LSX Computer Line

Operating Systems

X/OS UNIX® System V-based Operating System Advanced Utilities

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 Advanced Utilities

User Guide

olivetti

This manual is a user guide to the advanced utilities of the LSX X/OS operating system. It sets out to describe the more advanced utilities, which require a basic level of knowledge of the operating system. This information is available in the *User Guide*, which describes the commonly used utilities, and the *Shell / C Shell X/OS Command Langauge User Guide*, which describes the two shell environments supplied with X/OS.

SUMMARY

The manual begins with a brief Introduction. This describes the conventions used in the tutorial chapters The main body of the manual comprises a number of chapters, each containing one or more tutorials explaining the use of the X/OS advanced utilities. These are distributed as follows:

1. Introduction

2. The File Handling Commands:
   AT, BATCH, BDIFF, CRONTAB, CSPLIT, DIRCMP, EGREP/FGREP, JOIN, NEWFORM, OD

3. The Calculators:
   BC, DC, FACTOR, UNITS

4. The Editors:
   BFS, EDIT, EX

5. The Message and News Systems:
   MAILX, NEWS, WRITE

6. The Networking Systems:
   CU, UUCP, UUNAME, UUSTAT, UUTO, UUX

7. The Line Printer Utilities:
   CANCEL, LP, LPSTAT

8. The Terminal Filters:
   300, 300s, 4014, 450, GREEK

9. The System Handling Utilities:
   LOGNAME, TABS, TAR, TTY, WHO

REFERENCES

Read first ...

X/OS Operating Guide - Code 4055390 Y

X/OS User Manual - Code 4043610 C

For further information, read ...

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

This manual is the *Advanced Utilities User Guide* for the X/OS operating system. It sets out to present a clear guide to the *advanced* commands and utilities considered to be essential for effective use of the X/OS operating system, but which are used less frequently, or which rely on knowledge of the *basic* utilities. These basic utilities are described in the *User Guide*.

This first chapter of the manual is an introduction to the tutorials presented in the following chapters. Next comes the tutorials, which are organised into a number of chapters.

Each utility is described using the same basic pattern. There is a short *Introduction* offering a general description of the utility. Following this introduction is a section defining the *Syntax* of the command, showing the arguments which must be supplied, and the options that may be chosen. The meaning and contents of each is described in the next section, the *Description*, which is primarily a list of the possible elements in the command line. The last section is the *Examples* which provides one or more sample command lines. Each is described in terms of the effect it has, and the system output, such as prompts or status lines, is illustrated. An attempt has been made to illustrate the most commonly used functions offered by the utilities. In some cases, useful, but not immediately apparent, facilities are set out, with a step by step explanation of what is happening. For an example of this, see the last example in the **tar** tutorial.

The conventions used by the manual are simple: whenever an X/OS command line is shown, the following symbols are used:

> this symbol represents the X/OS system prompt. In reality, it may adopt just about any form, depending on how the system and the individual user's system account has been set up. The system prompt is printed whenever X/OS is ready to accept

a new command line. Where utilities, such as the editors and calculators, have their own prompts, these are explained prior to the sample screens.

key whenever a specific key is to be pressed, its keyboard legend is printed in bold type. For example, the symbol **CR** represents pressing the carriage return key. This key-stroke has the effect of entering the command line just typed. X/OS attempts to execute the command line as soon as this key is pressed. The **CTRL-d** type sequence will be encountered frequently: this implies that the key marked **CTRL** or **CONTROL** should be held down while the **d** key is typed. This is called a control sequence.

The *Syntax* section also has its conventions. The command or utility name always appears first, followed by one or more *arguments*. In some cases, these are mandatory, in others, they are optional. Where items are optional, they are enclosed in square brackets. Elipses indicate that the argument may be repeated. Where an argument may take a number of actual values, these are listed in the section entitled *Description*. For example, a command might be defined as follows:

```
tea [options] [user ...]
```

In such a case, it would be usual to see an explanation like the following:

DESCRIPTION

The **tea** command is used by the system administrator to make a cup of tea according to the arguments entered on the command line. These have the following form

options this argument may take one or more of the following values:

- **m** specifies the addition of milk.
- **l** specifies the addition of lemon.
- **s***n* specifies the addition of *n* lumps of sugar.

user identifies the user to receive the cup of tea. If no *user* is specified, the system administrator receives the cup of tea. User login names are used.

Note that the default tea has no milk, lemon or sugar.

A database of user's tea preferences is available in the file */etc/user/tea*.

From this information, it will be realised that a valid example of this command, along with possible system response, might be

```
>tea -ms2 spike nik sue CR
**** system run out of tea bags ****

>
```

In such cases, the system prompt is printed after the completion of each command line.

In some cases, an example may set out a sequence of commands. Where this occurs, cross references to the appropriate manuals and chapters are given.

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.

THE FILE HANDLING COMMANDS

INTRODUCTION

This chapter consists of 11 tutorials, each covering one of the more advanced of the X/OS file and directory handling commands. The utilities covered supply command scheduling, file and directory comparison and text formatting, among others. Other X/OS file and directory handling utilities are covered in chapter 3 of the *User Guide*.

The utilities included in this chapter are as follows:

AT	a command scheduler
BATCH	a command scheduler
BDIFF	big file comparison system
CRONTAB	a command scheduler
CSPLIT	a context file splitter
DIRCMP	directory comparison system
EGREP/FGREP	pattern search utilities
JOIN	relational database operator
NEWFORM	changes text file formats
OD	a dump system

The tutorials are presented in alphabetical order.

AT: executes commands at a specified time.

INTRODUCTION

This is a short tutorial-style introduction to the **at** utility which executes a command line at a time specified by the user. Any output from the command is mailed to the user instead of appearing on the screen, unless re-directed to a file or a printer.

