the desired location of a file within the archive. Files may be moved to before or after *posname*. Note that *posname* is the name of a file in the

archive. This option must be used when specifying the positioning options under the **r** option of the *key* parameter.

file This option specifies the name of a file in the archive.

name This option specifies a file in the archive. More than one *name* parameter may be given. Note that if the same file is specified twice within a string of *name* parameters, it may be archived twice.

EXAMPLES

The following examples comprise both command line input, and system response output. The **ls** command has been used to list the files to be archived. For details of this utility, see the **ls** tutorial in this manual. In the first example, **ar** has been used with the **q** option to illustrate how an archive called *testarc* is created. Remember that the **>** character in bold is used to represent the system prompt, and that **CR** is used to specify when the carriage return key is to be pressed in order to enter the command.

```
>ls CR
cats
dogs
monkeys
fish
birds
people

>ar q testarc cats dogs monkeys fish birds people CR
ar: creating testarc

>
```

The next example illustrates how a table of contents can be produced, using the **t** option that lists the files contained in archive *testarc*.

```
>ar t testarc CR
cats
dogs
monkeys
fish
birds
people

>
```

The third example shows how the **p** option can be used to list the contents of file *fish* within archive *testarc*.

```
>ar p testarc fish CR
cod
halibut
basking shark
lamprey
haddock

>
```

The final example illustrates combined use of the **t** and **v** options, to produce a more detailed list of the files in archive *testarc*:

```
>ar -tv testarc CR
rwxr-xr-x 225////33   110 May 29 17:54 1987 cats
rwxr-xr-x 225////33    85 May 26 10:23 1987 dogs
rwxr-xr-x 225////33    55 May 26 12:06 1987 monkeys
rwxr-xr-x 225////33   105 May 27 15:44 1987 fish
rwxr-xr-x 225////33    60 May 24 09:17 1987 birds
rwxr-xr-x 225////33   310 May 24 14:32 1987 people

>
```

PACK: compress files

INTRODUCTION

This is a tutorial introduction to the **pack** command which is used to compress and store specified files. The original file is replaced by a file with the same name and the extension *.z*, but retains the same access modes, access and modification dates, and owner. Text files, for example, may be compressed to 75% of their original size. Packed files can be unpacked later using the **unpack** or **pcat** commands. This is performed by reference to a de-coding tree placed at the beginning of the file by **pack**. This is constructed by way of a number of passes through the file, each of which compresses its contents a bit more. These passes are progressively less effective, that is, the overall effectiveness is asymptotic.

Pack uses a coding system based on character frequencies. Each character is represented by its byte code. The frequency distribution of characters determines the degree of compression that can occur. Files with a limited character set, for example, text files using only the normal keyboard characters, tend to achieve more compression than files with a large character set. Also, character frequencies that tend towards a skewed distribution are more easily compressed than files tending towards a normal distribution.

Also affecting compression is file size. Smaller files, especially those under three blocks in size, tend to retain their original size, partly because the de-coding tree added to the file during compression tends to override any reduction in size. However, where the character frequency distribution is very skewed, for example printer plots or pictures, some compression can be achieved despite this problem.

File packing will not occur if any of the following are true:

1. the file appears to be compressed already
2. the filename is longer than twelve characters
3. the file has links to another file
4. the file is a directory
5. the file cannot be opened
6. no disk storage blocks will be saved by packing
7. a file of the same name with the extension *.z* already exists
8. the *.z* file cannot be created
9. an input-ouput error occured during processing

The number of files that could not be compressed is printed after execution of the **pack** command line. Note that directories cannot be compressed.

SYNTAX

```
pack [-] name [name...]
```

DESCRIPTION

The arguments and options to **pack** are as follows:

- sets a flag to cause output of the number of occurences of each character, its relative frequency, and its byte code. This option can be set and reset as often as necessary within a string of filenames.

name the name of the file to be compressed. The compressed file is created with the filename *name.z*.

EXAMPLES

This example begins by using the **ls -l** command to display the files held by the current directory. Details of this command can be found in the **ls** tutorial in this manual. **Pack** is then used to compress the two files. Finally, **ls -l** is used again to illustrate the effect of compression on the files. Note that the filenames, access codes, date and owner are unchanged, but the size is reduced. The - option is used to display the compression statistics.

Remember that the > characters in bold represents the system prompt, and that **CR** indicates that the carriage return key should be compressed in order to enter the command line.

```
>ls -l CR
total xxx
-rwx-r--r-  1 spike   25649  July 21 16:07  textfile1
-rwx-r--r-  1 spike   18003  July 21 09:21  textfile2
```

```
>pack - textfile1 textfile1 CR
pack: textfile1: 27.8% Compression
      from 25649 to 18518 bytes
      Huffman tree has 15 levels below root
      88 distinct bytes in input
      dictionary overhead = xxx bytes
      effective entropy = x.xx bits/byte
      asymptotic entropy = x.xx bits/byte
pack: textfile2: 27.6% Compression
      from 18003 to 13034 bytes
      Huffman tree has 15 levels below root
      91 distinct bytes in input
      dictionary overhead = xxx bytes
      effective entropy = x.xx bits/byte
      asymptotic entropy = x.xx bits/byte

>ls -l CR
total xxx
-rwx-r--r-  1 spike   18518  July 21 16:07  textfile1.z
-rwx-r--r-  1 spike   13034  July 21 09:21  textfile2.z

>
```

The statistics state that *textfile1* underwent a compression of 27.8%, involving a reduction of 7131 bytes. The line mentioning the Huffman tree implies that **pack** made fifteen passes through the file. A total of 88 different characters were discovered. The entries concerning entropy give an indication of the effective compression in terms of bits per byte.

PCAT: concatenate and print packed files

INTRODUCTION

This is a tutorial introduction to the **pcat** utility which concatenates and prints files that have already been compressed using the **pack** utility. The packed file is de-coded, and its uncompressed form is printed. **Pcat** will not operate if any of the following are true:

1. the filename (excluding the *.z* extension added by **pack**) is longer than 12 characters.

2. the file cannot be opened.

3. the file does not appear to have been created using **pack**.

In the event of **pcat** failing to process one or more files, the number of files involved is returned.

SYNTAX

---

pcat file [file... [file]]

DESCRIPTION

The arguments to **pcat** takes the following form:

file identifies the file to be printed. One or more may be entered. Note that it is not necessary to specify the *.z* extension.

The output of **pcat** can be re-directed into another file using the re-directrion indicator. The new file will

contain the original form of the file.

EXAMPLES

The first example begins by using the **ls -l** command to list the files already packed. It continues by directing the output from **pcat** into an expanded file, one for each *.z* file listed by **ls** . Finally, **ls -l** is used again to list the newly created files. Remember that the **>** symbol in bold represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>ls -l CR
total 65
-rwx-r--r-  1 spike  18518  July 21 16:07  textfile1.z
-rwx-r--r-  1 spike  13034  July 21 09:21  textfile2.z

>pcat textfile1 > expand1 CR

>pcat textfile2 > expand2 CR

>ls -l CR
-rwx-r--r-  1 spike  18518  July 21 16:07  textfile1.z
-rwx-r--r-  1 spike  13034  July 21 09:21  textfile2.z
-rwx-r--r-  1 spike  25649  July 21 16:07  expand1
-rwx-r--r-  1 spike  18003  July 21 09:21  expand2

>
```

UNPACK: expands files

INTRODUCTION

This is a tutorial introduction to the **unpack** utility, which expands files that have been run through **pack**. Compressed files are returned to their original state. The compressed file is deleted, and the expanded file is given it's pre-compression filename, that is, the *.z* extension assigned by **pack** is dropped. For example, a text file called *testfile* would be packed into a file called *testfile.z*. After being run through **unpack**, it would be returned to its uncompressed form, with the name *testfile*.

Note that **unpack** will not work if any of the following are true:

1. the filename (excluding the *.z* extension added by **pack**) is longer than 12 characters.

2. the file cannot be opened.

3. the file does not appear to have been created using **pack**.

4. a file with the name to be assigned to the unpacked file already exists.

5. the unpacked file cannot be created.

In the event of **unpack** failing to process one or more files, the number of files involved is returned.

SYNTAX

```
unpack file [file ...]
```

DESCRIPTION

file identifies the file to be printed. One or more may be entered. Note that it is not necessary to specify the *.z* extension.

EXAMPLES

The first example begins by using the **ls -l** command to list the files already packed. It continues by using **unpack** on the two *.z* file listed by **ls**. Finally, **ls -l** is used again to list the newly created expanded files. Remember that the **>** symbol in bold represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>ls -l CR
total 65
-rwx-r--r-  1 spike  GRP2 18518  July 21 16:07  textfile1.z
-rwx-r--r-  1 spike  GRP2 13034  July 21 09:21  textfile2.z

>unpack textfile1 textfile2 CR
unpack: textfile1: unpacked
unpack: textfile2: unpacked

>ls -l CR
-rwx-r--r-  1 spike  GRP2 25649  July 21 16:07  textfile1
-rwx-r--r-  1 spike  GRP2 18003  July 21 09:21  textfile2

>
```

INTRODUCTION

This chapter consists of three tutorials, covering the most commonly used X/OS utilities for editing or examining files according to user-specified criteria. The utilities covered are as follows:

AWK the pattern scanning and processsing utility

GREP one of the family of pattern searching utilities

SED the stream editor

The tutorials are presented in alphabetical order.

AWK: pattern scanning and processing

INTRODUCTION

This section is a tutorial style introduction to the **awk** utility, a file-processing programming language designed to make many common information and retrieval text manipulation tasks easy to state and perform. **Awk** will:

- generate reports
- match patterns
- validate data
- filter data for transmission

The first part of this chapter gives a general statement of the **awk** syntax. The section entitled *Description* covers the various options and arguments to the system. The *Examples* section provides a number of examples that show the syntax rules in use.

SYNTAX

```
awk [[-f]prog] [parameters] [files]
```

DESCRIPTION

The arguments and options to **awk** are as follows:

[-**f**]*prog* specifies the pattern or patterns to be searched for. The *prog* argument can be either a literal string enclosed in single quotes, giving a single serach pattern, or it can be a filename. If it is a filename, the **-f** identifier must be used, and the file identified must contain one or more search patterns.

parameters identifies and assigns values to parameters internal to an **awk** operation. Parameters take the form of *x=y*, where *x* identifies the parameter, and *y* is its assigned value.

files identifies the file or files to be processed.

An **awk** program takes the form of a sequence of statements of the form

```
pattern {action}
pattern {action}
...
```

Awk runs on a set of input files. The basic operation of **awk** is to scan a set of input lines, in order, one at a time. In each line, **awk** searches for the pattern described in the **awk** program. If that pattern is found in the input line, a corresponding action is performed. In this way, each statement of the **awk** program is executed for a given input line. When all the patterns are tested, the next input line is fetched; and the **awk** program is once again executed from the beginning.

In the **awk** command, either the pattern or the action may be omitted, but not both. If there is no action for a

pattern, the matching line is simply printed. If there is no pattern for an action, then the action is performed for every input line. The null **awk** program does nothing. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

For example, this **awk** program

```
/x/     {print}
```

prints every input line that has an *x* in it.

An **awk** program has the following structure:

- a <BEGIN> section
- a <record> or main section
- an <END> section

The <BEGIN> section is run before any input lines are read, and the <END> section is run after all the data files are processed. The <record> section is run over and over for each separate line of input. The words <BEGIN> and <END> are actually special patterns recognized by **awk**.

Values are assigned to variables from the **awk** command line. The <BEGIN> section is run before these assignments are made.

LEXICAL UNITS

All **awk** programs are made up of lexical units called *tokens*. In **awk** there are eight token types:

1. numeric constants
2. string constants
3. keywords
4. identifiers
5. operators
6. record and field tokens
7. comments
8. tokens used for grouping

Numeric Constants

A *numeric constant* is either a decimal constant or a floating constant. A *decimal constant* is a non-null sequence of digits containing at most one decimal point as in **12**, **12.**, **1.2**, and **.12**. A *floating constant* is a decimal constant followed by **e** or **E** followed by an optional **+** or **-** sign followed by a non-null sequence of digits as in **12e3**, **1.2e3**, **B.2e-3**, and **1.2E+3**. The maximum size and precision of a numeric constant are machine dependent.

String Constants

A string constant is a sequence of zero or more characters surrounded by double quotes as in **","**, **"a"**, **"ab"**, and **"12"**. A double quote is put in a string by preceding it with a backslash, \, as in **"He said, \" Sit! \""**.

A newline is put in a string by using **\n** in its place. No other characters need to be escaped. Strings can be (almost) any length.

Keywords

Strings used as keywords are shown below.

KEYWORDS

BEGIN	END	FILENAME	FS	NF	NR
OFS	ORS	OFMT	RS	break	close
continue	exit	exp	for	getline	if
in	index	int	length	log	next
number	print	printf	split	sprintf	sqrt
string	substr	while			

Identifiers

Identifiers in **awk** serve to denote variables and arrays. An identifier is a sequence of letters, digits, and underscores, beginning with a letter or an underscore. Uppercase and lowercase letters are different.

Operators

Awk has assignment, arithmetic, relational, and logical operators similar to those in the C programming language and regular expression pattern matching operators similar to those in **egrep** and **lex**. For details of these utilities, see the *Utilities Reference Manual*.

The various operators are shown in the table below.

THE AWK OPERATORS

SYMBOL	USAGE	DESCRIPTION
THE ASSIGNMENT OPERATORS:		
=	assignment	
+=	plus-equals	X+=Y is similar to X=X+Y
-=	minus-equals	X-=Y is similar to X=X-Y
*=	times-equals	X*=Y is similar to X=X*Y
/=	divide-equals	X/=Y is similar to X=X/Y
%=	mod-equals	X%=Y is similar to X=X%Y
++	prefix and postfix increments	++X and X++ are similar to X=X+1
-	prefix and postfix decrements	--X and X-- are similar to X=X-1

THE ARITHMETIC OPERATORS:

+	unary and binary plus
-	unary and binary minus
*	multiplication
/	division
%	modulus
(...)	grouping

THE RELATIONAL OPERATORS:

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

THE LOGICAL OPERATORS:

&&	AND
||	OR
!	NOT

THE REGULAR EXPRESSION PATTERN MATCHING OPERATORS:

~	matches
!~	does not match

Record and Field Tokens

$0 is a special variable whose value is that of the current input record. **$1**, **$2**, and so forth, are special variables whose values are those of the first field, the second field, and so forth, of the current input record.

The keyword **NF** (Number of Fields) is a special variable whose value is the number of fields in the current input record. Thus **$NF** has, as its value, the value of the last field of the current input record. Notice that the first field of each record is numbered 1 and that the number of fields can vary from record to record. None of these variables is defined in the action associated with a BEGIN or END pattern, where there is no current input record.

The keyword **NR** (Number of Records) is a variable whose value is the number of input records read so far. The first input record read is 1.

Record Separators

The keyword **RS** (Record Separator) is a variable whose value is the current record separator. The value of **RS** is initially set to newline, indicating that adjacent input records are separated by a newline. Keyword **RS** may be changed to any character, *c*, by executing the assignment statement **RS** = **"***c***"** in an action.

Field Separators

The keyword **FS** (Field Separator) is a variable indicating the current field separator. Initially, the value of **FS** is a blank, indicating that fields are separated by white space, i.e., any non-null sequence of blanks and tabs. Keyword **FS** is changed to any single character, *c*, by executing the assignment statement **F** = **"***c***"** in an action or by using the optional command line argument **-F***c*. Two values of *c* have special meaning, **space** and **\t**. The assignment statement **FS** =**" "** makes white space (a tab or blank) the field separator; and on the command line, **-F\t** makes a tab the field separator.

If the field separator is not a blank, then there is a field in the record on each side of the separator. For

instance, if the field separator is **1**, the record **1XXX1** has three fields. The first and last are null. If the field separator is blank, then fields are separated by white space, and none of the **NF** fields are null.

Multiline Records

The assignment **RS =" "** makes an empty line the record separator and makes a nonnull sequence (consisting of blanks, tabs, and possibly a newline) the field separator. With this setting, none of the first **NF** fields of any record are null.

Output Record and Field Separators

The value of **OFS** (Output Field Separator) is the output field separator. It is put between fields by **print**. The value of **ORS** (Output Record Separators) is put after each record by **print**. Initially, **ORS** is set to a newline and **OFS** to a space. These values may change to any string by assignments such as **ORS = "abc"** and **OFS = "xyz"**.

Comments

A comment is introduced by a **#** and terminated by a newline. For example:

```
#    this line is a comment
```

A comment can be appended to the end of any line of an **awk** program.

Tokens Used for Grouping

Tokens in **awk** are usually separated by nonnull sequences of blanks, tabs, and newlines, or by other punctuation symbols such as commas and semicolons. Braces, {...}, surround actions, slashes, /.../, surround regular expression patterns, and double quotes, "...", surround string constants.

PRIMARY EXPRESSIONS

In **awk**, patterns and actions are made up of expressions. The basic building blocks of expressions are the primary expressions:

numeric constants
string constants
variables
functions

Each expression has both a numeric and a string value, one of which is usually preferred. The rules for determining the preferred value of an expression are explained below.

Numeric Constants

The format of a numeric constant was defined above, in the section called *Lexical Units*. Numeric values are stored as floating point numbers. The string value of a numeric constant is computed from the numeric value. The preferred value is the numeric value. Numeric values for string constants are shown below.

NUMERIC CONSTANT	NUMERIC VALUE	SPRING VALUE
0	0	0
1	1	1
.5	0.5	.5
.5e2	50	50

String Constants

The format of a string constant was defined above, in the section called *Lexical Units*. The numeric value of a string constant is 0 unless the string is a numeric constant enclosed in double quotes. In this case, the numeric value is the number represented. The preferred value of a string constant is its string value. The string value of a string constant is always the string itself. String values for string constants are shown below.

ARITHMETIC OPERATORS

STRING CONSTANT	NUMERIC VALUE	STRING VALUE
""	0	empty space
"a"	0	a
"XYZ"	0	xyz
"0"	0	0

"1"	1	1
".5"	0.5	.5
".5e2"	50	.5e2

Variables

A variable is one of the following:

- identifier
  identifier {expression}
  $term

The numeric value of any uninitialized variable is 0, and the string value is the empty string.

An identifier by itself is a simple variable. A variable of the form *identifier* {expression} represents an element of an associative array named by *identifier*. The string value of *expression* is used as the index into the array. The preferred value of *identifier* or *identifier* {expression} is determined by context.