SYNTAX

```
at time [date] [+increment]

at -r job ...

at -l [job ...]
```

DESCRIPTION

Command lines using **at** consist of the **at** command line (with arguments), a line break, and then the command to be executed.

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

time identifies the time at which the delayed job is to be executed. A 24-hour clock is assumed, unless one of the following optional characters is given:

A specifies a.m.

P specifies p.m.

N specifies noon

M specifies nidnight

The time is entered as two digits for the hour, an optional separator (:), and two digits for the minutes, for example **14:05**, or **1200M**.

date can take the form of the name of the month, a space, and the date of the day, for example **July 21**, or just the name of the day, for example **Thursday**. To specify an execution time exactly one week ahead, the word **week** can be entered.

+*increment* specifies a time interval, after which the command should be executed. It takes the form of a number followed by one of the following words: **minute**, **hour**, **day**, **week**, **month** or **year**. The plural form is also accepted. The word **now** can be used to initialise the command line, see below.

job identifies the command to be executed.

-r removes jobs already scheduled for **at** or **batch** (see below).

-l lists all jobs already scheduled using **at** or **batch**. Output consists of job numbers as allocated by the system. Jobs scheduled for **at** have the filename extension *.a* while those scheduled for **batch** have the extension *.b*.

EXAMPLES

The following example shows how to run an archiving command at 15:30 on July 21. Remember that the > symbol represents the X/OS system prompt, and that the symbol **CR** indicates that the carriage return key should be pressed in order to enter the command line. **CTRL-d**, that is holding down the key marked **CTRL** or **CONTROL** and pressing **d**, should be entered to indicate the end of the command sequence. The system responds with a confirmation line that contains the job number assigned to archive command, and the full time and date at which it is to be run.

```
>at 1530 July 21 CR
ar q testarc file1 file2 CR
CTRL-d
job 14426.a at Tue July 19 15:30:00 1987

>
```

The next example illustrates how to execute the archive command three hours ahead, assuming that it is currently 12:15, July 21.

```
>at now + 3 hours CR
ar q testarc file1 file2 CR
CTRL-d
job 14537.a at Tue July 21 15:15:00 1987

>
```

The third example shows how to check which archive commands are awaiting execution. The user is called **spike**.

```
>at -l CR
spike   14537.a   Tue  July 21 12:17:00 1987

>
```

Finally, the command intended for delayed execution can be cancelled using the -r option, giving the job number assigned to the command:

```
>at -r 14537.a CR

>
```

BATCH: executes commands at a later time

INTRODUCTION

This is a short tutorial-style introduction to the **batch** utility, which allows the user to execute commands at a later time, when the system load permits. Any output from the command is mailed to the user rather than appearing on the screen, unless redirected into a file.

SYNTAX

---

batch
command lines
CTRL-d

batch < file

DESCRIPTION

Batch has two formats. The first, above, accepts input directly from the user. The argument *command lines* identifies the commands that are to be executed. Typing **CTRL-d** indicates that the list of commands is complete.

The second format accepts input from a file. The argument *file* identifies the file that contains a sequence of commands to be executed at a later time.

Note that once a number of jobs have been scheduled using **batch**, these can be listed using the command **at -l** command. Removing batch jobs can be done using the **at -r** command. Both of these are covered in more detail, in the **at** tutorial in this manual.

EXAMPLES

The example shows how an archiving command can be executed at a later time. 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.

```
>batch CR
ar q testarc file1 file2 CR
CTRL-d
job 14587.b at Tue July 21 18:04:00 1987

>
```

The second example uses an input file to perform the same operation. The **cat** command is used to display the contents of the input file, here called *cmdfile*.

```
>cat cmdfile CR
ar q testarc file1 file2

>batch < cmdfile CR
job 14587.b at Tue July 21 18:04:00 1987

>
```

BDIFF: big file comparison utility

INTRODUCTION

This section is a brief tutorial introduction to the **bdiff** file comparison utility. It works by comparing the two files, then printing output in the form of pointers to the changes that must be made to make the two files the same. Files that are too large to be handled by the **diff** utility should be run through **bdiff**. The files are first split into segments, then standard **diff** operations are performed. The output is identical except that line numbers are adjusted to take account of the fact that the files have been split. For details of the **diff** utility, see the tutorial in the *User Guide*. See also the *diff*(1) entry in the *Utilities Reference Manual*.

SYNTAX

```
bdiff file1 file2 [n] [-s]
```

DESCRIPTION

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

file1 the name of the first file to be compared

file2 the name of the second file to be compared

n specifies the number of lines to be contained in each segment

-s suppresses **bdiff** diagnostics. Note that normal **diff** diagnostics are still printed

If either of the two *file* arguments is replaced by a hyphen (-), the specified file is compared with input from the terminal. Text should be entered exactly as it is to be compared with the specified file. End-of-input is signalled by **CTRL-d**. Note that terminal input may be entered at the keyboard, or it may be output piped into **bdiff** from another process (see the third example, below).

The default size of a segment is 3500 lines. If this is still too large for **diff** operations to contend with, the third argument, *n*, can be specified. Segmenting, whether to 3500 lines or some other specified number, is carried out by scanning the two files for the first occuring difference, and ignoring all text occuring before that point. The remainder of each file is divided into segments, and comparison proceeds from that point.

Note that if both the *n* and **-s** arguments are given, they must appear in this order.

The output from **bdiff** takes the form of two lines for each difference found. The first line is that found in *file1*, preceded by a less-than symbol (<). The second line is that found in *file2*, preceded by a greater-than symbol (>). The two lines are separated by a series of hyphens. Between each comparison pair, are the line numbers, in the form *xxx***c***yyy*, where *xxx* is the number of the line in *file1* and *yyy* is the number of the line in *file2*. The **c** stands for *compared with*.

EXAMPLES

The following example shows how **bdiff** can be used to compare two large files, called *release2.txt* and *release3.txt*. In these examples, the **$** character is used to represent the X/OS system prompt. **CR** is used to specify that the carriage return key should be pressed in order to enter the command.

```
$bdiff release2.txt release3.txt CR
33c33
< relating to release 2 of the documentation.
---
> relating to release 3 of the documentation.
174c174
< This release (2) contains the following sections:
---
> This release (3) contains the following sections:
186c186
< Updates (as of 12/7/1986)
---
> Updates (as of 10/1/1987)

$
```

The next sample screen shows how to split the files *oldchptl.txt* and *newchptl.txt* into segments of 1500 lines, and compare them, as before:

```
$bdiff oldchptl.txt newchptl.txt 1500 CR
2440c2440
< is a sample of the output
---
> shows the form of the output

$
```

Note that in this case, the first difference was discovered in the second of the 1500 line segments, that is, in line 2440. Although this is line 940 of the segment file, **bdiff** automatically adjusted the line count.

The next example shows how to format a file called *chpt3*, using a formatter called **form**, and to compare the output with a file called *oldchpt3*.

```
$form chpt3 | bdiff oldchpt3 - CR
214c214
< output from the old file
---
> output from the piped new file

$
```

Note that although two files are involved, only one is specified in the **bdiff** half of the pipe. Note also that the first line of the comparison pair to be displayed is that taken from *oldchpt3*. The line taken from the newly formatted *chpt3* can be displayed first by reversing the order of *oldchpt3* element and the - in the command line.

CRONTAB: clock used to schedule commands

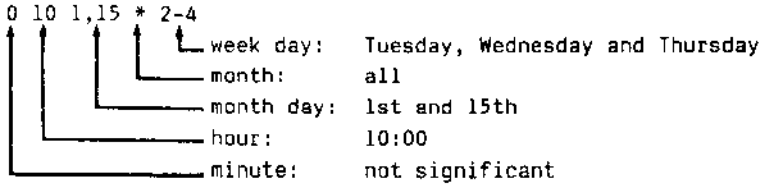
INTRODUCTION

This chapter is a tutorial introduction to the **Crontab** command. This facility allows selected users to submit jobs for execution, and is especially useful when commands need to be run on a regular basis, such as weekly reports or accounting systems. Users may use **crontab** only if their user-names are contained in the file */usr/lib/cron/cron.allow*. Users are specifically denied permission to use **crontab** if their user-names are contained in file *usr/lib/cron/cron.deny*. Where neither of these two files exist, root user only is allowed access to **crontab**.

Crontab will accept input in the form of a command file, or as standard input. Command files contain lines divided into five or six fields. Each field is an integer string, as follows:

```
minute (0-59)
hour (0-23)
day of the month (1-31)
month of the year (1-12)
day of the week (0-6, Sunday = 0)
optional command line
```

A single field may contain multiple values separated by commas; ranges separated by a hyphen, or an asterisk to denote all legal values. A command file containing the following sample data line: **0 0 1,15 * 2-4** would therefore execute a command as follows:



That is, the command would be executed at 10:00 on the 1st and 15th, and every Tuesday, Wednesday and Thursday, of every month.

If the sixth line of the command file is omitted, **crontab** accepts a command from the standard input. All commands are executed in the user's home directory. Output from the command is forwarded in the user via **mail** unless re-directed, therefore re-direction tends to be the standard practice.

SYNTAX

```
crontab [-l] [-r] [file]
```

DESCRIPTION

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

file specifies the file that contains the command lines that are to be executed by **cron**. If this argument is omitted, command lines are read from the standard input.

-l lists the contents of the *crontab* file for the invoking user.

-r removes the user's files from the *crontab* directory.

Note that none of these arguments can be used in combination.

EXAMPLES

The following example shows the **cat** command being used to display the contents of a file called *activities*. This file contains a **who** command, to find out who is using the system, and a **ps -a** command to list the active processes. These commands are to be executed every week-day (Monday to Friday) at 8:15 throughout the year. The output is re-directed into a file called *spy*. The file called *activities* is executed by specifying the filename as an argument to the **crontab** command. Remember that the **>** in bold type represents the system prompt, and the symbol **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>cat activities CR
15 8 * * 1-5 who;ps -a >> spy

>crontab activities CR

>
```

The next example illustrates how the same effect can be produced by entering the command line from the keyboard. All three lines are entered by the user. Note the end-of-input marker, **CTRL-d**.

```
>crontab CR
15 8 * * 1-5 who;ps -a >>spy CR
CTRL-d
```

The last example shows how to list the contents of the **crontab** command file using the -l argument.

```
>crontab -l CR
15 8 * * 1-5 who;ps -a >> spy

>
```

CSPLIT: split files by context

INTRODUCTION

This is a tutorial introduction to the **csplit** utility which divides files into segments according to conditions entered as arguments to the command line. An identifier ranging from **xx00** up to a maximum of **xx99** is prefixed to each segment as an identifier. Segment **xx00** consists of lines from the beginning of the original file up to, but not including, the line specified as the first argument to **csplit**. Segment **xx01** contains lines ranging from that identified in the first argument up to, but not including, that identified by the second argument. The last segment ranges from the line identified by the last argument up to the end of the original file. Note that the original file is not affected.

Note that a character count is printed by **csplit** for each segment as it is created. Note also that **csplit** terminates, and deletes all segments already created, whenever an error is detected. Both of these default activities can be supressed (see below).

SYNTAX