The variable **$0** refers to the current input record. Its string and numeric values are those of the current input record. If the current input record represents a number, then the numeric value of **$0** is the number and the string value is the literal string. The preferred value of **$0** is string unless the current input record is a number. **$0** cannot be changed by assignment.

The variables **$1**, **$2**, ... refer to fields 1, 2, and so forth, of the current input record. The string and numeric value of **$i** for 1<=*i*<=**NF** are those of the *i*th

field of the current input record. As with **$0**, if the *i*th field represents a number, then the numeric value of **$i** is the number and the string value is the literal string. The preferred value of **$i** is string unless the *i*th field is a number. **$i** may be changed by assignment; the value of **$0** is changed accordingly.

In general, **$term** refers to the input record if *term* has the numeric value 0 and to field *i* if the greatest integer in the numeric value of **term** is *i*. If *i*<0 or if *i*>=100, then accessing **$i** causes **awk** to produce an error diagnostic. If **NF**<*i*<=100, then **$i** behaves like an uninitialized variable.

Accessing **$i** for *i*>**NF** does not change the value of **NF**.

Functions

Awk has a number of built-in functions that perform common arithmetic and string operations. The arithmetic functions are shown below.

FUNCTIONS	
exp	(expression)
int	(expression)
log	(expression)
sqrt	(expression)

These functions (*exp*, *int*, *log*, and *sqrt*) compute the exponential, integer part, natural logarithm, and square root, respectively, of the numeric value of *expression*. The (*expression*) may be omitted; then the function is applied to **$0**. The preferred value of an arithmetic

function is numeric. String functions are shown below.

STRING FUNCTIONS

```
getline
index      (expression1,expression2)
length     (expression)
split      (expression, identifier)
split      (expression1, identifier, expression2)
sprintf    (format, expression1, expression2, ...)
substr     (expression1, expression2)
substr     (expression1, expression2, expression3)
```

The function **getline** causes the next input record to replace the current record. It returns 1 if there is a next input record or a 0 if there is no next input record. The value of **NR** is updated.

The function **index**(*e1,e2*) takes the string value of expressions *e1* and *e2* and returns the first position of where *e2* occurs as a substring in *e1*. If *e2* does not occur in *e1*, index returns 0. For example:

```
index ("abc", "bc")=2
index ("abc", "ac")=0.
```

The function *length* without an argument returns the number of characters in the current input record. With an expression argument, **length**(*e*) returns the number of characters in the string value of *e*. For example:

```
length ("abc")=3
```

```
length (17)=2.
```

The function **split**("*e, array, sep*") splits the string value of expression *e* into fields that are then stored in **array**[*1*], **array**[*2*], ..., **array**[*n*] using the string value of *sep* as the field separator. Split returns the number of fields found in *e*. The function **split**("*e, array*") uses the current value of **FS** to indicate the field separator. For example, after invoking

```
n = split ($0)
```

a[*2*], ..., **a**[*n*] is the same sequence of values as **$1**, **$2** ..., **$NF**.

The function **sprintf**(*f, e1, e2, ...*) produces the value of expressions **e1, e2,** ... in the format specified by the string value of the expression **f**. The format control conventions are those of the **printf** statement in the C programming language (except that the use of the asterisk, *****, for field width or precision is not allowed).

The function **substr**("*string, pos*") returns the suffix of *string* starting at position *pos*. The function **substr**("*string, pos, length*") returns the substring of *string* that begins at position *pos* and is *length* characters long. If *pos* + *length* is greater than the length of *string* then **substr**("*string, pos, length*") is equivalent to **substr**("*string, pos*"). For example:

```
substr("abc", 2, 1) = "b"
substr("abc", 2, 2) = "bc"
substr("abc", 2, 3) = "bc"
```

Positions less than 1 are taken as 1. A negative or zero length produces a null result. The preferred value of **sprintf** and **substr** is string. The preferred value of the remaining string functions is numeric.

TERMS

Various arithmetic operators are applied to primary expressions to produce larger syntactic units called *terms*.

All arithmetic is done in floating point. A term has one of the following forms:

```
primary expression
term binop term
unop term
incremented variable
(term)
```

Binary Terms

In a *term* of the form

```
term1
binop
term2
```

Binop can be one of the five binary arithmetic operators + (addition), - (subtraction), * (multiplication), / (division), or % (modulus). The binary operator is applied to the numeric value of the operands *term1* and *term2*, and the result is the usual numeric value. This numeric value is the preferred value, but it can be interpreted as a string value (see the section called

Numeric Constants, above). The operators *, /, and % have higher precedence than + and -. All operators are left associative.

Unary Term

In a term of the form

```
unop term
```

unop can be unary + or -. The unary operator is applied to the numeric value of *term*, and the result is the usual numeric value which is preferred. However, it can be interpreted as a string value. Unary + and - have higher precedence than *, /, and %.

Incremented Vars

An incremented variable has one of the forms

```
++ var
-- var
var ++
var --
```

The ++*var* has the value *var*+1 and has the effect of *var*=*var*+1. Similarly, --*var* has the value *var*-1 and has the effect of *var*=*var*-1. Therefore, *var*++ has the same value as *var* and has the effect of *var*=*var*+1. Similarly, *var*-- has the same value as *var* and has the effect of *var*=*var*-1. The preferred value of an *incremented* variable is numeric.

Parenthesized Terms

Parentheses are used to group terms in the usual manner.

EXPRESSIONS

An **awk** expression is one of the following:

```
term
term term ...
var asgnop expression
```

Concatenation of Terms

In an expression of the form *term1 term2 ...*, the string value of the terms are concatenated. The preferred value of the resulting expression is a string value. Concatenation of terms has lower precedence than binary + and -. For example, 1+2 3+4 has the string (and numeric) value 37.

Assignment Expressions

An *assignment expression* is one of the forms

```
var asgnop expression
```

where *asgnop* is one of the six assignment operators:

```
=
+=
-=
*=
```

```
/=
%=
```

The preferred value of *var* is the same as that of *expression*.

In an expression of the form

```
var = expression
```

the numeric and string values of *var* become those of *expression*.

```
var op = expression
```

is equivalent to

```
var = var op expression
```

where *op* is one of: **+**, **-**, *****, **/**, **%**. The *asgnops* are right associative and have the lowest precedence of any operator. Thus,

```
a += b *= c-2
```

is equivalent to the sequence of assignments

```
b = b * (c-2)
a = a + b
```

EXAMPLES

The remainder of this chapter undertakes to show the syntax rules of **awk** in action. The material is organized under the following topics:

- input and output
- patterns
- actions
- special features

Note that in the following examples, the **>** symbol in bold type represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

Input and Output

There are two ways to present your program of pattern/action statements to **awk** for processing:

1. If the program is short (a line or two), it is often easiest to make the program the first argument on the command line:

   awk ' *program* ' [*filename...*]

   where *program* is your **awk** program, and *filename...* is an optional input file(s). Note that there are single quotes around the program name in order for the shell to accept the entire string (program) as the first argument to **awk**. For example, write to the shell

```
awk ' /x/ {print} ' file1
```

to run the **awk** program */x/ {print}* on the input file *file1*. If no input file is specified, **awk** expects input from the standard input, **stdin**. You can also specify that input comes from **stdin** by using the hyphen, -, as one of the files. The pattern-action statement

```
awk ' program ' file1 -
```

looks for input from *file1* and from **stdin**. It processes first from *file1* and then from **stdin**.

2. Alternately, if your **awk** program is long or is one you want to preserve for re-use in the future, it is convenient to put the program in a separate file, *awkprog*, for example, and tell **awk** to fetch it from there. This is done by using the **-f** option on the command line, as follows:

```
awk -f awkprog filename...
```

where *filename...* is an optional list of input files that may include **stdin** as is shown above.

These alternative ways of presenting your **awk** program for processing are illustrated by the following:

```
awk ' BEGIN {print "hello, world" exit} '
```

prints

```
hello, world
```

on the standard output when given to the shell.

This **awk** program could be run by putting

```
BEGIN {
        print "hello, world"
        exit
      }
```

in a file named *awkprog*, and then the command

```
awk -f awkprog
```

given to the shell would have the same effect as the first procedure.

Input: Records and Fields

Awk reads its input one record at a time. Unless changed by you, a record is a sequence of characters from the input ending with a newline character or with an end of file. **Awk** reads in characters until it encounters a newline or end of file. The string of characters, thus read, is assigned to the variable **$0**.

Once **awk** has read in a record, it then views the record as being made up of fields. Unless changed by you, a field is a string of characters separated by blanks or tabs.

The following is a sample input file, called *countries* This file contains the area in thousands of square miles, the population in millions, and the continent for the ten

largest countries in the world. (Figures are from 1978; Russia is placed in Asia.)

```
Russia      8650     262      Asia
Canada      3852     24       North America
China       3692     866      Asia
USA         3615     219      North America
Brazil      3286     116      South America
Australia   2968     14       Australia
India       1269     637      Asia
Argentina   1072     26       South America
Sudan       968      19       Africa
Algeria     920      18       Africa
```

The wide spaces are tabs in the original input and a single blank separates North and South from America. We use this data as the input for many of the **awk** programs in this chapter since it is typical of the type of material that **awk** is best at processing (a mixture of words and numbers arranged in fields or columns separated by blanks and tabs).

Each of these lines has either four or five fields if blanks and/or tabs separate the fields. This is what **awk** assumes unless told otherwise. In the above example, the first record is

```
Russia  8650    262     Asia
```

When this record is read by **awk**, it is assigned to the variable **$0**. If you want to refer to this entire record, it is done through the variable, **$0**. For example, the following action:

```
{print $0}
```

prints the entire record.

Fields within a record are assigned to the variables **$1**, **$2**, **$3**, and so forth; that is, the first field of the present record is referred to as **$1** by the **awk** program. The second field of the present record is referred to as **$2**. The ith field of the present record is referred to as **$i** by the **awk** program. Thus, in the above example of the file *countries*, in the first record:

$1 is equal to the string *Russia*
$2 is equal to the integer 8650
$3 is equal to the integer 262
$4 is equal to the string *Asia*
$5 is equal to the null string

and so forth.

To print the continent, followed by the name of the country, followed by its population, use the following command:

```
awk '{print $4, $1, $3}' countries
```

You'll notice that this does not produce exactly the output you may have wanted because the field separator defaults to white space (tabs or blanks). *North America* and *South America* inconveniently contain a blank. Try it again with the following command line:

```
awk -F\t '{print $4, $1, $3}' countries
```

Input: From the Command Line

We have seen above, under the heading *Input and Output*, that you can give your program to **awk** for processing by either including it on the command line enclosed by single quotes, or by putting it in a file and naming the file on the command line (preceded by the **-f** flag). It is also possible to set variables from the command line.

In **awk**, values may be assigned to variables from within an **awk** program. Because you do not declare types of variables, a variable is created simply by referring to it. An example of assigning a value to a variable is:

```
x=5
```

This statement in an **awk** program assigns the value **5** to the variable *x*. This type of assignment can be done from the command line. This provides another way to supply input values to **awk** programs. For example:

```
awk ' {print x }' x=5 -
```

will print the value **5** on the standard output. The minus sign at the end of this command is necessary to indicate that input is coming from **stdin** instead of a file called *x=5*. After entering the command, the user must proceed to enter input. The input is terminated with a **CTRL-d** keystroke.

If the input comes from a file, named *file1* in the example, the command is

```
awk '{print x}' file1
```

It is not possible to assign values to variables used in the BEGIN section in this way.

If it is necessary to change the record separator and the field separator, it is useful to do so from the command line as in the following example:

```
awk -f awkprog RS=":" file1
```

Here, the record separator is changed to the character *:*. This causes your program in the file *awkprog* to run with records separated by the colon instead of the newline character and with input coming from *file1*. It is similarly useful to change the field separator from the command line.

There is a separate option, **-F***x*, that is placed directly after the command **awk**. This changes the field separator from white space to the character *x*. For example:

```
awk -F: -f awkprog file1
```

changes the field separator, **FS**, to the character *:*. Note that if the field separator is specifically set to a tab (that is, with the **-F** option or by making a direct assignment to **FS**), then blanks are not recognized by **awk** as separating fields. However, the reverse is not true. Even if the field separator is specifically set to a blank, tabs are still recognized by **awk** as separating fields.

Output: Printing

An action may have no pattern; in this case, the action is executed for all lines as in the simple printing program

```
{print}
```

This is one of the simplest actions performed by **awk**. It prints each line of the input to the output. More useful is to print one or more fields from each line. For instance, using the file *countries* that was used earlier,

```
awk '{  print $1, $3 }' countries
```

prints the name of the country and the population:

```
Russia 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 14
Sudan 19
Algeria 18
```

A semicolon at the end of statements is optional. **Awk** accepts

```
{print $1}
```

and

```
{print $1;}
```

equally and takes them to mean the same thing. If you want to put two **awk** statements on the same line of an **awk** script, the semicolon is necessary, for example, if you want the number **5** printed:

```
{x=5; print x}
```

Parentheses are also optional with the print statement.

```
{print $3, $2}
```

is the same as

```
{print ($3, $2)}
```

Items separated by a comma in a **print** statement are separated by the current output field separator (normally spaces, even though the input is separated by tabs) when printed. The **OFS** is another special variable that can be changed by you. (These special variables are summarized below.) **print** also prints strings directly from your programs, as with the **awk** script

```
{print "hello, world"}
```

As we have already seen, **awk** makes available a number of special variables with useful values, for example, **FS** and

RS. We introduce two other special variables in the next example. **NR** and **NF** are both integers that contain the number of the present record and the number of fields in the present record, respectively. Thus,

```
{print NR, NF, $0}
```

prints each record number and the number of fields in each record followed by the record itself. Using this program on the file *countries* yields:

```
1 4 Russia      8650    262     Asia
2 5 Canada      3852    24      North America
3 4 China       3692    866     Asia
4 5 USA         3615    219     North America
5 5 Brazil      3286    116     South America
6 4 Australia   2968    14      Australia
7 4 India       1269    637     Asia
8 5 Argentina   1072    26      South America
9 4 Sudan       968     19      Africa
10 4 Algeria    920     18      Africa
```

and the program

```
{print NR, $1}
```

prints

```
1 Russia
2 Canada
3 China
4 USA
5 Brazil
6 Australia
```

```
7 India
8 Argentina
9 Sudan
10 Algeria
```

This is an easy way to supply sequence numbers to a list. **Print**, by itself, prints the input record. Use

```
{print ""}
```

to print an empty line.

Awk also provides the statement **printf** so that you can format output as desired. **print** uses the default format **%.6g** for each numeric variable printed.

printf "*format*", *expr*, *expr*, ...

formats the expressions in the list according to the specification in the string *format*, and prints them. The *format* statement is almost identical to that of **printf** in the C library. Details of this C statement can be found in the *printf*(3C) in the *System Interfaces and Libraries Reference Manual*. For example:

```
{ printf "%10s %6d %6den", $1, $2, $3 }
```

prints **$1** as a string of 10 characters (right justified). The second and third fields (6-digit numbers) make a neatly columned table.

```
Russia      8650    262
Canada      3852    244
```

```
China       3692    866
USA         3615    219
Brazil      3286    116
Australia   2968    14
India       1269    637
Argentina   1072    26
Sudan       968     19
Algeria     920     18
```

With **printf**, no output separators or newlines are produced automatically. You must add them as in this example. The escape characters **\n** (newline), **\t** (tab), **\b** (backspace), and **\r** (carriage return) may be specified.

There is a third way that printing can occur on standard output when a pattern without an action is specified. In this case, the entire record, **$0**, is printed. For example, the program

```
/x/
```

prints any record that contains the character *x*.

There are two special variables that go with printing, **OFS** and **ORS**. By default, these are set to blank and the newline character, respectively. The variable **OFS** is printed on the standard output when a comma occurs in a **print** statement such as

```
{ x="hello"; y="world"
print x,y
}
```

which prints

```
hello world
```

However, without the comma in the print statement as

```
{ x="hello"; y="world"
print x y
}
```

you get

```
helloworld
```

To get a comma on the output, you can either insert it in the print statement as in this case

```
{ x="hello"; y="world"
print x"," y
}
```

or you can change **OFS** in a BEGIN section as in

```
BEGIN {OFS=", "}
{ x="hello"; y="world"
print x, y
}
```

Both of these last two scripts yield

```
hello, world
```

Note that the output field separator is not used when **$0** is printed.

Output: to Different Files

The X/OS operating system shell allows you to redirect standard output to a file. **Awk** also lets you direct output to many different files from within your **awk** program. For example, with our input file *countries*, we want to print all the data from countries of Asia in a file called *ASIA*, all the data from countries in Africa in a file called *AFRICA*, and so forth. This is done with the following **awk** program:

```
{ if ($4 == "Asia") print > "ASIA"
  if ($4 == "Europe") print > "EUROPE"
  if ($4 == "North") print > "NORTH_AMERICA"
  if ($4 == "South") print > "SOUTH_AMERICA"
  if ($4 == "Australia") print > "AUSTRALIA"
  if ($4 == "Africa") print > "AFRICA"
}
```

Flow of control statements are discussed later.

In general, you may direct output into a file after a **print** or a **printf** statement by using a statement of the form

print > "*filename***"**

where *filename* is the name of the file receiving the data. The **print** statement may have any legal arguments to it.

Notice that the filename is quoted. Without quotes, filenames are treated as uninitialized variables and all

output then goes to **stdout**, unless redirected on the command line.

If > is replaced by >>, output is appended to the file rather than overwriting it. Notice that there is an upper limit to the number of files that are written in this way. At present it is ten.

Output: to Pipes

It is also possible to direct printing into a pipe instead of a file. For example:

```
{
if ($2 == "XX") print | "mailx mary"
}
```

where *mary* is a person's login name. Any record with the second field equal to **XX** is sent to the user, *mary*, as mail. **Awk** waits until the entire program is run before it executes the command that was piped to; in this case, the **mailx** command. For example:

```
{
print $1 | "sort"
}
```

takes the first field of each input record, sorts these fields, and then prints them.

Another example of using a pipe for output is the following idiom, which guarantees that its output always goes to your terminal:

```
{
print ... | "cat -v > /dev/tty"
}
```

Only one output statement to a pipe is permitted in an **awk** program. In all output statements involving redirection of output, the files or pipes are identified by their names, but they are created and opened only once in the entire run.

Patterns

A pattern in front of an action acts as a selector that determines if the action is to be executed. A variety of expressions are used as patterns:

- certain keywords
- arithmetic relational expressions
- regular expressions
- combinations of these

BEGIN and END

The keyword, BEGIN, is a special pattern that matches the beginning of the input before the first record is read. The keyword, END, is a special pattern that matches the end of the input after the last line is processed. BEGIN and END thus provide a way to gain control before and after processing for initialization and wrapping up.

As you have seen, you can use BEGIN to put column headings on the output

```
BEGIN {print "Country", "Area", "Population", "Continent"}
      {print}
```

which produces

```
Country Area Population Continent

Russia   8650   262   Asia
Canada   3852   24    North America
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
Australia2968   14    Australia
India    1269   637   Asia
Argentina       1072  26South America
Sudan    968    19    Africa
Algeria  920    18    Africa
```

Formatting is not very good here; **printf** would do a better job and is generally used when appearance is important.

Recall also, that the BEGIN section is a good place to change special variables such as **FS** or **RS**. For example:

```
BEGIN { FS= "\t"
        printf "Country\tArea\tPopulation\tContinent\n\n"}
        {printf "%-10s\t%6d\t%6d\t\t% -14s\n", $1, $2, $3, $4}
END     {print "The number of records is", NR}
```

In this program, **FS** is set to a tab in the BEGIN section and as a result all records in the file *countries* have exactly four fields. Note that if BEGIN is present it is the first pattern; END is the last if it is used.

Relational Expressions

An **awk** pattern is any expression involving comparisons between strings of characters or numbers. For example, if you want to print only countries with more than 100 million population, use

```
$3 > 100
```

This tiny **awk** program is a pattern without an action so it prints each line whose third field is greater than 100 as follows:

```
Russia      8650    262     Asia
China       3692    866     Asia
USA         3615    219     North America
Brazil      3286    116     South America
India       1269    637     Asia
```

To print the names of the countries that are in Asia, type

```
$4 == "Asia" {print $1}
```

which produces

```
Russia
China
India
```

The conditions tested are <, <=, ==, !=, >=, and >. In such relational tests if both operands are numeric, a numerical comparison is made. Otherwise, the operands

are compared as strings. Thus,

```
$1 >= "S"
```

selects lines that begin with **S**, **T**, **U**, and greater, which in this case are

```
USA       3615    219     North America
Sudan     968     19      Africa
```

In the absence of other information, fields are treated as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters and prints the single line

```
Australia       2968      14 Australia
```

Regular Expressions

Awk provides more powerful capabilities for searching for strings of characters than were illustrated in the previous section. These are regular expressions. The simplest regular expression is a literal string of characters enclosed in slashes.

```
/Asia/
```

This is a complete **awk** program that prints all lines that contain any occurrence of the name *Asia*. If a line contains *Asia* as part of a larger word like *Asiatic*, it is also printed (but there are no such words in the *countries* file.)

Awk regular expressions include regular expression forms found in the text editor, **ed**, and the pattern finder, **grep**, in which certain characters have special meanings.

For example, we could print all lines that begin with **A** with

```
/^A/
```

or all lines that begin with **A**, **B**, or **C** with

```
/^[ABC]/
```

or all lines that end with **ia** with

```
/ia$/
```

In general, the circumflex, ^, indicates the beginning of a line. The dollar sign, **$**, indicates the end of the line and characters enclosed in brackets, **[ ]**, match any one of the characters enclosed. In addition, **awk** allows parentheses for grouping, the pipe, **|**, for alternatives, **+** for one or more occurrences, and **?** for zero or one occurrences. For example:

```
/x|y/   {print}
```

prints all records that contain either an **x** or a **y**.

```
/ax+b/  {print}
```

prints all records that contain an **a** followed by one or more **x**'s followed by a **b**. For example, axb, Paxxxxxxxb, QaxxbR.

```
/ax?b/  {print}
```

prints all records that contain an **a** followed by zero or one **x**'s followed by a **b**. For example: ab, axb, yaxbPPP, CabD.

The two characters, **.** and *****, have the same meaning as they have in **ed**, namely, **.** can stand for any character and ***** means zero or more occurrences of the character preceding it. For example:

```
/a.b/
```

matches any record that contains an **a** followed by any character followed by a **b**. That is, the record must contain an **a** and a **b** separated by exactly one character. For example, **/a.b/** matches axb, aPb and xxxxaXbxx, but not ab, axxb.

```
/ax*c/
```

matches a record that contains an **a** followed by zero or more **x**'s followed by a **c**. For example, it matches

```
ac
axc
pqraxxxxxxxxxc901
```

Just as in **ed**, it is possible to turn off the special meaning of metacharacters such as ^ and * by preceding these characters with a backslash. An example of this is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) by using the operators ~ or !~. For example, with the input file *countries* as before, the program

```
$1 ~ /ia$/      {print $1}
```

prints all *countries* whose name ends in *ia*:

```
Russia
Australia
India
Algeria
```

which is not the same as printing all *lines* that end in *ia*.

Combinations of Patterns

A pattern can be made up of similar patterns combined with the operators || (OR), && (AND), ! (NOT), and parentheses. For example:

```
$2 >= 3000 && $3 >= 100
```

selects lines where both area and population are large. For example:

```
Russia      8650    262     Asia
China       3692    866     Asia
USA         3615    219     North America
Brazil      3286    116     South America
```

while

```
$4 == "Asia" || $4 == "Africa"
```

selects lines with *Asia* or *Africa* as the fourth field. An alternate way to write this last expression is with a regular expression:

```
$4 ~ /^Asia|Africa)$/
```

which says to select records where the fourth field matches *Africa* or begins with *Asia*.

The expressions && and || guarantee that their operands are evaluated from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

The pattern that selects an action may also consist of two patterns separated by a comma as in

```
pattern1, pattern2   {action}
```

In this case, the *action* is performed for each line between an occurrence of *pattern1* and the next occurrence of *pattern2* (inclusive). As an example with no action

```
/Canada/,/Brazil/
```

prints all lines between the one containing *Canada* and the line containing *Brazil*. For example:

```
Canada      3852    24      North America
China       3692    866     Asia
USA         3615    219     North America
Brazil      3286    116     South America
```

while

```
NR == 2, NR == 5 { ... }
```

does the action for lines 2 through 5 of the input. Different types of patterns may be mixed as in

```
/Canada/, $4 == "Africa"
```

which prints all lines from the first line containing *Canada* up to and including the next record whose fourth field is *Africa*.

The foregoing discussion of pattern matching pertains to the pattern portion of the pattern/action **awk** statement. Pattern matching can also take place inside an **if** or **while** statement in the action portion. See the section below, entitled *Flow of Control*.

Actions

An **awk** action is a sequence of action statements separated by newlines or semicolons. These action statements do a variety of bookkeeping and string manipulating tasks.

Variables, Expressions, and Assignments

Awk provides the ability to do arithmetic and to store the results in variables for later use in the program. As an example, consider printing the population density for each country in the file *countries*.

```
{print $1, (1000000 * $3) / ($2 * 1000) }
```

(Recall that in this file the population is in millions and the area in thousands.) The result is population density in people per square mile.

```
Russia 30.289
Canada 6.23053
China 234.561
USA 60.5809
Brazil 35.3013
Australia 4.71698
```

```
India 501.97
Argentina 24.2537
Sudan 19.6281
Algeria 19.5652
```

The formatting is not good; using **printf** instead gives

```
{printf "%10s %6.1f\n", $1, (1000000 * $3) / ($2 * 1000)}
```

and the output

```
Russia     30.3
Canada      6.2
China     234.6
USA        60.6
Brazil     35.3
Australia   4.7
India     502.0
Argentina  24.3
Sudan      19.6
Algeria    19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are **+**, **-**, *****, **/**, and **%**.

To compute the total population and number of countries from Asia, we could write

```
/Asia/  { pop += $3; ++n }
END     {print "total population of", n, "Asian countries is", pop}
```

which produces

```
total population of 3 Asian countries is 1765.
```

The operators **++**, **--**, **-=**, **/=**, ***=**, **+=**, and **%=** are available in **awk** as they are in C. The same is true of the **++** operator; it adds one to the value of a variable. The increment operators **++** and **--** (as in C) are used as prefix or as postfix operators. These operators are also used in expressions.

Initialization of Variables

In the previous example, we did not initialize *pop* nor *n*; yet everything worked properly. This is because (by default) variables are initialized to the null string, which has a numerical value of 0. This eliminates the need for most initialization of variables in BEGIN sections. We can use default initialization to advantage in this program, which finds the country with the largest population.

```
maxpop < $3 {
        maxpop = $3
        country = $1
        }
END     {print country, maxpop}
```

which produces

```
China 866
```

Field Variables

Fields in **awk** share essentially all of the properties of variables. They are used in arithmetic and string operations, may be initialized to the null string, or have other values assigned to them. Thus, divide the second field by 1000 to convert the area to millions of square miles by

```
{ $2 /= 1000; print }
```

or process two fields into a third with

```
BEGIN   { FS = "\t" }
        { $4 = 1000 * $3 / $2; print }
```

or assign strings to a field as in

```
/USA/   { $1 = "United States" ; print }
```

which replaces **USA** by **United States** and prints the affected line:

```
United States 3615 219 North America
```

Fields are accessed by expressions; thus, **$NF** is the last field and **$(NF - 1)** is the second to the last. Note that the parentheses are needed since **$NF - 1** is 1 less than the value in the last field.

String Concatenation

Strings are concatenated by writing them one after the other as in the following example:

```
{ x = "hello"
  x = x ", world"
  print x
}
```

which prints the usual

```
hello, world
```

With input from the file *countries*, the following program:

```
/A/       { s = s " " $1 }
END       { print s }
```

prints

```
Australia Argentina Algeria
```

Variables, string expressions, and numeric expressions may appear in concatenations; the numeric expressions are treated as strings in this case.

Special Variables

Some variables in **awk** have special meanings. These are detailed here and the complete list given.

NR Number of the current record.

NF Number of fields in the current record.

FS Input field separator, by default it is set to a blank or tab.

RS Input record separator, by default it is set to the newline character.

$i The *i*th input field of the current record.

$0 The entire current input record.

OFS Output field separator, by default it is set to a blank.

ORS Output record separator, by default it is set to the newline character.

OFMT The format for printing numbers, with the print statement, by default is **%.6g**

FILENAME The name of the input file currently being read. This is useful because **awk** commands are typically of the form

awk -f *program file1 file2 file3 ...*

Type

Variables (and fields) take on numeric or string values according to context. For example, in

```
pop += $3
```

pop is presumably a number, while in

```
country = $1
```

country is a string. In

```
maxpop < $3
```

the type of *maxpop* depends on the data found in **$3**. It is determined when the program is run.

In general, each variable and field is potentially a string or a number, or both at any time. When a variable is set by the assignment

```
v = expr
```

its type is set to that of *expr*. (Assignment also includes **+=**, **++**, **-=**, and so forth.) An arithmetic expression is of the type **number**; a concatenation of strings is of type **string**. If the assignment is a simple copy as in

```
v1 = v2
```

then the type of *v1* becomes that of *v2*.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to strings if necessary and the comparison is made on strings.

The type of any expression may be coerced to numeric by a subterfuge such as

```
expr + 0
```

and to string by

```
expr ""
```

This last expression is **string** concatenated with the null string.

Arrays

As well as ordinary variables, **awk** provides 1-dimensional arrays. Array elements are not declared; they spring into existence by being mentioned. Subscripts may have any non-null value including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input line to the **NR**th element of the array *x*. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the following **awk** program:

```
        { x[NR] = $0 }
END     { ... program ... }
```

The first line of this program records each input line into the array *x*. In particular, the following program

```
{ x[NR] = $1}
```

(when run on the file *countries*) produces an array of elements with

```
x[1] = "Russia"
x[2] = "Canada"
x[3] = "China"
```

and so forth.

Arrays are also indexed by non-numeric values that give **awk** a capability rather like the associative memory of Snobol tables. For example, we can write

```
/Asia/{pop["Asia"] += $3}
/Africa/{pop[Africa] += $3}
END     {print "Asia=" pop["Asia"], "Africa="pop["Africa"] }
```

which produces

```
Asia=1765 Africa=37
```

Notice the concatenation. Also, any expression can be used as a subscript in an array reference. Thus,

```
area[$1] = $2
```

uses the first field of a line (as a string) to index the array *area*.

SPECIAL FEATURES

In this final section we describe the use of some special **awk** features.

Built-In Functions

The function **length** is provided by **awk** to compute the length of a string of characters. The following program prints each record preceded by its length:

```
{print length, $0 }
```

In this case the variable *length* means **length($0)**, the length of the present record. In general, **length(*x*)** will return the length of *x* as a string.

With input from the file *countries*, the following **awk** program will print the longest country name:

```
length($1) > max  {max = length($1); name = $1 }
END               {print name}
```

The function **split**

```
split(s, array)
```

assigns the fields of the string *s* to successive elements of the array, *array*.

For example;

```
split("Now is the time", w)
```

assigns the value **Now** to *w*[1], **is** to *w*[2], **the** to *w*[3], and **time** to *w*[4]. All other elements of the array **w**[ ], if any, are set to the null string. It is possible to have a character other than a blank as the separator for the elements of *w*. For this, use **split** with three elements.

```
n = split(s, array, sep)
```

This splits the string *s* into *array*[1], ..., *array*[n]. The number of elements found is returned as the value of **split**. If the *sep* argument is present, its first character is used as the field separator; otherwise, **FS** is used. This is useful if in the middle of an **awk** script, it is necessary to change the record separator for one record. Also provided by **awk** are the math functions

```
sqrt
log
exp
int
```

They provide the square root function, the base *e* logarithm function, exponential and integral part functions. This last function returns the greatest integer less than or equal to its argument. These functions are the same as those of the C math library

(**int** corresponds to the *libm* **floor** function) and so they have the same return on error as those in *libm*. (See the *System Interfaces and Libraries Reference Manual*.)

The function **substr**

```
substr(s,m,n)
```

produces the substring of *s* that begins at position *m* and is at most *n* characters long. If the third argument (*n* in this case) is omitted, the substring goes to the end of *s*. For example, we could abbreviate the country names in the file *countries* by

```
{ $1 = substr($1, 1, 3); print }
```

which produces

```
Rus 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa
```

If *s* is a number, **substr** uses its printed image:

```
substr(123456789,3,4)=3456.
```

The function **index**

```
index (s1,s2)
```

returns the leftmost position where the string *s2* occurs in *s1* or zero if *s2* does not occur in *s1*.

The function **sprintf** formats expressions as the **printf** statement does but assigns the resulting expression to a variable instead of sending the results to **stdout**. For example:

```
x = sprintf("%10s %6d", $1, $2)
```

sets *x* to the string produced by formatting the values of **$1** and **$2**. The *x* may then be used in subsequent computations.

The function **getline** immediately reads the next input record. Fields **NR** and **$0** are set but control is left at exactly the same spot in the **awk** program. **getline** returns 0 for the end of file and a 1 for a normal record.

Flow of Control

Awk provides the basic flow of control statements within actions:

if ... else

while

for

with statement grouping as in C language.

The **if** statement is used as follows:

if (*condition*) *statement1* **else** *statement2*

The *condition* is evaluated; and if it is true, *statement1* is executed; otherwise, *statement2* is executed. The **else** part is optional. Several statements enclosed in braces, **{ }**, are treated as a single statement. Rewriting the maximum population computation from the pattern section with an **if** statement results in

```
{       if (maxpop < $3) {
                maxpop = $3
                country = $1
        }
}
END     { print country, maxpop }
```

There is also a **while** statement in **awk**.

while (*condition*) *statement*

The *condition* is evaluated; if it is true, the *statement* is executed. The *condition* is evaluated again, and if true, the *statement* is executed. The cycle repeats as long as the condition is true. For example, the following prints all input fields, one per line:

```
{       i = 1
        while (i <= NF) {
                print $i
                ++i
```

```
        }
}
```

Another example is the Euclidean algorithm for finding the greatest common divisor of **$1** and **$2**:

```
{printf "the greatest common divisor of " $1 "and ", $2, "is"
while ($1 != $2) {
   if ($1 > $2) $1 -= $2
   else     $2 -= $1
}
printf $1 "\n"
}
```

The **for** statement is like that of C, which is:

for (*expression1*;*condition*;*expression2*) *statement*

So

```
{       for (i = 1 ; i <= NF; i++)
                print $i
}
```

is another **awk** program that prints all input fields, one per line.

There is an alternate form of the **for** statement that is useful for accessing the elements of an associative array in **awk**.

for (*i* **in** *array*) *statement*

executes *statement* with the variable *i* set in turn to each subscript of *array*. The subscripts are each accessed once but in undefined order. Chaos will ensue if the variable *i* is altered or if any new elements are created within the loop. For example, you could use the **for** statement to print the record number followed by the record of all input records after the main program is executed.

```
        { x[NR] = $0 }
END     { for(i in x) print i, x[i] }
```

A more practical example is the following use of strings to index arrays to add the populations of countries by continents:

```
BEGIN   {FS="\t"}
        {population[$4] += $3}
END     {for(i in population)
                print i, population[i]
        }
```

In this program, the body of the **for** loop is executed for *i* equal to the string *Asia*, then for *i* equal to the string *North America*, and so forth until all the possible values of *i* are exhausted; that is, until all the strings of names of countries are used. Note, however, the order the loops are executed is not specified. If the loop associated with *Canada* is executed before the loop associated with the string *Russia*, such a program produces

```
South America 26
Africa 16
Asia 637
Australia 14
```

North America 219

Note that the expression in the condition part of an **if**, **while**, or **for** statement can include the following:

1. relational operators like <, <=, >, >=, ==, and !=.

2. regular expressions that are used with the matching operators ~ and !~

3. the logical operators ||, &&, and !

4. parentheses for grouping

The **break** statement (when it occurs within a **while** or **for** loop) causes an immediate exit from the **while** or **for** loop.

The **continue** statement (when it occurs within a **while** or **for** loop) causes the next iteration of the loop to begin.

The **next** statement in an **awk** program causes **awk** to skip immediately to the next record and begin scanning patterns from the top of the program. (Note the difference between **getline** and **next**. **getline** does not skip to the top of the **awk** program.)

If an **exit** statement occurs in the BEGIN section of an **awk** program, the program stops executing and the END section is not executed (if there is one).

An **exit** that occurs in the main body of the **awk** program causes execution of the main body of the **awk** program to stop. No more records are read, and the END section is executed.

An **exit** in the END section causes execution to terminate at that point.

Report Generation

The flow of control statements in the last section are especially useful when **awk** is used as a report generator. **awk** is useful for tabulating, summarizing, and formatting information. We have seen an example of **awk** tabulating populations in the last section. Here is another example of this. Suppose you have a file *prog.usage* that contains lines of three fields called *name*, *program*, and *usage*:

```
Smith   draw    3
Brown   eqn     1
Jones   nroff   4
Smith   nroff   1
Jones   spell   5
Brown   spell   9
Smith   draw    6
```

The first line indicates that Smith used the **draw** program three times. If you want to create a program that has the total usage of each program along with the names in alphabetical order and the total usage, use the following program, called *list1*:

```
        {use[$1 "\t" $2] += $3}
END     {for (np in use)
                print np "       " use[np]  | "sort +0 +2nr"
        }
```

This program produces the following output when used on the input file, *prog.usage*.