```
csplit [-s] [-k] [-f prefix] filename arg ...
```

DESCRIPTION

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

-s turns off the character count facility.

-k overrides the error handling facility which deletes all previously created segments. Existing segments are preserved, but **csplit** halts at the point of error, and no further processing occurs.

-f *prefix* segments are identified within the range *prefix***00** to *prefix***99** instead of the default **xx00** to **xx99**.

filename identifies the file to be processed. **Csplit** begins at the beginning of the file, and searches for the first argument, *arg1*, see blow. That section is written into a file, and the argument is used as the beginning of the next section

arg arguments can take any of the following forms:

/string/ [**+**/**-***n*] the line containing the character string is used as the dividing point between one segment and the next. Note that if the string has blanks or special characters, it must be enclosed in quotes. An optional **+***n* or **-***n* may be added to the string definition, where *n* is a line-number. This has the effect of indicating how many lines following (+) or preceding (-) the line

containing the string, are to be placed in the segment.

%*string***%** — this option functions like the */string/* option, above, except that the relevent lines are excluded from the lines used to form segments.

xxx — a segment file is created from the current line up to, but not including, line number *xxx*. This line becomes the current line.

{*num*} — repeats the preceding argument *num* times. If the preceding argument is a string, that character string is searched for *num* times. If the preceding argument is of the type *xxx*, the original file is split *num* times, every *xxx* lines. Note that it is safer to use the **-k** option in this case, because if *num* is set too high, all existing segment files will be deleted

EXAMPLES

The first example shows **csplit** used to create segments identified using the standard notation, starting at **xx00**. Each segment is to be 150 lines long, and to be sure that the whole file is split, an arbitrary number of repetitions has been specified, 200. Note that if the original file is longer than 30000 lines, it will still not all be processed. Character counts have been supressed. The original file is called *testfile*.

Remember that the > 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.

```
>csplit -s -k testfile 150 {200} CR

>
```

The next example creates a series of three segments from *testfile* identified by the names **seg00** to **seg02**. The first contains lines from the start of *testfile* up to, but not including, the line containing the string **volcanic eruptions**. The second ranges from the line containing this string up to, but not including, the line containing the string **basalt**. The third segment ranges from this line to the end of *testfile*. Note that the character counts have not been suppressed.

```
>csplit -f seg testfile "/volcanic eruptions/" /basalt/ CR
4214
751
3218

>
```

The last example creates a single segment called **xx00** from *testfile*, consisting of text beginning with the line containing the string *orthoclase feldspar* and ending at the end of the file.

```
>csplit testfile "%orthoclase feldspar%" CR

>
```

DIRCMP: directory comparison

INTRODUCTION

This chapter is a short tutorial-style introduction to the **dircmp** utility which compares two directories and generates output about the names and contents of the files. Files that are found only in one directory are listed first, followed by the files with names in common. The files present in both directories are compared, and the output indicates whether the files are the same or different.

SYNTAX

```
dircmp [-d] [-s] dir1 dir2
```

DESCRIPTION

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

-d executes a comparison of the files common to both directories, and indicates the measures to be taken to bring the two into agreement. For full details of the format of this output, see the **diff** tutorial in the *User Guide*.

-s suppresses messages about identical files. Output therefore concerns only those files which are different from eachother.

dir1 the first directory to be compared

dir2 the second directory to be compared

EXAMPLES

The first screens show listings of the files contained in each of the sample directories, obtained using the **ls** command. Details of this command can be found in the *User Guide*. The **>** symbol 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.

The first listing shows the files held in directory *manual1*; the second listing is the files in directory *manual2*.

```
>ls manual1 CR
preface   contents    chpt1     chpt2
chpt3     appendixa   appendixb

>ls manual2 CR
preface   contents    chpt1     chpt2
stats     appendixa   appendixb

>
```

The next screen shows the two directories being run through **dircmp**.

```
>dircmp manual1 manual2 CR
July 21 11:06 1987 ../manual1 only and ../manual2 only  Page 1

./chpt3       ./stats


July 21 11:06 1987 Comparison of ../manual1 ../manual2  Page 1

directory     .
different     ./preface
different     ./contents
```

```
different     ./chpt1
same          ./chpt2
different     ./appendixa
same          ./appendixb

>
```

EGREP, FGREP: pattern search utilities

INTRODUCTION

This is a tutorial introduction to the **egrep** and **fgrep** pattern search utilities. Both systems will accept files, data input from the terminal, or piped output from other commands. In the event of more than one file being searched, the matching lines are identified by the filename. Note that the **grep** tutorial is available in the *User Guide*.

Character patterns take the form of regular expressions or fixed character strings in the style of the **ed** text editor. When specifying patterns, it is usual practice to enclose them within single quotes in order to release special character meanings.

Egrep is the expression search utility which searches for full regular expressions. Expressions may be defined using the following conventions:

- a regular expression followed by a plus sign, +, matches 1 or more occurences of the pattern.

- a regular expression followed by a question mark, ?, matches 0 or 1 occurences of the pattern.

- two regular expressions separated by a pipe symbol, |, or a new line, **CR**, matches two patterns separated by either.

- regular expressions may be grouped by enclosing them in parentheses, ().

Fgrep is the fast search utility which searches for fixed patterns.

SYNTAX

```
egrep [option ...] [expr] [file ...]

fgrep [option ...] [string] [file ...]
```

DESCRIPTION

The options and arguments supported by these two utilities are as follows:

option the options available to **egrep** and **fgrep** are as follows:

- **-b** prints the block number of the line containing the regular expression specified.
- **-c** prints only the number of lines matching the regular expression specified.
- **-e** *expr* same as the *expr* argument, see below, but is useful where the expression contains a hyphen (-).
- **-f** *file* the pattern (specified as *expr* for **egrep** and *string* for **fgrep**), is taken from *file*.
- **-l** outputs only the filenames containing matching expressions.
- **-n** each matching line is identified by a line number.

`-v` prints lines not matching the pattern.

`-x` prints lines matching the pattern exactly. This is available only to the **fgrep** utility.

expr defines the search pattern for **egrep**.

string defines the search pattern for **fgrep**.

file defines the files to be searched, separated by a space where more than one filename is given.

EXAMPLES

The following examples illustrate **egrep** being used to search through two files, called *list1* and *list2*. Both contain a list of some of the elements plus their atomic weights. The first contains entries entered at random, while the netries in the second are ordered according to the atomic weight of the element.

In the first example, the two list files are searched for the patterns **helium** and **curium**. The **-n** option is used to display the number of the line matching the pattern. Note that where there are more than one file, the name of the matching file is given, and that the output fields are separated by colons (:). Remember that the **>** symbol in bold type represents the X/OS system prompt, and that **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

```
>head 4 list1 CR
047 silver
096 curium
018 argon
026 iron
```

```
>head 4 list2 CR
001 hydrogen
002 helium
007 nitrogen
010 neon

>egrep -n `helium|curium' list1 list2 CR
list1:2:096 curium
list1:2:002 helium
list2:?:002 helium
list2:28:096 curium

>
```

The next example illustrates how the new line (**CR**) can be used instead of the pipe symbol (|) to separate patterns. Notice that **egrep** supplies a secondary system prompt for the second line. Note also that the output is the same.

```
>egrep -n `helium CR
>curium' list1 list2 CR
list1:2:096 curium
list1:2:002 helium
list2:2:002 helium
list2:28:096 curium

>
```

The third example shows the **-v** option being used to scan the two files to find lines that do not contain the patterns **helium** and **curium**. Only a few lines from the middle of the output have been reproduced.

```
>egrep -v `helium|curium' list1 list2 CR
.
.
list1:050 tin
list1:030 zinc
list1:033 arsenic
list2:001 hydrogen
list2:007 nitrogen
list2:010 neon
.
.

>
```

The next example uses the **-f** option to read the patterns to be located from a file called *patterns*. This file contains two patterns, which are listed below, using the **cat** command. The effect of these will be to locate all elements beginning with the letters **c** to **f**, and all elements with atomic weights in the range 80 to 90 which begin with the letter **p**. Note that all of the **grep** family of utilities use the **ed** rules for specifying a regular expression. For details, see the **ed** tutorial in the *User Guide*, or the *ed*(1) entry in the *Utilities Reference Manual*.

```
>cat patterns CR
... [c-f]
^.[8-9]. p

>
```

This file of patterns is now applied to the file *list2*. Note that no filename is given on the output line when only one input file is involved, and that line number printing has not been selected.

```
>egrep -f patterns list2 CR
020 calcium
029 copper
084 polonium
094 plutonium
096 curium

>
```

The entries for **calcium**, **copper** and **curium** are the only entries in *list2* that match the first condition, while **polonium** and **plutonium** are the only entries in the file matching the conditions of an atomic weight in the range 80-90 with **p** as the initial letter.

JOIN: relational database operator

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **join** utility, which joins common data fields taken from two separate files. The two files must have been ordered in ASCII code sequence, using **sort** before **join** is used. Standard input, for example from the keyboard, can be joined with data already held in a file. The fields are separated by a blank, a tab, or a new line after processing. Leading blanks or separators are stripped, and multiple separators are counted as one.

For each pair of fields joined, one line of output is generated. Each line consists of the common field, the rest of the line from the first file, then the rest of the line from the second file.

SYNTAX

```
join [options] file1 file2
```

DESCRIPTION

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

options The following options are available:

-a*n* generates an output line for unpaired input lines from file *n*, as well as the normal paired lines. Note that *n* takes the value 1 or 2.

-e *str* replaces empty output lines with the character string *str*.

-j*n m* joins on field number *m* of file *n*. If *n* is absent, use field number *m* of each file.

-o *list* in each output line, include the fields specified in *list*, where *list* contains entries in the form *n.m*, where *n* is a file number, and *m* is a field number.

-t*c* use character *c* as a separator or tab character. Every appearence of *c* in a line is significant.

file1 specifies the first file to be joined.

file2 specifies the second file to be joined

EXAMPLES

The following example begins by displaying the contents of two list files, *animals1* and *animals2*, using the **cat** command. For details of this command, see the **cat** tutorial in the *User Guide*. Each type of animal has been given a number, for example, all large cats are type 1. This number is the field common to both files. A list of related animals is then prepared using **join**.

The output therefore consists of the common field, then the rest of the line from the first file, followed by the rest of the line from the second file. Remember that the **>** symbol in bold type represents the X/OS system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>cat animals1 CR
1 leopard
2 mouse
3 baboon
4 falcon
5 horse

>cat animals2 CR
1 tiger
2 rat
3 chimpanzee
4 eagle

>join -a1 animals1 animals2 CR
1 leopard tiger
2 mouse rat
3 baboon chimpanzee
4 falcon eagle
5 horse

>
```

Note that the last line consists of the unpaired line from *animals1*. This was the result of the **-a1** option to the command line.

NEWFORM: changes text file format

INTRODUCTION

This section is a tutorial introduction to the **newform** utility which reads lines of text and reformats them. Text lines can be obtained from the standard input or from files. A wide range of options allow the formatting operations to be specified precisely.

SYNTAX

```
newform [-s] [-itabspec] [-otabspec] [-ln] [-bn]
[-en] [-pn] [-an] [-f] [-cchar] [-bn] [file ...]
```

DESCRIPTION

The options and arguments to **newform** are as described below. Except for -s, which must be specified first, options and arguments can be given in any order, and may be repeated. Note that they are processed in the order they are specified on the command line. This means that the effect of a command can differ considerably, depending on the order of its components.

-s shears the leading characters from each input line up to and including the first tab. Note that a tab must exist on each line, otherwise an error message will be returned. Up to eight of the characters removed are appended to the end of the line. If more than eight characters have been removed, the last character appended is replaced with an asterisk (*), and the remaining characters are discarded.

Note that the sheared characters are saved internally until all the other options have been executed. Only after the command line has been fully processed, are the sheared characters appended.