```
Brown   eqn     1
Brown   spell   9
Jones   nroff   4
```

```
Jones   spell   5
Smith   draw    9
Smith   nroff   1
```

If you would like to format the previous output so that each name is printed only once, pipe the output of the previous **awk** program into the following program, called *format1*:

```
{       if ($1 != prev) {
                print $1 ":"
                prev = $1
        }
        print " " $2 " " $3
}
```

The variable *prev* is used to ensure each unique value of **$1** prints only once. The command

```
awk -f list1 prog.usage | awk -f format1
```

gives the output

```
Brown:
        eqn     1
        spell   9
Jones:
        nroff   4
        spell   5
Smith:
        draw    9
        nroff   1
```

It is often useful to combine different **awk** scripts and other shell commands such as **sort**, as was done in the *list1* script.

Cooperation with the Shell

Normally, an **awk** program is either contained in a file or enclosed within single quotes as in

```
awk '{print $1}' ...
```

Since **awk** uses many of the same characters the shell does (such as **$** and the double quote) surrounding the program by single quotes ensures that the shell passes the program to **awk** intact.

Consider writing an **awk** program to print the *n*th field, where *n* is a parameter determined when the program is run. That is, we want a program called *field* such that

```
field n
```

runs the **awk** program

```
awk '{print $n}'
```

How does the value of *n* get into the **awk** program?

There are several ways to do this. One is to define *field* as follows:

```
awk '{print $'$1'}'
```

Spaces are critical here: as written there is only one argument, even though there are two sets of quotes. The **$1** is outside the quotes, visible to the shell, and therefore substituted properly when *field* is invoked.

Another way to do this job relies on the fact that the shell substitutes for **3$** parameters within double quotes.

```
awk "{print \$ $1}"
```

Here the trick is to protect the first **$** with a ****; the **$1** is again replaced by the number when *field* is invoked.

Multidimensional Arrays

You can simulate the effect of multidimensional arrays by creating your own subscripts. For example:

```
for (i = 1; i <= 10; i++)
        for (j = 1; j <= 10; j++)
                mult[i "," j] = . . .
```

creates an array whose subscripts have the form *i,j*; that is, 1,1; 1,2 and so forth; and thus simulate a 2-dimensional array.

GREP: file pattern search utility

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **grep** utility, which is used to search a file for a specific pattern called a *regular expression.* Output consists of the lines that match the pattern. Options are available to alter the form of the output. Other members of the **grep** family are described in the *Advanced Utilities User Guide.*

SYNTAX

```
grep [options] expr [ file ... ]
```

DESCRIPTION

The **grep** utility will search through a file for a specific word, phrase or group of characters. The command name is short for ***g**lobally search for a **r**egular **e**xpression and **p**rint.* Put simply, a regular expression is any specified pattern of characters, taking the form of a word, a phrase or an equation.

Note that there are two other forms of the **grep** utility, called **egrep** and **fgrep**. These are described in the *grep*(1) entry in the *Utilities Reference Manual*, and in their own tutorial in the *Advanced Utilities User Guide.*

The options and arguments to **grep** are as follows:

options the following options are available:

-v all lines except those matching the regular expression are printed.

-c a count of the lines matching the regular expression is printed.

-i the case of characters is ignored when patterns are being searched for.

-s the error messages displayed in the case of non-existent or unreadable files are supressed.

-l only the names of files containing patterns that match the regular expression are printed.

-n lines are preceded by their line numbers

-b lines are preceded by the number of the block on which they were found.

expr the regular expression to be matched. Note that when specifying regular expressions that contain characters meaningful to the shell, these should be escaped. Details of how to do this are to be found in the shell tutorials in the *Shell / C Shell X/OS Command Language User Guide*.

file specifies the file or files to be searched.

EXAMPLES

In the following examples, **grep** is used to locate patterns that may or may not be contained in a file or files. The > symbol in bold type represents the system prompt, while **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

In the first example, **grep** is used to locate any lines that contain the word *automation* in the file *mechanics.txt*.

```
>grep automation mechanics.txt CR
and office automation software.

>
```

The output consists of all the lines in the file *mechanics.txt* that contain the specified pattern, that is, the word *automation*. In this case, there was only one line.

If the pattern contains multiple words or any character that conveys special meaning to the X/OS system, (such as **$**, **|**, *, ?, and so on), the entire pattern must be enclosed in single quotes. (For an explanation of the special meaning for these and other characters see the shell tutorials.)

The second example expands the first example, by attempting to locate any lines in *mechanics.txt* containing the pattern *office automation*. The command line and the system's response are as follows:

```
>grep 'office automation' mechanics.txt CR
and office automation software.
```

Where the pattern might be in one of several files, the following command line can be used. The files to be searched are *mechanics.txt* and *letter1.doc*.

```
>grep 'office automation' mechanics.txt letter1.doc CR
mechanics.txt:and office automation software.

>
```

The output states that the pattern *office automation* is found once in the *mechanics.txt* file, but is not present in *letter1.doc*.

The next example uses the **-v** and **-i** options to print all lines except those that match the regular expression, where the case of the characters being checked is ignored. Note that the example begins by displaying the contents of *textfile.txt* using the **cat** command, which has its own tutorial in this manual. The second command line uses **grep** to search for the word *chosism*. All lines not including this pattern are printed. Note that the **-i** option tells **grep** to ignore the case of the letters in the pattern: this means that *Chosism* and *chosism* are considered to be same.

```
>cat textfile.txt CR
What is chosism?
Chosism (a French term, literally thingism),
describes the obsessively detailed description
of trivial objects, used by authors of the New
Novel school of literature.

>grep -v -i chosism textfile.txt CR
describes the obsessively detailed description
of trivial objects, used by authors of the New
Novel school of literature.
```

SED: stream editor

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **sed** utility, which is a non-interactive means of editing input files according to a script of commands. It is particularly useful where the editing operations are complex, or where the input files are very large. The new version of the file is either displayed on the standard output, or directed to a new file.

SYNTAX

```
sed [-n] [-e script] [-f sfile] [file ...]
```

DESCRIPTION

The **sed** editor operates by copying a line from the input file into an area called a *pattern space*. Each relevent editing instruction is applied to the data in the pattern space, and the revised data is copied to the standard output. The pattern space is then deleted, and the cycle begins with a new set of data, although there are functions that retain the some or all of the contents of the pattern space for further operations. This involves the use of an area called a *hold space*. Because intermediate files are not created, the only limit on the amount of input material that can be handled by **sed** is the availability of disk space.

The options and arguments to **sed** are as follows:

-n specifies that only those line accessed by one or more addresses should be printed.

-e *script* specifies a single operation to be carried out on the relevent data. The format of a script is given below.

-f *sfile* one or more editing instructions are contained in the file identified by *sfile*. Each instruction should start on a new line.

file identifies the input file or files to be editied.

A script can consist of either a single editing instruction entered on the command line, or a series of instructions collected in a script file. This file will be used by **sed** to carry out the editing procedures required. Instructions are carried out in the order they appear in the script file, unless the flow-of-control statements are used.

An instruction has the following format:

[address [,address]] function [argument [,argument]]

These elements are as follows:

address selects the range of data to be edited (and therefore determines whether the current contents of the pattern space are relevent). An address can take the following forms:

- a decimal number identifying the line to be edited. If more than one file is input for editing, lines should be counted cumulatively across the whole range of files. Therefore, where files *file1*, *file2* and *file3* are to be input, each 100 lines in length, the 20th line of *file2* is identified as 120.

- a **$** character, which identifies the last line of the last input file.

- a *context* address, which is located according to the presence of a particular string of characters. When this method is used, **sed** searches for the specified string, and performs the editing operation only when the string is found. Such a string is often called a *regular expression*. Full details of the **sed** regular expressions can be found in the **ed** tutorial in this manual, and the *ed*(1) entry in the *Utilities Reference Manual*. Exceptions to **ed** are listed in the *sed*(1) entry of the *Utilities Reference manual*. The following list shows some of the notations available:

  . matches any character

  *char***expr* a single character, *char*, followed by an asterisk, matches 0 or more occurences of *char* followed by *expr*.

  ^*expr* indicates that the expression following the ^ occurs at the *beginning* of a line.

  [ ] matches any single character in the list enclosed in the brackets. The list may take the form of a series of characters, or a range of characters specified by the hyphen, -.

Note that editing functions can be applied to data *not* selected by *address*, by using the negation operator, !. This character precedes the editing function.

Note also that the *address* parameter is

optional. This means that an editing script that does not specify an address will operate on all lines in the input files or files. Where two addresses are given, editing is carried out on all lines occuring between the two addresses.

function this parameter specifies the editing operation to be performed on the data specified by *address*.

argument

EXAMPLES

In this section, a series of examples will be given which illustrate the principles of addressing and editing the material stored in two input files. In the sample screens, the > character in bold type represents the system prompt, while **CR** indicates that the carriage return key should be pressed.

The first screen uses the **cat** utility to display the contents of two small input files. These are called *shelley1* and *shelley2*, and together contain a poem. These files are used throughout the examples. The **cat** utility is described in its own tutorial in this manual.

```
>cat shelley1 CR
I met a traveller from an antique land
Who said: Two vast and trunkless legs of stone
Stand in the desert. Near them on the sand,
Half sunk, a shatter'd visage lies, whose frown
And wrinkled lip and sneer of cold command
Tell that its sculptor well those passions read
Which yet survive, stamp'd on these lifeless things,
The hand that mock'd them and the heart that fed;
```

```
>cat shelley2 CR
And on the pedestal these words appear:
`My name is Ozymandias, king of kings:
Look on my works, ye Mighty, and despair!'
Nothing beside remains. Round the decay
Of that colossal wreck, boundless and bare,
The lone and level sands stretch far away.

>
```

The following section uses these files to illustrate the use of the three addressing methods.

USING ADDRESSES

The following list of examples illustrates how addresses work when using **sed**. In all cases, the command line used is as follows, where the **>** character in bold type represents the X/OS system prompt, and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>sed -n -f list1 shelley1 shelley2 CR
```

The contents of the script file called *list1*, in sequence, are as follows:

2 p prints the second line of the first file.

2,4 p prints lines 2 to 4 inclusive of the first file.

10,12 p prints lines 2 to 4 inclusive of the second file.

$ p points to the last line of the second file.

/ar/ p searches the input files for all lines containing the regular expression *ar*, and accesses the following lines:

```
Stand in the desert. Near them on the sand,
The hand that mock'd them and the heart that fed;
And on the pedestal these words appear:
Of that colossal wreck, boundless and bare,
The lone and level sands stretch far away.
```

/^ar/ p searches the input files for all lines *beginning* with the pattern *ar*. It therefore accesses no lines because this string does not occur at the beginning of a line.

/./ p prints all lines because all lines contain at least one character.

/[bf]ar/ p prints any line containing the string *ar* preceded by any one of the characters in the square brackets. The following lines are therefore accessed:

```
Of that colossal wreck, boundless and bare,
The lone and level sands stretch far away.
```

/[b-f]ar/ p prints any line containing the string *ar* preceded by any one of the characters from the group *b*, *c*, *d*, *e* or *f*. The following lines are therefore accessed:

```
Stand in the desert. Near them on the sand,
The hand that mock'd them and the heart that fed;
And on the pedestal these words appear:
```

Of that colossal wreck, boundless and **bare**,
The lone and level sands stretch **far** away.

These forms of addressing can all be used in conjunction with the **sed** functions. The **p** (print) function has been used throughout. The others are used in the same way. The next sections describe the functions, which may be divided into the following types:

- Whole-Line Functions
- Substitute Functions
- Input/Output Functions
- Multiple Line Functions
- Hold and Get Functions
- Flow of Control Functions

THE WHOLE-LINE FUNCTIONS

There are four whole-line functions in **sed**, as follows:

d the **d** (delete) function deletes from the named files all lines matching the addresses. As soon as a line has been deleted, no further commands can be carried out on it, and a new line is read from the input. The list of commands begins at the beginning with the newly read line. The maximum number of addresses is two.

n the **n** command reads a line from the input, writing it the output if required. The next line in the input is then read into the pattern space. The next command in the list is then performed. The maximum number of addresses is two.

a*text* the **a** (append) command places *text* on the output *after* the addressed line, then reads the next line from the input. Note that *text* may consist of any number of lines. The maximum number of addresses is one.

i*text* the **i** (insert) command places *text* on the output *before* the addressed line, then reads the next line from the input. Note that *text* may consist of any number of lines. The maximum number of addresses is one.

c*text* the **c** (change) command deletes the current line and replaces it with *text*. The maximum number of addresses is two.

In the following examples, the command line used is the same as that used above. In the first example, the contents of *list1* are as follows:

```
n
a\**new line inserted here
d
```

This produces the following output. Only the first few lines are illustrated:

```
I met a traveller from an antique land
**new line inserted here
Stand in the desert. Near them on the sand,
**new line inserted here
And wrinkled lip and sneer of cold command
**new line inserted here
Which yet survive, stamp'd on these lifeless things,
**new line inserted here
```

The same effect could have been produced with the following version of *list1*:

```
n
i\
**new line inserted here
d
```

and again with the following:

```
n
c\
**new line inserted here
```

SUBSTITUTE FUNCTIONS

These functions change parts of a line selected by way of a context address. A substitute operation takes the following general form:

s/*regular expression*/*replacement*/*flags*

while a transformation operation takes the following form:

y/*string1*/*string2*/

The components of the general substitution form are as follows: the **s** (substitute) command looks for strings pointed to by the context address contained in the regular expression, and replaces them with the second string. The flags are used to alter the behaviour of the

operation, as follows:

g the **g** (global) flag tells **sed** to substitute all *non*-overlapping instances of the regular expression with the replacement string, not just the first one.

p the **p** (print) flag tells **sed** to print the pattern space if a replacement was made.

w *wfile* the **w** (write) command tells **sed** to append the pattern space to a file idenfified by *wfile* if a replacement was made.

In the following example, the command line is the same as that used above, but the contents of *list1* are as follows:

```
s/^T/**T/w shelley3
```

Used for no particular reason, this script would have the effect of creating a file called *shelley3*, containing the following:

```
**Tell that its sculptor well those passions read
**The hand that mock'd them and the heart that fed;
**The lone and level sands stretch far away.
```

The script

```
s/[b-f]ar/***/gp
```

would produce the following on the standard output:

```
Stand in the desert. N*** them on the sand,
The hand that mock'd them and the h***t that fed;
And on the pedestal these words app***:
Of that colossal wreck, boundless and ***e,
The lone and level sands stretch *** away.
```

The transformation operation replaces the characters in *string1* with the characters in *string2*. This implies that the two strings must be the same length. In this way, if the contents of *list1* are

```
y/abc/ABC/
p
```

the first part of the poem would become

```
I met A trAveller from An Antique lAnd
Who sAid: Two vAst And trunkless legs of stone
StAnd in the desert. NeAr them on the sAnd,
HAlf sunk, A shAtter'd visAge lies, whose frown
And wrinkled lip And sneer of Cold CommAnd
```

INPUT/OUTPUT FUNCTIONS

The input/output functions available to **sed** are as follows:

p the **p** (print) command displays the addressed lines on the standard output. Note that where a sequence of commands makes up a single script, **p** will display the lines in their existing form, irrespective of any editing operations to be performed by commands occurring later in the sequence. Up to two addresses may be given.

w *file* the **w** (write) command writes the addressed lines to the file identified by *file*. If the file exists, the pattern space is appended: if not, it is created. Note that where a sequence of commands make up a single script, **w** will write the pattern space in its existing form, irrespective of any editing operations to be performed by commands occurring later in the sequence. Up to two addresses may be given.

r *file* the **r** (read) command displays the contents of the named file after the addressed line. The next input line is then read. Note that a space must appear the two elements of the function.

l the **l** (list) command displays the current contents of the pattern space on the standard output in a non-ambiguous fashion. That is, non-printable characters are displayed in the form of two-digit ASCII codes, and long lines are folded.

= this function displays the current line number on the standard output.

In the following examples, the command line is as declared above. The contents of *listl* are as follows:

```
p
/land/r Notel
```

The file called *Notel* contains a brief biographical paragraph. It's contents are displayed after the line containing the string *land*:

```
I met a traveller from an antique land
Note: Percy Bysshe Shelley (1792-1822),
radical, traveller and poet. Friend of
```

```
Byron and Godwin.
Who said: Two vast and trunkless legs of stone
Stand in the desert. Near them on the sand,
Half sunk, a shatter'd visage lies, whose frown
.
.
```

MULTIPLE LINE FUNCTIONS

These functions, all of which consist of a single upper case letter, apply to multiple lines of input separated by embedded newline characters. They provide a valuable multiple line pattern matching facility.

N the **N** command appends the next line of input to the pattern space, with an embedded newline. This permits pattern matching to extend across the newline.

D the **D** (delete) command deletes up to and including the first newline character in the pattern space. If the result of this operation is to empty the pattern space (that is, the only newline was the terminal newline, a new line of input is read into the pattern space).

P The **P** (print) command copies the contents of the pattern space up to the first newline character to the output.

Note that the **D** and **P** commands are equivalent to **d** and **p** where no embedded newline characters exist in the pattern space.

HOLD AND GET FUNCTIONS

The so-called hold and get functions allow the saving of part of the pattern space for further use in later operations. they are as follows:

h the **h** (hold) command copies the contents of the pattern space into the hold space. Any existing contents of the hold space are over-written.

H the **H** command appends the contents of the pattern space to the contents of the hold space.

g the **g** (get) command copies the contents of the hold space into the pattern space. Any existing contents of the pattern space are over-written.