-i*tabspec* expands tabs into spaces. The *tabspec* option uses the same tab specifications as are used by the **tabs** utility. For details, see the **tabs** tutorial in this manual. If *tabspec* takes the form --, **newform** assumes that the tab specifications are to be found in the first line of the input. If no value is given for *tabspec*, the default of **-8** is used. A value of **-0** tells **newform** to expect no tabs: if any are found, they are treated as **-1**.

-o*tabspec* replaces spaces with tabs. This option uses the same specifications as **-i***tabspec*, above. Where *tabspec* is not specified, the default of **-8** is assumed. Where **-0** is specified, no replacement takes place.

-l*n* sets the effective line length to *n* characters. If not specified, a default of 80 spaces is assumed. If **-l** is specified without an *n* element, 72 character spaces are assumed. Tabs and backspaces are viewed as being single characters.

-b*n* checks text lines against the line length set by the **-l***n* option, and shortens by *n* characters, all lines longer than this effective line length. Characters are removed from the beginning of the line. If **-b** is specified without an *n* element, each line is truncated by as many characters as are needed to obtain the effective line length.

-e*n* performs the same operation as **-b***n* except that characters are removed from the end of

the line.

-p*n* prefixes *n* characters to the beginning of a line when the line length is less than that set by the -l*n* option. Spaces are the default prefix character. If -p is specified without an *n* element, the number of characters necessary to attain the effective line length set by the -l*n* option are prefixed.

-a*n* performs the same operation as -p*n* except that characters are appended to the end of the line.

-c*char* sets the prefix and append characters (see -p*n* and -a*n*, above) to *char*.

-f writes the tab specification format line on the standard output before printing the formatted output. The format line to be printed is determined by the format specified in the last -o option. If no -o is given, the format line to be printed will contain the default specification of -8.

file identifies the file to be formatted.

EXAMPLES

In this example, a file called *projects* contains a list of projects, and gives both their deadlines and their dates of completion. It is intended to remove the first field so that the file gives only project references and completion dates. The first operation involves using the **cat** command to display the current contents of the file. Details of this command can be found in the **cat** tutorial of the *User Guide*. Remember that the > symbol represents the system prompt, and that **CR** in bold type indicates that the carriage return key should be pressed in order to enter the command line.

The second operation reduces *projects* from three fields to two. The **-i** options without a *tabspecs* element expands existing tabs into spaces. The **-l1** argument sets the effective line length to 1 character. This ensures that the next argument, **-b12** is definately activated. In the event of the actual line length being longer than that set by **-l**, **-b12** will strip off the first twelve characters from each line. Because all the text lines have more than one character, the first twelve characters are stripped. Because the first field of data was exactly twelve characters in length, this whole operation has the effect of removing the **DEADLINE** field.

```
>cat projects CR
DEADLINE    PROJECT REF.    DATE COMPLETED

15/06/1987  AS-01/6         17/05/1987
21/06/1987  PL-02/6         21/06/1987
04/07/1987  CO-01/7         16/07/1987
09/08/1987  PL-01/8         -

>newform -i -l1 -b12 projects CR
PROJECT REF.    DATE COMPLETED

AS-01/6         17/05/1987
PL-02/6         21/06/1987
CO-01/7         16/07/1987
PL-01/8         -

>
```

Another version of this operation strips off the last field instead. This can be done by using the **-e** option instead of **-b**, and giving it a value of 14. This would have the following effect:

```
>newform -i -l1 -el4 projects CR
DEADLINE    PROJECT REF.

15/06/1987  AS-01/6
21/06/1987  PL-02/6
04/07/1987  CO-01/7
09/08/1987  PL-01/8

>
```

OD: dump utility

INTRODUCTION

This section is a tutorial introduction to the **od** utility which reads input and prints its contents in octal, ASCII, decimal, or hexadecimal format. The default mode is octal, hence the name. Input can be from a file, from the standard input, or piped into **od** from another command.

SYNTAX

```
od [option] [file] [[+]offset[.][b]]
```

DESCRIPTION

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

option specifies the output format. The available values are

- **-b** interprets bytes in octal
- **-c** interprets bytes in ASCII
- **-d** interprets words in unsigned decimal
- **-o** interprets words in octal. This is the default option. Calling **od** with no option causes output to be formatted in this style
- **-s** interprets 16-bit words in signed decimal

-x interprets words in hexadecimal

file identifies the input file. If this is not specified, **od** uses the standard input

offset points to the location in the file from which output begins. It is expressed as the number of bytes (in octal) to be skipped before output commences. Appending a full stop or period (.) causes the offset to be interpreted in decimal. Appending the letter **b** causes the offset to be interpreted in blocks of 512 bytes. If input is taken from the standard input rather than a file, a plus sign (+) must be prepended. This identifies the argument as being *offset* rather than *file*.

EXAMPLES

The first example shows the default form of **od** being used on a file called *listfile*. First, the contents of the file are displayed using the **cat** command, and then an octal dump is obtained. For details of **cat**, see the **cat** tutorial in the *User Guide*. Remember that the **>** in bold type represents the system prompt, and that **CR** indicates that the carriage return key should be pressed.

```
>cat listfile CR
cat
dog
mouse
horse
cow

>od testfile CR
0000000 060543 005164 067544 005147 067555 071565 005145 067550
0000020 071562 005145 067543 005167
0000030
```

Note that the first column gives a line number sequence.

The next example uses the **-b** option on *testfile*. This option outputs the octal value for each character. An offset has also been specified, so that **15** causes **od** to begin at the invisible new line character (012 octal) at the beginning of the line reading **horse**. Note that each input character has its own three digit code. It can be seen from this that the code 157 represents the letter **o**.

```
>od -b testfile 15 CR
0000015 012 150 157 162 163 145 012 143 157 167 102
0000030

>
```

INTRODUCTION

This third chapter of the guide supplies four tutorials, covering the X/OS calculator utilities. The utilities covered are as follows:

BC calculator utility

DC interactive desk calculator

FACTOR finds prime factors of numbers

UNITS unit conversion facility

The tutorials are presented in alphabetical order.

BC: Calculator utility

INTRODUCTION

This section is a tutorial introduction to the **bc** arbitrary precision calculator utility. It can be used to perform arithmetic operations, trigonometric functions, exponential calculations, natural logarithm calculations, Bessel functions, and number base convertions. These are supported by a range of in-built library functions. It will accurately handle infinitely large integers, and provides a scaling system for decimal point notation. Numbers to many decimal places can be handled. **Bc** will read input from command files and from the standard input.

Bc is a compiler offering its own command structure, but is not intended to be a complete programming langauge. It is a pre-processor for the **dc** calculator utility: output is piped into **dc** unless the **-c** option is used to specify compile-only operations. Note that **bc** uses a C-like notation, so that C programmers can use it without difficulty.

The command language uses a set of special operators and functions for performing calculations. These are as follows:

+	adds two numbers, and displays the result.
-	subtracts two numbers, and displays the result.
/	divides the left argument by the right argument, and displays the result.
*	multiplies two numbers, and displays the result.
%	displays the remainder from a division operation (/).

^	raises the left argument to the power indicated by the right argument.
sqrt	calculates the square root of a number.
scale	sets the accuracy of calculations.
a	calculates the arctangent of the right argument, and displays the result.
c	calculates the cosine of the right argument, and displays the result.
s	calculates the sine of the right argument, and displays the result.
e	calculates the exponential of the right argument, and displays the result.
l	calculates the natual logarithm of the right argument, and displays the result.
ibase	changes the base of the input.
obase	changes the base of the output.
scale	changes the number of digits to the right of the decimal point.
quit	quits from **bc**.

SYNTAX

---

bc
calculation

bc [-c] [-l] file

DESCRIPTION

Bc is a pre-processor to the **dc** calculator utility. Input to **bc** is piped to **dc** for computation. **Bc** has two formats. The first given above in the syntax definition allows the user to type in as many calculations as required. These are terminated with the **quit** command. The second format accepts input from a command file, using the following options:

-c causes the preprocessed output from **bc** to be compiled only, rather than piped to **dc** for calculation. The output is written to the standard output, and no calculations are performed.

-l causes the input to be passed through an arbitrary precision maths library, which contains the sine, cosine, exponential, logarithm, arctangent and Bessel functions. Examples of these in use are given below.

file identifies the file to be used by **bc**. Information regarding the contents and format of such files is given below.

EXAMPLES

This section contains a number of sample calculations, which illustrate the more commonly used **bc** facilities. Remember that the **>** symbol represents the X/OS system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

SIMPLE ARITHMETICAL OPERATIONS

The simplest operations are those involving two numbers and a single operator. The following example contains operations that respectively add, subtract, multiply and divide the numbers 12 and 7. Note that the division operation (12/7) produces the answer 1. To find the remainder from this calculation, the **%** operator was used.

```
>bc CR
12+7 CR
19
12-7 CR
5
12*7 CR
84
12/7 CR
1
12%7 CR
5
quit CR

>
```

The accuracy of such calculations can be altered using the **scale** command, described below.

CALCULATIONS WITH NEGATIVE NUMBERS

To carry out operations with negative numbers, the negative number must be preceded by the minus sign (-). In the next example, negative numbers are used to further illustrate the addition and subtraction operators.

```
>bc CR
-12+7 CR
-5
-12+-7 CR
-19
quit CR

>
```

RAISING NUMBERS TO A POWER

To raise a number to a power, **bc** uses the ^ operator. The following example illustrates this in use.

```
>bc CR
12^7 CR
35831808
quit CR

>
```

FINDING THE SQUARE ROOT

To calculate the square root of a number, **bc** uses the **sqrt** function. The number should be given inside parentheses:

```
>bc CR
sqrt(64) CR
8
sqrt(66) CR
8
quit CR

>
```

If the accuracy of the second operation in the last example is a little disappointing, **bc** supplies a means of changing the accuracy of a calculation, using the **scale** function.

CHANGING THE ACCURACY OF AN OPERATION

In the preceding examples, there have been several examples of calculations where the accuracy of the answer leaves much to be desired. This can be corrected using the **scale** command, which sets the number of digits after the decimal point. **Bc** will accept **scale** settings of up to 99. Note that **bc**'s default scale is 0. The following example shows a series of operations which use the **scale** command.

```
>bc CR
sqrt(66) CR
8
scale=5 CR
```

```
sqrt(66) CR
8.12403
scale=10 CR
sqrt(66) CR
8.1240384046
scale=2 CR
12/7 CR
1.71
scale=0 CR
scale CR
0
quit CR

>
```

Note that **scale** is a variable in the same way that registers are variables, and it can be handled in the same way. The last command line in the above example entered the **scale** command without arguments. This displays the current **scale** setting.

COMBINING CALCULATIONS

Bc allows the user to combine operations within a single calculation. The rules of precedence used are the same as those in standard arithmetic:

1. All calculations within parentheses are carried out first. Where nested parentheses occur, the innermost calculations are carried out first, followed by those calculations on the next level outwards.