G the **G** command appends the contents of the hold space to contents of the pattern space.

x the **x** (exchange) command exchanges the contents of the pattern and hold spaces.

FLOW OF CONTROL FUNCTIONS

These functions are not directly concerned with the editing of text, but have the effect of controlling the sequence of commands that are applied to text from within a script file. The flow of control functions are as follows:

! the NOT function tells **sed** to apply editing commands to those lines of text *not* selected by the addresses.

{ } the brackets are used to group commands together. Commands combined in this way are applied as a group to lines selected by the addresses. Note that groups may be nested so that groups occur within groups. Each command

may appear on a new line.

b *label* the **b** (branch) command tells **sed** to carry out the commands contained in the sub-routine identified by **:** *label*. A **b** function with no label is taken to be a call to branch to the end of the script.

t *label* the **t** (test) command branches to the sub-routine identified by *label*, and tests whether any successful substitutions have been made since the last reading of an input line or execution of a **t** function. If **t** is used without a label, **sed** branches to the end of the script.

: *label* this construction is used to identify a sub-routine of commands. These are accessed by the **b** and **t** functions described above.

QUITTING FROM SED

To escape from **sed** back to the shell, the **q** (quit) function is used. Any suitable text output operations are performed.

INTRODUCTION

This eighth chapter contains three tutorials, covering the basic X/OS commands used to check on the state of the system. They are as follows:

DF reports free disk blocks and inodes

DU summarises disk usage

SUM prints a file's checksum and block count

The tutorials are presented in alphabetical order.

DF: reports free disk blocks and inodes

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **df** utility, which reports on the availability of free memory blocks and free inodes on mounted file systems, directories and mounted memory resources by examining the counts kept in the super blocks.

SYNTAX

---

df [-i] [-ttype] [filesys ...] [filename ...]

DESCRIPTION

If executed with no options or arguments, **df** reports on the available (local and remote) mounted file systems. This behaviour can be modified using the following options and arguments:

-i reports the number of used and free inodes.

-t*type* reports on file systems of a given *type*, for example 4.2 or nfs.

filesys the file system to be reported may be specified either by device name or by mount point directory name.

filename specifies a directory stored on a device. Information on the device that contains the directory is reported.

EXAMPLES

In this example, **df** is used without arguments to produce a report on all the currently mounted file systems. In the sample screens, the **>** symbol in bold represents the X/OS system prompt, while **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>df CR
Filesystem  kbytes  used  avail  capacity  Mounted on
/dev/ex0b   7445    4714  1986   70%       /
/dev/ex0d   42277   35291 2758   93%       /usr

>
```

DU: disk usage summary utility

INTRODUCTION

This is a short tutorial-style introduction to the **du** utility. It's name stands for *disk usage*. It summarises the disk space used in total, for for specified directory and file systems.

SYNTAX

```
du [-a] [-snames] [-r] names
```

DESCRIPTION

Du is used to return the number of blocks contained in directories and files held on the system. The notation . is used to indicate *all* directories and files. The options and arguments to **du** are as follows:

-a causes a block count to be generated for each file to be found in the current and lower directories

-s*names* causes a total block count to be generated for the directory or directories identified by *names*.

-r causes **du** to print messages warning about directories that cannot be read. Normally, **du** is silent about such directories.

names identifies the directory or directories to be checked by **du**.

EXAMPLES

The first example uses the simplest form of **du**, to generate a block count for a directory called */spike*. Directories are explained in detail in the Introduction to this manual. Note that the output is recursive, in that it checks the lowest directories, then moves up, before counting other lower directories. The numbers on the left are the block counts.

Remember that the > symbol in bold face represents the system prompt, and that the **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>du /spike CR
292     /spike/project1/docs
87      /spike/project1/macros
380     /spike/project1
130     /spike/project2/docs/notes
221     /spike/project2/docs/update
352     /spike/project2/docs
22      /spike/project2/memos
375     /spike/project2
756     /spike

>
```

Note that sub-totals are obtained by adding the block counts of lower-level directories, and adding 1 for the parent directory. In this way, */spike/project1* has a block count of 380, consisting of 292 plus 87 for its constituent directories, plus 1 for itself.

The second example uses the **-s** option to print a count for the named directories.

```
>du -s /spike CR
756     /spike

>du -s /spike/project2/docs CR
352     /spike/project2/docs

>
```

SUM: print a file's checksum and block count

INTRODUCTION

This is a tutorial introduction to the **sum** utility which calculates and prints a 16-bit checksum for a specified file, and prints the number of blocks in the file. This command is often used to check that data has been successfully transmitted over a communications system. In such cases, the checksum of the file is calculated before and after transmission, and the two results are compared. A match indicates that transmission was successful. Note that it is essential that the same form of the command be used at both ends.

SYNTAX

```
sum [-r] file
```

DESCRIPTION

The arguments and options to **sum** are as follows:

-r indicates that an alternative checksum algorithm is to be used. Note that if this option is used to calculate the checksum at *source*, it must also be used at *destination*.

file specifies the file to be checked. Note that if necessary, a complete pathname can be entered.

EXAMPLES

The following examples show the effect of using the two different checksum algorithms on a file called *testfile*. Remember that the > symbol in bold represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line. The first output field is the checksum, the second is the number of blocks occupied by the file, and the third confirms the name of the file.

```
>sum testfile CR
2496 1 testfile

>sum -r testfile CR
55792 1 testfile

>
```

INTRODUCTION

This chapter of the manual consists of a single tutorial. It provides coverage of the **mail** utility, which is used for communicating with other users.

MAIL: message transmission utility

INTRODUCTION

The X/OS system offers a choice of commands that enable you to communicate with other X/OS system users. Specifically, they allow you to: send and receive messages from other users (on either your system or another X/OS system); exchange files; and form networks with other X/OS systems. Through networking, a user on one system can exchange messages and files between computers, and execute commands on remote computers.

The available range of communications utilities is as follows:

For exchanging messages: **mail**, **mailx**, **uname** and **uuname**

For exchanging files: **uucp**, **uuto** and **uustat**

For networking: **cu** and **uux**

This chapter is a tutorial-style introduction to the **mail** utility for the transmission of messages between users and systems. Also covered are the **uname** and **uuname** utilities. The **uname** utility provides the name of the current system, while **uuname** lists the remote systems currently connected to the current system.

EXCHANGING MESSAGES

To send messages you can use either of the **mail** or **mailx** commands. **Mailx** is described in the *Advanced Utilities User Guide*. These commands deliver your message to a file belonging to the recipient. When the recipient logs in (or while already logged in), he or she receives a message that says *you have mail*. The recipient can use either the **mail** or **mailx** command to read your message and reply at his or her leisure.

The main difference between **mail** and **mailx** is that only **mailx** offers the following features:

- a choice of text editors (**ed** or **vi**) for handling incoming and outgoing messages
- a list of waiting messages that allows the user to decide which messages to handle and in what order
- several options for saving files
- commands for replying to messages and sending copies (of both incoming and outgoing messages) to other users

You can also use **mail** or **mailx** to send short files containing memos, reports, and so on. However, if you want to send someone a file that is over a page long, use one of the commands designed for transferring files: **uuto** or **uucp**. These are also described in the *Advanced Utilities User Guide*.

SYNTAX

```
mail [-epqr] [-ffile]

mail [-t] persons
```

Note that the syntax of the **uname** and **uuname** commands is defined later in the tutorial.

DESCRIPTION

The first form of **mail** is used to read incoming messages, and has the following options and arguments:

-e returns a value of 0 if the user has mail, and 1 if there is no mail. This exit value is of use when writing shell scripts. These are explained in the two shell tutorials in the *Shell / C Shell X/OS Command Language User Guide*. Note that no mail is printed with this option.

-p causes all mail to be printed.

-q causes **mail** to terminate after interrupts. These usually have the effect of terminating only the current message.

-r causes messages to printed in a first-in, first-out sequence.

-f*file* causes **mail** to use the named *file* rather than the default *mbox* file to store the message after reading.

The second form of **mail** is used to send messages, and has the following options and arguments:

-t causes the message to be preceded by the names of all the recipients of the message. The name of a user is taken from the list of recognised log in names.

persons a list of one or more users who are to receive the message.

If used without arguments, **mail** prints the user's mail, message by message in last-in, first out sequence. Each message is preceded by a prompt, and the user can respond with one of the following options:

newline go on to the next message

+ same as **newline**

d delete the current message and go on to the next message

- go back to the previous message

s [*file*] save the message in the named *file* (one or more filenames may be specified) instead of the default, *mbox*

w [*file*] save the message, without its header, in the named *file* (one or more filenames may be specified) instead of the default, *mbox*

m [*persons*] mail the message to the named *persons*. The current user is the default

q store all undeleted messages in the *mbox* or other specified file, and quit **mail**

CTRL-d acts as the end-of-text marker, and has the same effect as **q**

x store all messages in the *mbox* or other specified file, and stop

!*cmd*	escape to the shell, and execute command *cmd*
*	print a command summary

This section presents the **mail** command. It discusses the basics of sending mail to one or more people simultaneously, whether they are working on the local system (the same system as you) or on a remote system. It also covers receiving and handling incoming mail.

SENDING MESSAGES

The basic command line format for sending mail is

```
mail login
```

where *login* is the recipient's login name on a X/OS system. This login name can be either of the following:

- a login name if the recipient is on your system (for example, *odin*)
- a system name and login name if the recipient is on another X/OS system that can communicate with yours (for example, *asgard!odin*) where the system name is *asgard*

For the moment, assume that the recipient is on the local system. (We will deal with sending mail to users on remote systems later.) Type the **mail** command at the system prompt, press the **CR** key, and start typing the text of your message on the next line. There is no limit to the length of your message. When you have finished typing it, send the message by typing a period or full stop (.) or a **CTRL-d** (hold down the **CTRL** key and press the **d** key) at the beginning of a new line.

The following example shows how this procedure will appear on your screen. Remember that throughout this manual, the > symbol in bold face represents the system prompt, and that the characters in bold represent a key on the keyboard. Thus, **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>mail loki CR
Dear Loki CR
CR
Please stop playing those silly CR
tricks on Thor. You know what CR
he's like when he gets paranoid. CR
CR
Odin CR
. CR

>
```

The prompt on the last line means that your message has been queued (placed in a waiting line of messages) and will be sent.

UNDELIVERABLE MAIL

If you make an error when typing the recipient's login, the **mail** command will not be able to deliver your mail. Instead, it will print two messages telling you that it has failed and that it is saving your mail. Then it will save your mail in a file called *dead.letter*.

For example, say you (owner of the login *loki*) want to send a message to a user with the login *odin* on a system called *asgard*. Your message is as follows:

```
The meeting with the Ice Giants has been changed to 2:00.
Hope the eye-drops helped.
```

Failing to notice that you have incorrectly typed the login as **odn**, you try to send your message.

```
>mail odn CR
The meeting with the Ice Giants has been changed to 2:00.
Hope the eye-drops helped.
. CR
mail: Can't send to odn
Mail saved in dead.letter

>
```

SENDING MAIL TO ONE PERSON

The following screen shows a typical message.

```
>mail thor CR
Dear Thor, CR
The weather forecast indicates thunder and tempests CR
for this afternoon. Could you delay them a couple of CR
hours? I have to do a spot of gardening. The roses CR
are looking terrible. Thanks, CR
Odin CR
. CR

>
```

When *thor* logs in at his terminal (or while he is already logged in), he receives a message that tells him he has mail waiting:

```
you have mail
```

To find out how he can read his mail, see the section called *Managing Incoming Mail*, later in this chapter.

You can practice using the **mail** command by sending mail to yourself. Type in the **mail** command and your login ID, and then write a short message to yourself. When you type the final period or **CTRL-d**, the mail will be sent to a file named after your login ID in the */usr/mail* directory, and you will receive a notice that you have mail.

Sending mail to yourself can also serve as a handy reminder system. For example, suppose you (login ID *heimdall*) want to call someone the next morning. Send yourself a reminder in a mail message.

```
>mail heimdall CR
Call the opticians and find out CR
if the new contact lenses are ready CR
. CR

>
```

When you log in the next day, a notice will appear on your screen informing you that you have mail waiting to be read.

SENDING MAIL TO SEVERAL PEOPLE SIMULTANEOUSLY

You can send a message to a number of people by including their login names on the **mail** command line. For example:

```
>mail loki thor heimdall CR
Dear Gods! CR
Office party now on for Friday, 8.00 P.M. CR
Bring your own swords. CR
Roast boar and horns of mead provided. CR
Be there or be square. CR
Odin CR
. CR

>
```

SENDING MAIL TO REMOTE SYSTEMS

Until now we have assumed that you are sending messages to users on the local X/OS system. However, your company may have three separate computer systems, each in a different part of a building, or you may have offices in several locations, each with its own system.

You can send mail to users on other systems simply by adding the name of the recipient's system before the login ID on the command line, for example:

```
mail asgard!thor CR
```

Notice that the system name and the recipient's login ID are separated by an exclamation mark (!).

Before you can run this command, however, you need three pieces of information:

- the name of the remote system

- whether or not your system and the remote system communicate

- the recipient's login name

The **uname** and **uuname** commands allow you to find this information. The **uname** utility tells you the name of your current system, while **uuname** tells you the names of all systems connected to your own system. They have the following syntax:

```
uname [-snrvma]
```

The options to **uname** are as follows:

-s prints the system name

-n prints the *nodename* of the system, where a nodename is the identifier of the system to a communications network

-r prints the release number of the operating system

-v prints the version number of the operating system

-m prints the machine hardware name

-a prints all the above information

```
uuname [-lv]
```

Used without arguments, **uuname** gives a list of names for the systems linked to the current system. The options to **uuname** are as follows:

-l prints the name of the current system. Note that is the same as using **uname**.

-v prints additional information about each system connected to the current system

In order to run send mail to remote users, you must get the name of the remote system and the recipient's login name from the recipient. If the recipient does not know the system name, have him or her issue the following command on the remote system:

```
uname -n
```

The command will respond with the name of the system. For example:

```
>uname CR
asgard

>
```

Once you know the remote system name, the **uuname** command can help you verify that your system can communicate with the remote system. At the prompt, type:

```
uuname
```

This generates a list of remote systems with which your system can communicate. If the recipient's system is on that list, you can send messages to it by **mail**.

You can simplify this step by using the **grep** command to search through the **uuname** output. At the prompt, type:

```
uuname | grep asgard
```

If **grep** finds the specified system name, it prints it on the screen. For example:

```
>uuname | grep asgard CR
asgard

>
```

This means that *asgard* can communicate with your system. If *asgard* does not communicate with your system, **uuname** returns a prompt.

```
>uuname | grep asgard CR

>
```

To summarize our discussion of **uname** and **uuname**, consider an example. Suppose you want to send a message to login *loki* on the remote system *asgard*. Verify that *asgard* can communicate with your system and send your message. The following screen shows both steps.

```
>uuname | grep asgard CR
asgard

>mail asgard!loki CR
Dear Loki, CR
The King of the Ice Giants wants his bicycle back. CR
Apparently, he's going on holiday on Tuesday, CR
and if it's not back by then, there'll be trouble. CR
.

>
```

MANAGING INCOMING MAIL

As stated earlier, the **mail** command also allows you to display messages sent to you by other users on your screen so you can read them. If you are logged in when someone sends you mail, the following message is printed on your screen:

```
you have mail
```

This means that one or more messages are being held for you in a file called */usr/mail/your_login*, usually referred to as your mailbox. To display these messages on your screen, type the **mail** command without any arguments:

```
mail
```

The messages will be displayed one at a time, beginning with the one most recently received. A typical **mail** message display looks like this:

```
>mail CR
From heimdall Wed May 21 15:33 CST 1986
Dear Odin,
My wage check has bounced again. If you think I'm
prepared to sit up here listening to
the grass grow at the other side of the Universe
for nothing, you're wrong! I want cash next time.
Heimdall.
?
```

The first line, called the header, provides information about the message: the login name of the sender and the date and time the message was sent. The lines after the header (up to the line containing the ?) comprise the text of the message.

If a long message is being displayed on your terminal screen, you may not be able to read it all at once. You can interrupt the printing by typing **CTRL-s**. This will freeze the screen, giving you a chance to read. When you are ready to continue, type **CTRL-q**, and the printing will resume.

After displaying each message, the **mail** command prints a ? prompt and waits for a response. You have many options, for example, you can leave the current message in your mailbox while you read the next message; you can delete the current message; or you can save the current message for future reference. For a list of **mail**'s available options, type a ? in response to **mail**'s ? prompt.

To display the next message without deleting the current message, press the **CR** key after the question mark.

```
? CR
```

The current message remains in your mailbox and the next message is displayed. If you have read all the messages in your mailbox, a prompt appears.

To delete a message, type a **d** after the question mark:

```
? d CR
```

The message is deleted from your mailbox. If there is another message waiting, it is then displayed. If not, a

prompt appears as a signal that you have finished reading your messages.

To save a message for later reference, type an **s** after the question mark:

```
? s CR
```

This saves the message, by default, in a file called *mbox* in your home directory. To save the message in another file, type the name of that file after the **s** command.

For example, to save a message in a file called *mailsave* (in your current directory), enter the response shown after the question mark:

```
? s mailsave CR
```

If *mailsave* is an existing file, the **mail** command appends the message to it. If there is no file by that name, the **mail** command creates one and stores your message in it. You can later verify the existence of the new file by using the **ls** command. (The **ls** command lists the contents of your current directory, and is described in its own tutorial in this manual.)

You can also save the message in a file in a different directory by specifying a path name. For example:

```
? s project1/memo CR
```

This is a relative path name that identifies a file called *memo* (where your message will be saved) in a subdirectory (*project1*) of your current directory. You can use either relative or full path names when saving

mail messages. (For instructions on using path names, see the Introduction to this manual.)

To quit reading messages, enter the response shown after the question mark:

```
? q CR
```

Any messages that you have not read are kept in your mailbox until the next time you use the **mail** command.

To stop the printing of a message entirely, press the **BREAK** key. The **mail** command will stop the display, print a ? prompt, and wait for a response from you.

THE PROCESS HANDLING COMMANDS

INTRODUCTION

This chapter of the manual contains four tutorials, covering the basic X/OS commands for monitoring and controlling the behaviour of proceeses active on the system. The utilities covered are as follows:

KILL terminates a process

NOHUP runs a command immune from hangup and quit signals

PS reports on process status

UMASK sets the file-creation mode

The tutorials are presented in alphabetical order.

KILL: terminate a process

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **kill** utility which terminates the process represented by a specified process number. The various job-control facilities provided by X/OS are described in the shell tutorials of the *Shell / C Shell X/OS Command Language User Guide*. The type of signal sent to the process can be varied in order to produce different effects.

SYNTAX

```
kill [-signo] PID ...
```

DESCRIPTION

Used without the *-signo* parameter, **kill** sends signal **15** (terminate) to the process identified by *PID*. This has the effect of killing processes that do not have built-in terminate signal handling systems.

A full list of the signals available to **kill** is given in the *signal*(2) entry in the *System Interfaces and Libraries Reference Manual*. The following is a list of commonly used signals:

SIGHUP 01 hangup

SIGINT 02 interrupt

SIGQUIT 03 quit

SIGKILL 09 kill (this signal cannot be caught or ignored)

SIGTERM 15 terminate (default)

SIGUSR1 16 user-defined signal 1

SIGUSR2 17 user-defined signal 2

The *-signo* parameter takes the form of the two-digit identification number given in the above list. The *PID* parameter takes the form of the *process ID number*. Two methods are available for finding this number. When a process is started in the *background* (that is, when the command line is followed by an ampersand **&**), the shell returns the process ID of the process created. Details of background jobs are given in the shell tutorials in the *Shell / C Shell X/OS Command Language User Guide*.

The second means of determining a process ID involves using the **ps** command, which has its own tutorial in this manual. The next section gives an example of each of these methods.

EXAMPLES

In the first example, the **ls -R** command is used to list all the files on the directory system and direct the output into a file called *listing.doc*. Because this may be a time-consuming procedure, the job is started in the background, using the ampersand character. The shell returns a job number and a process ID. Remember that **>** in bold type represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

Note that these examples suppose that the C Shell is being used. The **jobs** command does not exist under the Bourne Shell. Instead, the more general **ps** command can be used. This is illustrated in the second example, below.

```
>ls -R > listing.doc & CR
[1] 23564

>kill 23564 CR

>jobs CR
[1] terminated  ls -R > listing.doc

>
```

In the above example, **[1]** is the job number and **23564** is the process ID. The second command line terminated the **ls -R** command by directing the terminate signal at process 23564. The third command line, **jobs**, confirmed that the correct command was terminated.

In the next example, the **ls -R** command is repeated. The **ps** command is then used to confirm the process ID. Note that **ps** includes itself in the list of active processes. Finally, the **SIGKILL** signal (9) is used to kill the process. Note that this signal is a sure way of terminating a job, even if it has built-in signal handling systems. This is often known as the *terminate with extreme prejudice* signal.

```
>ls -R > listing.doc & CR
[2]  23567

>ps CR
  PID  TTY      TIME  COMMAND
 3398  tty35    1:26  sh
 3567  tty35    0:01  ls
 3569  tty35    0:01  ps

>kill -9 3567 CR
```

```
>ps CR
  PID  TTY      TIME  COMMAND
 3398  tty35    1:26  sh
 3573  tty35    0:00  ps
 [2] + Killed                ls -R > listing.doc

>
```

Note that a normal user cannot kill a process belonging to another user.

NOHUP: runs a command immune to hangup and quit signals

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **nohup** utility, which tells X/OS to execute a specified command while ignoring hangup and quit signals.

SYNTAX

```
nohup command [args]
```

DESCRIPTION

The *command* argument takes the form of a normal shell command line, where the specified command has access to its full range of options and arguments. Note that where pipes or lists of commands are to be used, these must be placed in a list file. This list file is then specified as an argument to **nohup**, in the form of a shell script. This means that the list file must be executable. An example of this system is given below.

Commands running under **nohup** are immune to hangups and quit signals. Once activated, the **BREAK** key is used to interrupt such a command. By running a **nohup** process in the background (using the **&** character), the specified command will continue to run after the user has logged off from the system. Background **nohup** processes can be terminated using the **kill** command, which has its own tutorial in this manual.

By default, **nohup** places output from the specified command or commands in a file called *nohup.out*, placed in

the current directory. This assumes that files can be written to the current directory. If this access permission is not available, *nohup.out* is placed in a specially created directory called *$HOME*. However, it is possible to redirect this output to any desired file.

EXAMPLES

In the first example, **nohup** is used to run the **spell -v** command on a range of files, all ending with the filename extension *.txt*. (Details of this utility can be found in the **spell** tutorial in this manual.) Because no output filename was specified, **nohup** directs output from **spell** into a file called *nohup.out*. This is confirmed with a status message on the next line, where the current directory is called *spike/project1*, and the correct access permissions are set. The example ends with the **ls** command, which confirms that **nohup** and **spell** have executed correctly.

Remember that in these examples, the **>** symbol in bold type represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>nohup spell -v *.txt CR
Sending output to nohup.out

>ls CR
chap1.txt
chap2.txt
chap3.txt
chap4.txt
nohup.out

>
```

In the second example, **nohup** is used to repeat the **spell** operation, this time as a background process, using the **&** character. Note that the output is redirected to a file called *faults*. The two numbers returned are the job number and the process ID. While this long-winded process is executing, the user logs off the system, waits a while, then logs back on. After using the **cd** command to change position to *spike/project1*, **ls** is used to check that the operation was a success.

```
>nohup spell -v *.txt > faults & CR
4963

>logout CR

login: spike CR

>cd spike/project1 CR

>ls CR
chap1.txt
chap2.txt
chap3.txt
chap4.txt
faults

>
```

In the next example, a sequence of commands linked by a pipeline is to be run under **nohup**. In order to do this, it is necessary to run these commands from an executable command-list file. The example begins by using the **cat** utility (described in its own tutorial in this manual) to display the contents of the file, which is called *cmds.lst*. It goes on to assign execute permission to this file, using **chmod**, also described in this manual, then runs this command sequence under **nohup**.

```
>cat cmnds.lst CR
spell *.txt | sort > listfile

>chmod u+x cmnds.lst CR

>nohup cmnds.lst CR
Sending output to nohup.out

>
```

PS: report process status

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **ps** utility, which reports on the status of any processes currently active on the system.

SYNTAX

```
ps [options]
```

DESCRIPTION

Used without options, **ps** prints information about the currently active processes on the current terminal. Output appears under the following headings:

```
PID  TTY  TIME COMMAND
```

These have the following meanings:

PID the process identification number. This is assigned automatically by the system. Where several processes are grouped into a single job, for example with the pipe system, each process is given its own ID. Each **PID** is unique.

TTY the terminal identification. A terminal may be running more than one active process.

TIME the cumulative execution time of the process.

COMMAND the command name. For a listing of all the options and arguments given to the command, the **-f** option to **ps** can be used.

The options allow the user to specify a range of processes across the whole system. A partial list of these options is as follows:

-e lists information about all processes, across all terminals.

-d lists all processes, across all terminals, except for group processes

-a lists all processes, across all terminals, except for group processes, and processes not directly associated with a terminal.

-f and

-l these two options print a much greater range of information for each active process. Details of the output can be found in the *ps*(1) entry of the *Utilities Reference Manual*.

It is possible to create files containing lists of terminals and process identifiers. These can be used to restrict the **ps** output to the range of terminals and/or processes listed. These, and other, options are listed in the *ps*(1) entry in the *Utilities Reference manual*.

EXAMPLES

In the following examples, the currently active processes are listed, first for the user's own terminal, then for a range of terminals. It is assumed that the computer system supports three terminals. Remember that the > symbol in bold type represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>ps CR
   PID TTY      TIME COMMAND
 14972 tty73   0:00 ps

>
```

This first example used **ps** in its default form. The user's terminal is identified by the name **tty73**, and is running a single process, **ps**, identified by the process ID **14972**. It's cumulative execution time is negligable.

The next example uses a more complex form of **ps** to check on the active processes owned by the whole range of terminals supported by the system.

```
>ps -a CR
   PID TTY      TIME COMMAND
 13877 tty73   2:05 sh
 14972 tty73   0:00 ps
 13397 tty75   1:13 sh
 13778 tty75   1:02 vi chapl.txt
 14653 tty77   0:12 sh
 14788 tty77   0:00 ps -al

>
```

UMASK: set file-creation mode

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **umask** utility, which establishes the access permissions that will be given to newly created files and directories. The **chmod** tutorial in this manual explains these access permissions in detail.

SYNTAX

---

umask [xxx]

DESCRIPTION

Whenever a file or directory is created, it is given a set of access permissions that define whether the different classes of user (*owner*, *group* members, and *other* users) can *read*, *write* and *execute* it.

The system's default permissions setting depends on the value passed to the **creat** routine. (Details of this routine are given in the *creat*(2) entry of the *System Interfaces and Libraries Reference Manual.*) The **umask** utility allows individual users to alter this default setting.

When typed on its own, **umask** displays its current value. The *xxx* parameter allows the user to set the value of **umask**. It takes the form of a three-digit octal number. Each digit in the number is subtracted from its corresponding digit in the three-digit octal **creat** value. The resulting number represents the permissions that will be applied to new files and directories created by the

user.

Note that, where this **umask** value is required to be a permanent setting, it can be added to the user's *.profile* file, which is located in the user's home directory.

EXAMPLES

The example below assumes that the **creat** mode gives read, write and execute permissions to all classes of user. This means that new files are created using mode **777**, and new directories are created using **666**. These values are the sum of the various codes listed in the *chmod*(1) entry of the *Utilities Reference Manual*.

The first command line sets **umask** to **022**. This has the effect of setting the file creation mode to **755** and the directory creation mode to **644**. Remember that the > character in bold type represents the system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>umask 022 CR

>umask CR
0022

>
```

The second command line checked the setting of **umask**. The resulting file creation modes of **755** for files and **644** for directories has the effect of allowing read, write and execute permission for the owner, and read and execute permissions only for all other users.

INTRODUCTION

This last chapter presents a tutorial on one of the many X/OS programming tools. **M4** is a macro processor for the C and Rational FORTRAN (Ratfor) programming languages.

M4: C and RATFOR macro processor

GENERAL

The **M4** macro processor is a front end for the Rational Fortran (Ratfor) and the C programming languages. The **#define** statement in C language and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor.

At the beginning of a program, a symbolic name or symbolic constant can be defined as a particular string of characters. The compiler will then replace later unquoted occurrences of the symbolic name with the corresponding string. Besides the straightforward replacement of one string of text by another, the **M4** macro processor provides the following features:

- arguments
- arithmetic capabilities
- file manipulation
- conditional macro expansion
- string and substring functions

The basic operation of **M4** is to read every alphanumeric token (string of letters and digits) input and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The user also has the capability to define new macros. Built-ins and user-defined macros work exactly the same way except that some of the built-in macros have side-effects on the state of the process. A list of 21 built-in macros provided by the **M4** macro processor, together with their functions, is shown below:

changequote Restores original characters or makes new quote characters the left and right brackets.

changescom Changes left and right comment markers from the default **#** and newline.

deer Returns the value of its argument decremented by 1.

define Defines new macros.

defn Returns the quoted definition of its argument(s).

divert Diverts output to 1-out-of-10 diversions

divnum Returns the number of the currently active diversion.

dnl reads and discards characters up to and including the next new line.

dumpdef Dumps the current names and definitions of items names as arguments.

errprint Prints its arguments on the standard error file.

eval Prints arbitrary arithmetic on integers.

ifdef Determines if a macro is currently active.

ifelse	Performs arbitrary conditional testing.
include	Returns the contents of the file named in the argument. A fatal error occurs if the file name cannot be accessed.
incr	Returns the value of its arguments incremented by 1.
index	Returns the position where the second argument begins in the first argument pf index.
len	Returns the number of characters that make its argument.
m4 exit	causes immediate exit from **M4**.
m4wrap	Pushes the exit code back at final EOF.
maketemp	Facilitates making unique file names.
popdef	Removes current definition of its argument(s). exposing any previous definitions.
pushdef	Defines new macros but saves any previous definitions.
shift	Returns all arguments of shift except the first argument.
sinclude	Returns the contents of the file named in the arguments. The macro remains silent and and continues if the file is inaccessible.
substr	Produces substrings of strings.
syscmd	Executes the System command given in the first argument.

traceoff Turns macro trace off.

traceon Turns the macro trace on.

translit Performs character translation

undefine Removes user-defined or built-in macro definitions.

undivert Discards the diverted text.

To use the **M4** macro processor input the following command:

```
m4 [optional files]
```

Each argument is processed in order. If there are no arguments or if an argument is "-", the standard input is read at that point. The processed text is written on the standard output which may be captured for subsequent processing with the following input:

```
m4 [files] > output file
```

DEFINING MACROS

The primary built-in function of **M4** is **define**. **Define** is used to define new macros. The following input:

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter

(the underscore counts as a letter). *Stuff* is any text. Use of slashes may extend *stuff* over multiple lines. Thus, as a typical example:

```
define(N, 100)
...
if (i > N)
```

defines *N* as 100 and uses the symbolic constant *N* in a later **if** statement.

The left parenthesis must immediately follow the word **define** to signal that **define** has arguments. If a macro is not immediately followed by "(" it is assumed to have no arguments. Macro calls have the following general form:

name(arg1,arg2, ... argn)

A macro name is only recognised if it appears surrounded by non-alphanumerics. Using the following example:

```
define(N, 100)
...
if (NNN > 100
```

the variable *NNN* is absolutely unrelated to the defined macro, *N*, even though the variable contains several *N*s.

Macros may be defined in terms of other names. For example:

```
define(N, 100)
define(M, N)
```

defines both *M* and *N* as 100. If *N* is redefined to a different value *M* still retains the value of 100 unless and until it is redefined to a different value.

The **M4** macro processor expands macro names into their defining text as soon as possible. The string *N* is immediately replaced by 100. Then the string *M* is replaced by 100. The end result is the same as the following:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
...
if (i > M)
```

Now *M* is defined as the string *N*, so when the variable *M* is used later, it will be replaced by the value of *N* at that time (100, assuming neither variable has been redefined).

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by left and right single quotes is not expanded immediately but has the quotes stripped off. The value of a quoted string is the string with the quotes stripped off. In the input:

```
define(N, 100)
define(M,'N')
```

the quotes round the *N* are stripped off as the argument is collected. The result of using quotes is to define *M* as the string *N*, not 100 (see previous example). The general rule is that **M4** always strips off one level of single quotes when it evaluates anything. This is true even outside macros. If the word **define** is to appear in the output, the word must be quoted in the input as follows:

```
'define' -l;
```

Another example of using quotes is redefining *N*. To redefine *N*, the evaluation must be delayed, by quoting:

```
define(N, 100)
...
define('N', 200)
```

In **M4** it is often wise to quote variables. The following example will fail:

```
define(N, 100)
...
define(N, 200)
```

The *N* in the second definition is replaced by 100, and becomes equivalent to:

```
define(100, 200)
```

This statement is ignored by **M4**, since only variables which look like names can be defined.

If left and right single quotes are not convenient for some reason, the quote characters can be changed, using the built-in **changequote** macro, as shown in the following example:

```
changequote([, ])
```

The built-in macro **changequote** makes the left and right square brackets the new quote characters. The original, default quote characters can be restored, using the **changequote** macro without arguments, as follows:

```
changequote
```

There are two more built-in macros related to **define**. The **undefine** macro removes the definition of any macro, as follows:

```
undefine('N')
```

This removes the definition of *N*. Built-in macros can be removed using **undefine**, as follows:

```
undefine('define')
```

Note that a definition cannot be reused after it has been

removed.

The built-in macro **ifdef** provides a way to determine if a named macro is currently defined. Depending on the system, a definition appropriate to the particular machine can be made, as follows:

```
ifdef('pdp11', 'define(wordsize, 16)')
ifdef('u3b', 'define(wordsize, 32)')
```

The **ifdef** macro actually permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not defined, the value of **ifdef** is the third argument. If there is no third argument the value of **ifdef** is null. If the name is undefined, the value of **ifdef** is then the third argument, as in:

```
ifdef('x/os', on X/OS, not on X/OS)
```

ARGUMENTS

So far, only the simplest form of macro processing has been discussed, the replacement of one string by another, fixed, string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of $*n* is replaced by the *n*the argument when the macro is actually expanded. Thus, the macro, **bump**, defined as:

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1. The

bump(x) statement is equivalent to x = x + 1.

A macro may have as many arguments as needed, but only the first nine are accessible (**$1** through **$9**). The macro name is **$0**, although it is not commonly used. Arguments that are not supplied are replaced by null strings, so a macro can be defined which simply concatenates its arguments, as follows:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus, **cat(x, y, z)** is equivalent to xyz. Arguments **$4** through **$9** are null, since no corresponding arguments are provided. Leading unquoted blanks or tabs, or newlines, which occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a, b  c)
```

defines **a** as **b c**.

Arguments are separated by commas; however when commas are within parenthesis, the argument is neither terminated nor replaced. For example:

```
define(a, (b,c))
```

has only two arguments. The first argument is **a**. The second argument is literally **(b,c)**. A single comma or parenthesis can be inserted by quoting it.

ARITHMETIC BUILT-IN MACROS

M4 provides three built-in macros for doing arithmetic (on integers only). The simplest are **incr** and **decr** which **increment** and **decrement**, respectively, their numeric arguments by 1. Thus, to handle the common programming situation where a variable is to be defined as being one more than *N*xa, the following can be used:

```
define(N, 100)
define(N1, 'incr(N)')
```

This defines *N1* as one more than the current value of *N*.

The more general mechanism for arithmetic is the built-in macro, **eval** which can perform arbitrary arithmetic on integers. The operators, in decreasing order of precedence, are:

+ -	unary plus and minus
** or ^	exponentiation
* / %	multiply, divide and modulus
+ -	binary plus and minus
== != < <= > >=	equal to, not equal to, less than, less than or equal to, greater than and greater than or equal to
!	not
& or &&	logical AND
| or ||	logical OR

Parenthesis may be used to group operations, where needed. All the operands of an expression given to eval

must ultimately be numeric. The numeric value of a *true* relation (for instance 1>0) is 1 and that of a *false* relation (for instance 0>1) is 0. The precision of **eval** is 32 bits under the LSX machine operating system.

As a simple example, *M* can be defined to be **2==N+1** using **eval**, as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

The defining text for a macro should be quoted, unless the text is very simple. Quoting the defining text usually gives the required result and is a good habit to get into.

FILE MANIPULATION

A new file can be included in the input at any time using the built-in function, **include**. For example:

```
include(filename)
```

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (**include**'s replacement text) is the contents of the file. If needed, the contents can be captured in definitions, etc.

A fatal error occurs if the file named in an **include** cannot be accessed. To get some control over this situation, the alternate form, **sinclude**, can be used. The built-in, **sinclude**, (silent include) says nothing and continues if the named file cannot be accessed.

The output of **M4** can be diverted to temporary files during processing, and the collected material output on command. **M4** maintains nine of these diversions, numbered 1 through 9. If the built-in macro

```
divert(n)
```

is used, all subsequent output is appended to a temporary file referred to as *n*. Diversion to this file is stopped by the **divert** or **divert(0)** command which resumes the normal output process.

All diverted text is normally output at the end of processing with the diversions output in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The built-in **undivert** brings back all diversions in numerical order. The built-in **undivert** with arguments brings back the selected diversions in the order specified. The act of undiverting discards the diverted text (as does diverting) into a diversion whose number is not between 0 and 9, inclusive.

The value of **undivert** is *not* the diverted text. Furthermore, the diverted material is *not* rescanned for macros. The built-in **divnum** returns the number of the currently active diversion. The current output stream is zero during normal processing.

SYSTEM COMMAND

Any program in the local operating system can be run by using the **syscmd** built-in macro. For example,

```
syscmd(date)
```

on the X/OS operating system runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**. To facilitate creating unique file names, the built-in **maketemp** macro is provided with specifications identical to the system function, *mktemp*. The **maketemp** macro fills in a string of **XXXXX** in the argument with the process id of the current process.

CONDITIONALS

Arbitrary conditional testing is performed via built-in **ifelse**. In the simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, **ifelse** returns the string *c*. Otherwise, string, *d* is returned. Thus, a macro called **compare** can be defined as one which compares two strings and returns *yes* or *no* depending on whether they are the same or different, respectively, as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes, which prevent evaluation of **ifelse** occurring too early. if the fourth argument is missing, it is treated as empty.

The built-in **ifelse** macro can actually have any number of arguments and provides a limited form of multi-way decision capability. In the input

```
ifelse(a, b, c, fd, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* matches *e* the result is *f*. Otherwise the result is *g*. If the final argument is omitted, the result is null, so in

```
ifelse(a, b, c)
```

the result is *c* if *a* matches *b*. Otherwise the result is null.

STRING MANIPULATION

The built-in **len** macro returns the length of the string (number of characters) that make up its argument. Thus, the result of

```
len(abcdef)
```

is 6, and the result of

```
len((a,b))
```

is 5.

The built-in **substr** macro can be used to produce substrings of strings. Using the input

```
substr(s, i, n)
```

returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. Inputting

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time
```

if *i* or *n* are out of range, various actions occur.

The built-in **index(s1, s2)** macro returns the index (position) in *s1* where *s2* occurs, or -1 if it does not occur. As with **substr** the origin for strings is 0.

The built-in **translit** macro performs character translation and has the general form

```
translit(s, f, t)
```

which modifies *s* by replacing any character found in *f* by the corresponding character in *t*. Using the input

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f* characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at

all, characters from *f* are deleted from*s*. So

```
translit(s, aeiou)
```

would delete all vowels from *s*.

There is also a built-in macro called **dnl** that deletes all characters that follow it up to and including the next newline. The **dnl** macro is useful mainly for throwing away empty lines that otherwise tend to clutter up **M4** output. Using the input

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new line at the end of each line that is not part of the definition. So, the new line is copied into the output where it may not be wanted. If the built-in **dnl** macro is added to each of these lines, the newlines will disappear. Another means of achieving the same results is to output

```
divert(-1)
define(...)
...
divert
```

PRINTING

The built-in **errprint** macro writes its arguments out on the standard error file. An example would be

```
errprint('fatal error')
```

The built-in **dumpdef** macro is a debugging aid that dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Do not forget to quote the names.

$

$ parameter indicator 3-65

&

& running a background process 10-2, 10-6, 10-7

'

'' removing meaning of special characters 4-13

*****

* matching any characters 4-13

.

. current directory indicator 2-29, 2-45
.. parent directory indicator 2-10, 2-23, 2-29, 2-39, 2-45

/

/ root directory indicator 2-2, 2-36
/etc/magic file 3-18

>

> redirecting output 2-6
>> redirecting and appending output 2-6

****

\ removing meaning of special characters 4-13

A

access permissions 2-12, 2-32, 10-13
ar command 6-2
ar.out file 6-2
archive and library system 6-2
arithmetic built-in macros, m4 11-11
arithmetic operators, m4, macros 11-11
ASCII codes 3-3, 3-39, 3-40, 3-67
awk
  actions 7-44
  arrays 7-51, 7-58, 7-64
  assignment expressions 7-19
  assignments 7-44
  BEGIN keyword 7-35
  binary terms 7-17
  command line input 7-25
  comments 7-9
  concatenated terms 7-19
  conditional statements 7-56
  END keyword 7-35

expressions 7-11, 7-19, 7-37, 7-38, 7-44
field separators 7-8, 7-9
field tokens 7-7
flow of control 7-56
for statement 7-58
functions 7-13, 7-54
grouped terms 7-18
grouping tokens 7-10
I/O operations 7-21
identifiers 7-5
if ... else statement 7-57
if statement 7-57
incremented variables 7-17
input files 7-3
input/output operations 7-21
keywords 7-5
lexical units 7-5
line scanning 7-3
looping 7-56
multiline records 7-9
numeric constants 7-5, 7-11
operators 7-6
output printing 7-27
output redirection 7-33
parameters 7-3
parenthesised terms 7-18
pattern combination 7-42
pattern ranges 7-43
patterns 7-42
piping output 7-34
primary expressions 7-11
printing 7-27
program execution 7-3, 7-63
program structure 7-3, 7-56
record separators 7-8, 7-9
record tokens 7-7
regular expressions 7-38
relational expressions 7-37
report generation 7-61
shell access 7-63
string constants 7-5, 7-11
strings 7-48
terms 7-17
tokens 7-5
unary terms 7-17
variables 7-12, 7-17, 7-44, 7-46, 7-49
while statement 7-57

B

background processes 10-2, 10-6, 10-7
banner command 4-2
built-in macros, m4 11-1
bytes 2-27

C

cal command 4-4
calendar command 4-6
calendar file 4-6
calendar system 4-4
cat command 2-5
cd command 2-9, 2-41
changequote macro, m4 11-7
changing directories 2-9
character translation 3-67
checksum and block count 8-7

child directories 2-23
chmod command 2-12, 3-65, 10-7
clear_screen variable 3-28
comm command 3-3
communications systems 9-1
compressing files 6-8
concatenating compressed files 6-12
concatenating files 2-5
copying files 2-17
cp command 2-17
creat routine 10-13
cron command 4-6
crontab command 4-7
crontab file 4-7
cu command 9-1
current working directory 2-2, 2-9, 2-36
cut command 3-7

D

date command 4-9
decr macro, m4 11-11
define macro, m4 11-1, 11-4
delayed macro expansion, m4 11-6
deleting directories 2-45
deleting files 2-42
df command 8-2
diff command 3-11
diff3 command 3-14
directories 2-2
disk usage 8-4
divert macro, m4 11-12
divnum macro, m4 11-12
dnl macro, m4 11-16
dot files 2-21, 2-26, 2-45
du command 8-4
dumpdef macro, m4 11-18

E

echo command 4-12
ed
  buffers 5-2
  changing text 5-36
  changing the current line 5-9
  command mode 5-4
  copying text 5-66
  creating text 5-4, 5-31
  current line 5-2
  deleting text 5-8, 5-40
  displaying text 5-6, 5-29
  filenames 5-4, 5-77
  help facilities 5-74
  input mode 5-4
  interrupts 5-80
  line addresses 5-15, 5-24
  moving text 5-64
  nonprinting characters 5-76
  problems 5-70
  quitting 5-11
  relative addresses 5-19
  reversing commands 5-41
  saving text 5-10
  search functions 5-25
  special characters 5-53
  starting up 5-4
  substitution functions 5-44
  symbolic addresses 5-17
  undoing commands 5-41
  using the shell 5-79
ed command 3-11, 3-12, 3-15, 3-30, 5-2, 7-71
editors 3-11, 3-12, 3-15,

3-30, 5-2, 5-96, 7-70
egrep command 7-6, 7-66
errprint macro, m4 11-18
eval macro, m4 11-11
executable files 10-7
execute permission 2-12
expanding compressed files 6-14
expressions 3-62

F

fgrep command 7-66
field delimiters 3-7, 3-40
file and directory listing 2-21
file command 3-18
file comparison 3-3, 3-11, 3-14, 3-72
file copying 2-17
file links 2-27
file paging 3-28, 3-35
file permissions 2-12, 2-32, 10-13
file protection modes 2-12, 2-32, 10-13
file size assessment 3-76
file systems 8-2
file types 3-18, 3-62
filename extensions 2-2
filenames 2-2
files 2-2

G

grep command 7-66, 9-10

H

hashcheck file 3-52
hashmake file 3-52
home directory 2-9, 2-36, 10-6

I

if statement 3-62
ifdef macro, m4 11-8
ifelse macro, m4 11-14
include macro, m4 11-12
incr macro, m4 11-11
index macro, m4 11-15
inodes 8-2

J

jobs command 10-3

K

kill command 10-2, 10-6

L

ld command 6-2
len macro, m4 11-15
lex command 7-6
library and archive system 6-2
line numbering 3-21
line printers 3-35
link editor 6-2
linked files 2-27
listing dot files 2-26,

2-45
logging in 2-9
logging on 2-36
logical pages 3-21
ls command 2-13, 2-17, 2-21, 2-43, 2-45

M

m4 command 11-1
m4, command execution 11-3
m4, macro names 11-1
m4, macros
  argument quoting 11-5
  arguments 11-1, 11-9
  arithmetic built-in 11-11
  arithmetic operators 11-11
  built-in macros 11-1
  changequote 11-7
  decr 11-11
  define 11-1, 11-4
  definition 11-4
  delayed expansion 11-6
  divert 11-12
  divnum 11-12
  dnl 11-16
  dumpdef 11-18
  errprint 11-18
  eval 11-11
  expansion 11-5
  format 11-4
  ifdef 11-8
  ifelse 11-14
  include 11-12
  incr 11-11
  index 11-15
  len 11-15
  maketemp 11-14
  sinclude 11-12
  substr 11-15
  syscmd 11-14
  translit 11-15
  undefine 11-7
  undivert 11-12
m4, output diversions 11-12
m4, output redirection 11-3
macro argument quoting, m4 11-5
macro arguments, m4 11-1, 11-9
macro definition, m4 11-1, 11-4
macro expansion, m4 11-5
macro format, m4 11-4
macro names, m4 11-1
macro processor 11-1
magic numbers 3-18
mail
  command summary 9-3
  incoming mail 9-3, 9-13
  local systems 9-5
  mailboxes 9-13
  mbox file 9-14
  message lengths 9-5
  quitting mail 9-15
  remote system names 9-10
  remote systems 9-9
  remote user names 9-10
  remote users 9-10
  saving messages 9-14
  screen handling 9-13
  sending mail 9-3, 9-5
  system names 9-9
  uname command 9-9
  user mailboxes 9-13
  user names 9-5
  uuname command 9-9
mailx command 9-1
maketemp macro, m4 11-14
making new directories 2-29

mbox file 9-14
metacharacters
  $ 3-65
  & 10-2, 10-6, 10-7
  '' 4-13
  * 4-13
  . 2-29, 2-45
  .. 2-10, 2-23, 2-29, 2-39, 2-45
  / 2-2, 2-36
  > 2-6
  >> 2-6
  \ 4-13
mkdir command 2-29
modification times 2-17, 2-21
mounted file systems 8-2
moving files 2-32
multi-user systems 2-2
mv command 2-32

N

nl command 3-21
nohup command 10-6
nohup.out file 10-6

O

output diversions, m4 11-12
output redirection, m4 11-3

P

pack command 6-8
parent directories 2-10, 2-23, 2-29
paste command 3-25
pathnames 2-2, 2-10, 2-23, 2-29, 2-36, 2-38
pattern scanning 7-2, 7-66
pattern searching 3-30
pcat command 6-8, 6-12
pg command 3-28
PID, process identification 10-2, 10-3, 10-10
pipes 3-59
pr command 3-35
printing files 2-5
printing packed files 6-12
process control
  creat routine 10-13
  job control 10-10
  jobs command 10-3
  kill command 10-6
  listing active processes 10-10
  nohup command 10-6
  process identification 10-2, 10-3, 10-10
  process status 10-10
  ps cmmand 10-4
  ps command 10-10
  SIGHUP signal 10-2
  SIGINT signal 10-2
  SIGKILL signal 10-2
  SIGQUIT signal 10-2
  SIGTERM signal 10-2
  SIGUSR1 signal 10-2
  SIGUSR2 signal 10-2
  terminating processes 10-2
  umask command 10-13
process identification 10-2, 10-3, 10-10
ps command 10-4, 10-10
pwd command 2-9, 2-36

R

read permission 2-12
reading incoming mail 9-3
redirection operators 2-4
regular expressions 3-30, 7-66, 7-71
reminder system 4-6
removing directories 2-45
removing files 2-42
renaming files 2-32
rm command 2-42
rmdir command 2-45
root directory 2-2

S

screen control 3-28, 3-35, 3-56
sed
  addresses 7-70, 7-74
  flow of control 7-83
  functions 7-75
  get functions 7-83
  hold functions 7-83
  hold spaces 7-70
  input/output functions 7-80
  instructions 7-70
  pattern spaces 7-70
  quitting 7-84
  regular expressions 7-71, 7-74
  scripts 7-70, 7-74
  substitute functions 7-78
  whole-line functions 7-76
selecting text fields 3-7
sending mail 9-3, 9-5
shell scripts 3-62
SIGHUP signal 10-2
SIGINT signal 10-2
SIGKILL signal 10-2
SIGQUIT signal 10-2
SIGTERM signal 10-2
SIGUSR1 signal 10-2
SIGUSR2 signal 10-2
sinclude macro, m4 11-12
sort command 3-4, 3-39
sort keys 3-40
spell command 3-46, 10-7
spellin file 3-52
spelling checker 3-46
split command 3-54
standard input 2-4
standard output 2-4
strings 4-2, 4-12
sub-directories 2-2, 2-21, 2-30
substr macro, m4 11-15
sum command 8-7
syscmd macro, m4 11-14

T

tail command 3-56
tee command 3-59
TERM variable 3-28
terminal identification 10-10
terminating processes 10-2
terminfo database 3-28
test command 3-62
tr command 3-67
translating characters 3-67
translit macro, m4 11-15
troff command 3-46
TTY, terminal identification 10-10

U

umask command 10-13
uname command 9-1, 9-9
undefine macro, m4 11-7
undivert macro, m4 11-12
uniq command 3-72
unpack command 6-8, 6-14
unpacks packed files 6-14
user disk space 2-2
uucp command 9-1
uuname command 9-1, 9-9
uustat command 9-1
uuto command 9-1
uux command 9-1

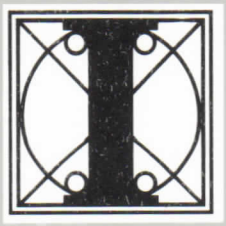
V

variables
  $ parameters 3-65
  clear_screen 3-28
  TERM 3-28
vi
  adding to the buffer 5-170
  append mode 5-100
  changing text 5-152, 5-154, 5-170
  character case 5-165
  clearing the window 5-164
  command mode 5-101
  copying text 5-159
  creating text 5-100, 5-107, 5-140
  cursor control 5-102, 5-112, 5-115, 5-131
  cutting and pasting 5-159
  deleting text 5-105, 5-145
  deleting the buffer 5-169
  displaying text 5-176
  filenames 5-99
  joining lines 5-164
  line editing 5-167
  line numbers 5-131, 5-168
  moving text 5-159
  moving the cursor 5-102, 5-112, 5-115, 5-131
  new files 5-167
  quitting 5-108, 5-173
  recovering lost text 5-176
  redrawing the window 5-164
  repeating commands 5-164
  reversing commands 5-146
  search functions 5-114, 5-132
  shell access 5-167
  special commands 5-164
  special options 5-176
  starting up 5-99
  substitute functions 5-152
  transposed characters 5-159
  undoing commands 5-146
  window control 5-122

W

wc command 3-76
while statement 3-62

Printed in Italy



olivetti

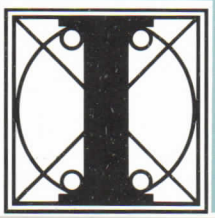
LSX Computer Line

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

1. INTRODUCTION

2. SH: default shell

2-1 INTRODUCTION

2-2 SHELL COMMAND LANGUAGE

2-3 THE SHELL'S METACHARACTERS

2-39 COMMAND LANGUAGE EXERCISES

2-40 SHELL PROGRAMMING

2-41 SHELL PROGRAMS

2-46 VARIABLES

2-53 ASSIGNING A VALUE TO A VARIABLE

2-59 SHELL PROGRAMMING CONSTRUCTS

2-81 DEBUGGING PROGRAMS

2-85 MODIFYING YOUR LOGIN ENVIRONMENT

2-86 SETTING TERMINAL OPTIONS

2-88 USING SHELL VARIABLES

2-91 SHELL PROGRAMMING EXERCISES

2-93 ANSWERS TO EXERCISES

2-93 ANSWERS TO COMMAND LANGUAGE EXERCISES

2-94 ANSWERS TO SHELL PROGRAMMING EXERCISES

3. CSH: alternative shell

3-1 INTRODUCTION

3-1 C SHELL COMMAND LANGUAGE

3-2 USING THE TERMINAL

3-2 Entering Commands

3-5 Command Flags

3-6 Combining Commands

3-7 Diagnostic Output

3-8 Standard Input and Standard Output

3-8 FILES AND DIRECTORIES

3-10 THE C SHELL'S METACHARACTERS

3-10 The C Shell's Directory Indicators

3-14 The C Shell's Re-direction Operators

3-19 The Pipe Metacharacter

3-20 Filename Substitution Metacharacters

3-24 Releasing Metacharacters

3-26 Summary of the Metacharacters

3-27 Directing Output to Existing Files

3-27 JOB CONTROL

3-28 Foreground Jobs

3-28 Background Jobs

3-31 Terminating Commands

3-32 Job Control Commands

3-34 **OUTPUT CONTROL**

3-34 The Screen Control Signals

3-34 The pg Command

3-35 **THE C SHELL VARIABLES**

3-35 Predefined and Environment Variables

3-37 .cshrc

3-38 .login

3-40 .logout

3-40 .history

3-41 Changing Variable Values

3-42 **THE HISTORY MECHANISM**

3-45 **ALIASES**

3-46 **THE DIRECTORY STACK**

3-51 **THE SECURITY SYSTEM**

3-53 **MORE BUILT-IN COMMANDS**

3-53 The prompt Command

3-54 The repeat Command

3-54 The time Command

3-55 Unsetting Aliases and Variable Definitions

3-55 PROGRAMMING THE C SHELL

3-56 Introduction

3-56 Invoking a Script and Using the argv Variable

3-59 Expressions

3-60 Control Structures

3-62 The Modifiers

3-64 A Sample C Shell Script

3-65 Executing a C Shell Script

INTRODUCTION

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	**-c***list*	*file*
	-f*list* [**-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 **Ivar1**.

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 -r*file*	true if the file exists and is readable
test -w*file*	true if the file exists and has write permission
test -x*file*	true if the file exists and is executable
test -s*file*	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 (*-n***b**) or characters (*-n***c**) 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 */project1/bin*. Edit your *.profile*, and add the entry *:/project1/bin* to the end of your **PATH**, as in the next example.

```
PATH=":/mylogin/bin:/bin:/usr/lib:/project1/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.

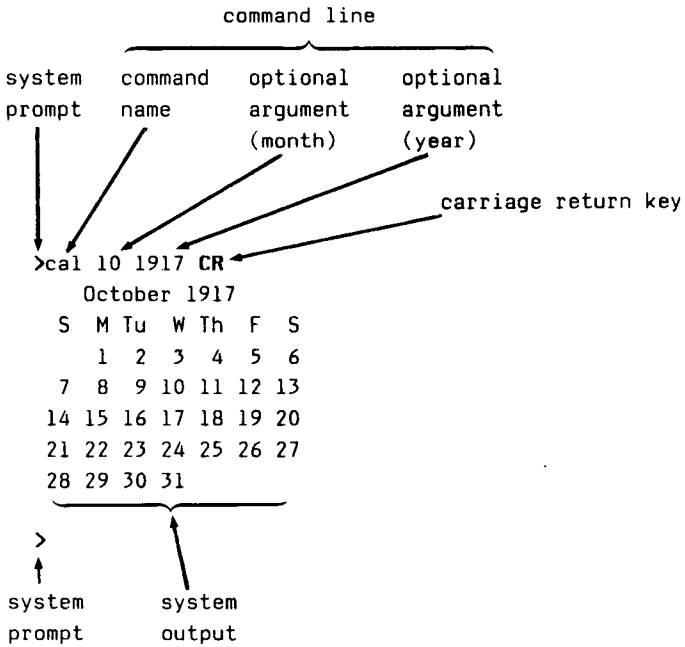
Once a command has been typed onto the screen, it must be entered, using the carriage return key. Once this has been pressed, the shell begins to process what has been typed.

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
chaptl.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 *chapt10.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 chapt10.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  hstory
 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

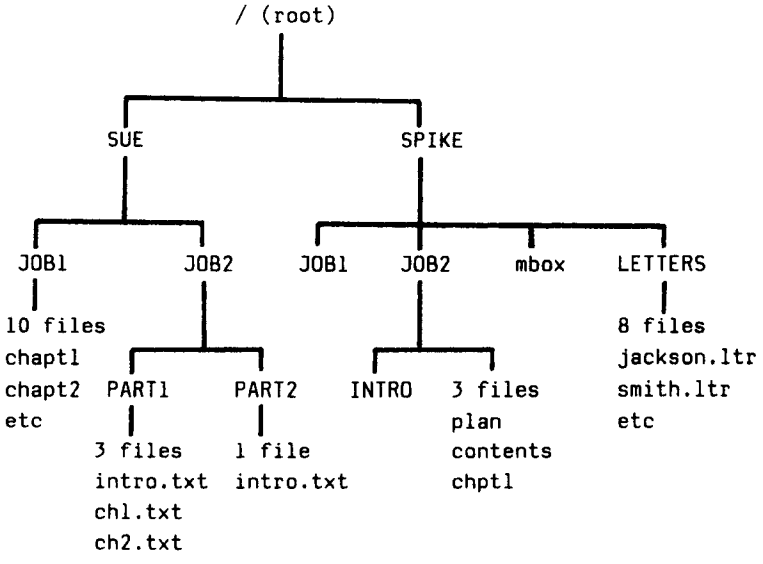
The directory system is created using the command **mkdir** (make directory), and users can move around the resulting system using **chdir**, optionally shortened to **cd** (change directory). Empty directories can be deleted with **rmdir** (remove directory), and the current location can be printed 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 12:07   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 *scriptl*, 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 *scriptl*:

```
>chmod u+x scriptl 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.

| redirecting output into another command 2-2, 2-22, 2-24, 3-19
|| Boolean OR expression 3-59

!

! prompt command number indicator 3-53
!! accesses last history command line 3-42
!= not equal to expression 3-59
!n accesses history command line n 3-42
!x accesses history command line beginning with string x 3-42
!~ special string matching expression 3-59

#

comment delimiter 2-59, 3-64
returns number of variables set 3-56

$

$ positional parameters 2-2, 2-46, 2-57
$ variable identifier 3-56
$# special parameters 2-48
$* special parameters 2-48, 2-65
$< reads terminal input into a shell script 3-57
$? special parameters 2-63
$status variable 3-59

%

% job control identifier 3-32

&

& background processing 2-2, 2-10, 2-21, 2-37, 3-28
&& Boolean AND expression 3-59

'

'' removing meaning of special characters 2-2, 2-13, 2-14, 3-24

(

( ) command grouping 3-28

)

) pattern statement delimiter 2-75

*

* matching any characters 2-2, 2-3, 2-7, 2-10, 2-75, 2-77, 3-20

-

- specifying a range of characters 2-10

.

. current directory indicator 3-11
.. parent directory indicator 3-10
.cshrc file 3-37
.history file 3-40
.login file 3-38
.logout file 3-40
.profile file 2-85

/

/ root directory indicator 3-11
/dev/null 2-69, 2-71

;

; running a sequence of commands 2-2, 2-12
;; conditional statement delimiter 2-75

<

< redirecting input 2-2, 2-17, 3-14
<< redirecting standard input 3-16

=

== equal to expression 3-59
=~ special string matching expression 3-59

>

> redirecting output 2-2, 2-17, 2-54, 3-14, 3-27
>! forces file clobbering 3-38
>& redirecting diagnostic output 3-15
>> redirecting and appending output 2-2, 2-18, 2-54, 3-15, 3-27

?

? matching any single character 2-2, 2-6, 2-7, 2-10, 2-77, 3-20
? returns whether a variable value is set 3-56

[

[ ] matching a sequence of

characters 2-2, 2-9, 2-10, 2-77, 3-20

\

\ removing meaning of special characters 2-2, 2-13, 3-24

`

`` substituting output of a command line 2-2, 2-28, 2-53, 2-56

A

absolute pathnames 3-12
alias command 3-45
alias mechanism 3-45
aliases 3-37
arguments 2-48, 3-2
argv variable 3-35, 3-48, 3-56
at command 2-29

B

background processes 2-10, 2-21, 3-28
banner command 2-15, 2-23, 2-28, 2-31
batch command 2-29
batch processing 2-29
bin directories 2-42
break command 2-79
breaksw statement 3-61
built-in commands 3-53

C

cal command 3-2, 3-28
case ... esac statement 2-75
case statement 3-61
cat command 2-19, 2-40, 3-8
cd command 2-12, 2-43, 2-88, 3-10, 3-38
CDPATH variable 2-51, 3-48
chmod command 2-41, 2-55, 3-51, 3-65
clear command 3-40
clobbering files 3-38
cmp command 3-64
combining commands 3-6
command flags 3-5
command lines 3-2
command names 3-2
command options 3-5
comments 2-59
conditional constructs
) notation 2-75
;; notation 2-75
breaksw statement 3-61
case ... esac statement 2-75
case statement 3-61
default statement 3-61
end statement 3-60
endif statement 3-60
endsw statement 3-61
fi statement 2-70, 2-72
foreach statement 3-60
goto statement 3-60
if ... then ... else statement 2-71, 3-60
if ... then statement

2-70
if statement 3-60
in statement 2-75
patterns 2-75
switch statement 3-61
while statement 3-60
continue command 2-79
core dumps 3-31
cp command 2-85, 3-20
csh command 3-56
cursors 3-2
cut command 2-24, 2-28
cwd variable 3-35, 3-46, 3-48

D

date command 2-24, 2-28
debugging 2-81
default statement 3-61
diagnostic output 3-7
directories 3-8, 3-10
directory stack mechanism
dirs command 3-47
handling directories 3-46
popd command 3-46
pushd command 3-46
dirs command 3-47
do statement 2-65, 2-68
done statement 2-65, 2-68
dot files
.cshrc file 3-37
.history 3-40
.login 3-38
.logout 3-40
.profile 2-85

E

echo command 2-3, 2-52, 2-63, 2-85, 3-24, 3-56
ed command 2-40, 2-61
editors 2-40, 2-61
end statement 3-60
endif statement 3-60
endsw statement 3-61
entering commands 3-2
env command 2-52
erase character 2-85
error messages 3-7
executable files 2-42
executing a shell program 2-41, 3-65
exit command 2-64
export command 2-78
expressions 3-59

F

fg command 3-32
fi statement 2-70, 2-72
file enquiry expressions 3-59
file security 3-51
filename substitution metacharacters 3-20
filenames 2-3, 2-43, 3-20
files 3-8
for statement 2-64
foreach statement 3-60
foreground jobs 3-28

G

goto statement 3-60
grep command 2-11, 2-13,

2-14, 2-21, 2-30, 2-34, 2-37

H

here document 2-32, 2-59
histchars variable 3-48
history command 3-42
history mechanism
  .history 3-40
  accessing past commands 3-42
  changing the metanotation 3-48
  history command 3-42
  history lists 3-42
  history variable 3-37
  metacharacters 3-42
  modifiers 3-43
  savehist variable 3-37
history variable 3-37, 3-48
home directory 3-12
HOME variable 2-51, 2-74, 2-88, 3-35, 3-38, 3-48

I

if ... then ... else statement 2-71, 3-60
if ... then statement 2-70
if statement 3-60
IFS variable 2-51
ignoreeof variable 3-38, 3-48
in statement 2-65, 2-75
input redirection 2-16, 2-17, 3-14
interrupt signal 3-31

J

job control (see: process control)
job numbers 2-30, 3-28, 3-29
job queueing 2-29
jobs command 3-32

K

kernel 3-1
kill character 2-85
kill command 2-35, 3-32

L

logging in 2-85, 2-89
logging off 2-37
login environment 2-85
LOGNAME variable 2-51
looping
  break command 2-79
  continue command 2-79
  do statement 2-65, 2-68
  done statement 2-65, 2-68
  end statement 3-60
  for statement 2-64
  foreach statement 3-60
  in statement 2-65
  iteration 2-65
  test command 2-73
  while statement 2-67, 3-60
ls command 2-4, 2-6, 2-12, 2-18, 2-42, 3-5, 3-29, 3-51

M

mail command 2-23, 2-31, 2-81
MAIL variable 2-51, 3-48
metacharacters
| 2-22, 2-24, 3-19
|| 3-59
! 3-53
!! 3-42
!n 3-42
!x 3-42
2-59, 3-56, 3-64
$ 2-2, 2-46, 3-56
$# 2-48
$* 2-48, 2-65
$< 3-57
$? 2-63
% 3-32
& 2-2, 2-10, 2-21, 2-37, 3-28
'' 2-2, 2-13, 2-14, 3-24
( ) 3-28
* 2-2, 2-3, 2-7, 2-10, 2-75, 2-77, 3-20
- 2-10
. 3-11
.. 3-10
/ 3-11
; 2-2, 2-12
< 2-2, 2-17, 3-14
<< 3-16
> 2-2, 2-17, 2-54, 3-14, 3-27
>! 3-38
>& 3-15
>> 2-2, 2-18, 2-54, 3-15, 3-27
? 2-2, 2-6, 2-7, 2-77, 3-20, 3-56
?* 2-10
[ ] 2-2, 2-9, 2-10, 2-77, 3-20
\ 2-2, 2-13, 3-24
`` 2-2, 2-28, 2-53, 2-56
directory indicators 3-10
escape sequences 3-24
filename substitution 3-20
redirection operators 3-8
releasing special meanings 3-24
{ } 3-59
~ 3-12, 3-47
mkdir command 2-43
modifiers 3-43, 3-62
mv command 2-7, 2-43

N

named variables 2-50
noclobber variable 3-38, 3-48
noglob variable 3-49, 3-64
nohup command 2-37
nonomatch variable 3-49
notify variable 3-49

O

output control 3-34
output redirection 2-16, 2-17, 2-18, 2-19, 2-21, 2-22, 2-53, 3-14, 3-27, 3-28
output substitution 2-28

P

pack command 3-28

PATH variable 2-42, 2-51, 2-88, 3-35, 3-37, 3-49
pathnames 2-74, 3-10, 3-12
permissions 2-42
pg command 3-34
PID, process identification 2-34, 3-28, 3-29
pipe metacharacter 2-2, 2-22, 2-24, 3-19
popd command 3-46
positional parameters 2-46, 2-57, 2-67
pr command 2-5, 2-10
process control
  at command 2-29
  background jobs 3-28
  batch command 2-29
  exit command 2-64
  fg command 3-32
  foreground jobs 3-28
  interrupt signal 3-31
  jobs command 3-32
  kill command 2-35, 3-32
  nohup command 2-37
  process status 2-34
  ps command 2-34
  quit signal 3-31
  return codes 2-63, 2-64
  terminating processes 2-28, 2-35, 3-31
process identification 2-34, 3-28, 3-29
process numbers 2-11, 2-30, 2-34, 3-28, 3-29
process status 2-34
programming the shell 2-40, 2-41, 2-59, 3-56
prompt command 3-53
prompt variable 3-35, 3-37, 3-41, 3-49
ps command 2-34
PS1 variable 2-90
PS2 variable 2-51
pushd command 3-46
pwd command 2-12

Q

quit signal 3-31

R

read command 2-53, 2-71
redirection
  input 2-16, 2-17, 3-14
  output 2-16, 2-17, 2-18, 2-19, 2-21, 2-22, 2-53, 3-14, 3-27, 3-28
redirection operators 3-8, 3-14
rehash command 3-41
relative pathnames 3-12
repeat command 3-54
return codes 2-63, 2-64
rm command 2-4, 2-23
root directory 3-11

S

savehist variable 3-37, 3-40, 3-49
screen control 3-34
security system 2-42, 3-51
set command 3-35, 3-37, 3-41, 3-56
setenv command 3-38
sh command 2-41, 2-81
shell programming
  # metacharacter 3-56

$ metacharacter 2-50, 2-57, 3-56
$# metacharacter 2-48
$* metacharacter 2-48
$< metcharacter 3-57
$? metacharacter 2-63
? metacharacter 3-56
command termination enquiry 2-63, 3-59
comments 2-59, 3-64
conditional constructs 2-70, 3-60
control statements 3-60
csh command 3-56
debugging 2-81
exit command 2-64
expressions 3-59
file enquiry expressions 3-59
invoking a shell script 2-41, 2-54, 3-56, 3-65
looping 2-64
modifiers 3-62
named variables 2-50
naming scripts 2-43
positional parameters 2-46
return codes 2-63, 2-64
set command 3-56
sh command 2-41, 2-81
shell script compatability 3-65
special parameters 2-48
unset command 3-57
variables 2-48, 2-51, 3-56
shell scripts 2-40, 2-41, 2-59, 3-56
shell variables 2-42, 2-46, 2-51, 3-35, 3-38, 3-49
shells 3-1
sort command 2-21, 3-17, 3-29
source command 3-41
special parameters 2-48
spell command 2-20
standard input 3-8
standard output 3-8
status variable 3-35, 3-49
stty command 2-85, 2-86
switch statement 3-61
system load 2-29, 3-54
system prompts 3-2
system responses 3-2

T

tail command 2-86
tee command 2-82
TERM variable 2-51, 2-53, 2-77, 2-89, 3-35, 3-38
terminal identification 2-34
terminal options 2-77, 2-85, 2-86
TERMINFO variable 2-51
test command 2-73
time command 3-54
time variable 3-49
TTY, terminal identification 2-34
TZ variable 2-52

U

umask command 3-37
unalias command 3-55
unset command 3-55, 3-57
unsetting variable values 3-55
user variable 3-35
using the terminal 3-2
utilities 3-1

V

variables
  $status 3-59
  argv 3-35, 3-48, 3-56
  assigning values 2-53, 2-57, 2-65, 3-56
  CDPATH 2-51, 3-48
  changing values 3-41
  cwd 3-35, 3-46, 3-48
  histchars 3-48
  history 3-37, 3-48
  HOME 2-51, 2-74, 2-88, 3-35, 3-38, 3-48
  IFS 2-51
  ignoreeof 3-38, 3-48
  LOGNAME 2-51
  MAIL 2-51, 3-48
  named variables 2-50
  noclobber 3-38, 3-48
  noglob 3-49, 3-64
  nonomatch 3-49
  notify 3-49
  output substitution 2-56
  PATH 2-42, 2-51, 2-88, 3-35, 3-37, 3-49
  positional parameters 2-46
  prompt 3-35, 3-37, 3-41, 3-49
  PS1 2-90
  PS2 2-51
  savehist 3-37, 3-40, 3-49
  set command 3-35, 3-37, 3-41
  setenv command 3-38
  shell 3-35, 3-38
  shell variables 2-88, 3-35, 3-49
  special parameters 2-48, 2-63
  status 3-35, 3-49
  TERM 2-51, 2-53, 2-77, 2-89, 3-35, 3-38
  TERMINFO 2-51
  time 3-49
  TZ 2-52
  unalias command 3-55
  unset command 3-55, 3-57
  unsetting values 3-55
  user 3-35
  verbose 3-49
verbose variable 3-49
vi command 2-40

W

while statement 2-67, 3-60
who command 2-47, 2-69
wildcards (see: metacharacters)
write command 3-14

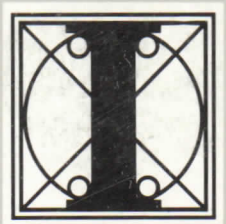
{

{ } command termination status enquiry 3-59

~

~ home directory indicator 3-12, 3-47

Printed in Italy



olivetti