2. Calculations involving square roots and exponentiation are performed next, from left to right.

3. Multiplication and division calculations are performed next, from left to right.

4. Addition and subtraction calculations are carried out last, from left to right.

The following example shows a calculation using multiple operations.

```
>bc CR
scale=5 CR
((7^3)/sqrt(15))+8(8*(3^7)+16)-14/2 CR
17593.56339
quit CR

>
```

CHANGING THE INPUT BASE

The default base for **bc** calculations is 10 (decimal). However, calculations can be carried out in octal or hexadecimal (or any other base between 1 and 16) by re-setting the base of the numbers used. In the following example, the input base is changed, so that numbers being input are translated from one base to another. At the beginning of the example, the input base is set to its default (10). Typing the number 644 causes **bc** to return the same number. **Ibase** is then used to change the input base from 10 (decimal) to 8 (octal). 644 (octal) is shown to be equal to 420 (decima)l. The input base is then changed from octal to base 16 (hexadecimal). Note that 16 (decimal) is equal to 20 (octal). Changing the input base requires base-change commands to be entered in the base currently set. 644 (hexadecimal) returns 1604 (decimal). Finally, the input base is changed back to decimal. Note that 10 is A (hex). 644 again returns 644.

```
>bc CR
644 CR
644
ibase=8 CR
644 CR
420
ibase=20 CR
644 CR
1604
ibase=A CR
644 CR
644
quit CR

>
```

Note that the current setting of the input base can be found by typing **ibase** on a new line.

CHANGING THE OUTPUT BASE

While **bc**'s default output base is decimal, it can be changed to octal or hexadecimal, (or any other base, including very large or strange bases, for example 0, 1 or negative) using the **obase** command. The following example begins with **bc** being set to its default output base, decimal. Inputting the number 644 returns 644. When **obase** is set to octal, 644 returns 1204. Next, **obase** is set to hexadeimal. In hexadeimal, 644 returns 284. Finally, **obase** is set back to decimal, where 644 again returns 644.

```
>bc CR
644 CR
644
obase=8 CR
644 CR
1204
obase=16 CR
644 CR
284
obase=10 CR
644 CR
644
quit CR

>
```

Note that the current setting of the output base can be found by typing **obase** on a new line.

USING REGISTERS

In addition to allowing simple one-line calculations, **bc** allows the use of up to 26 registers. These are used for storing values, and are identified by a single letter, from **a** to **z**. Register values are set by statements in the form of *ID=value*. The value can be an actual number or a calculation. The contents of the register will be the result of the calculation. To display the contents of a register, the register name is typed, followed by **CR**.

Note that if the calculator is being used in the form **bc -l**, only one register may be used.

In the following examples, three registers are set, their contents are checked, and they are used in a series of simple arithmetical operations. Note that registers can be set to have values that depend on the values of other registers.

```
>bc CR
scale=5 CR
x=8 CR
y=-12 CR
z=(2^x)/6 CR
x CR
8
y CR
-12
z CR
42
x+y+z CR
26.66666
x+y/z CR
7.43750
z-y-z CR
-10.66666
(x^3)+y/z CR
511.43750
quit CR

>
```

Note that a register that has been set with reference to another register that does not exist, will always contain 0:

```
>bc CR
y=-12 CR
z=(2^x)/6 CR
y CR
-12
z CR
0
quit CR

>
```

Note also that these registers are only accessible from within **bc**.

TRIGONOMETRIC AND EXPONENTIAL FUNCTIONS

Bc supplies an arbitrary precision maths library which can be accessed using the **-l** option on the command line. This option is used to perform the following operations:

find the sine of an angle

find the cosine of an angle

find the arctangent of a number

find the natural logarithm of a number

raise a number to the *e* power (2.718)

Bc performs all calculations on angles in radians. In order to perform a calculation in degrees, it is necessary first to translate the degrees into radians. The conversion factor is *degrees * 0.01745 = radians*. (The reverse conversion factor is *radians / 57.29578 = degrees*. These conversion factors are available using the X/OS **units** utility, which is described in this chapter. In this way, trigonometric calculations carried out in degrees involves two operations.

The next example illustrates how to obtain the sine of 55 degrees. Note the use of the **-l** option on the command line. The answer returned is 0.81904.

```
>bc -l CR
scale=5 CR
55*0.01745 CR
0.95975
s(0.95975) CR
```

```
0.81904
quit CR

>
```

The next example calculates the cosine of 1.25 radians (71.6 degrees). The answer returned is 0.31532.

```
>bc -l CR
scale=5 CR
c(1.25) CR
0.31532
quit CR

>
```

The third example calculates the natural logarithm of 45, returning the answer 3.80666.

```
>bc -l CR
scale=5 CR
l(45) CR
3.80666
quit CR

>
```

The fourth example calculates the exponential of 6, returning the answer 403.42878.

```
>bc -l CR
scale=5 CR
e(6) CR
403.42878
quit CR
```

The last example illustrates the use of a register in trigonometric operations. By establishing a register with the conversion factor that translates degrees into radians, calculations using degrees become simpler to enter. Using the register *x*, the sine and cosine of 52 degrees is found.

```
>bc -l CR
scale=5 CR
x=0.01745 CR
s(52*x) CR
0.78790
c(52*x) CR
0.61579
quit CR

>
```

ESTABLISHING FUNCTIONS

In addition to the 26 variable (register) names, **bc** will support up to 26 functions. These are also identified with a single letter (a-z), and are allowed to have the same names as existing variables. These are user-defined, and take the general form

```
define a(x){
   statement
   statement
```

```
    return
}
```

In this case, a function called *a* has been established with a single argument identified by *x*. It consists of a number of *statements*, and ends with the **return** instruction. Return of control from a function will occur automatically, even if the **return** instruction is not present, as soon as the statements within brackets, { }, have been executed. The **return** statement may take two forms: **return** entered alone causes the function to exit with a value of 0. **Return(***n***)** causes the function to exit with the value *n*. The value of *n* is determined by the calculations contained by the function.

Variables may be defined within the function. These are accessable only within the function, and so the same variable names may be used within several different functions. *Automatic* variables exist only during execution of the function. They are initialised to 0, and their contents are discarded on return from the function. Note that variables of the same name existing outside the function are not affected. Automatic variables must be declared in the *first* statement of the function. Note that a function may contain only one of these **auto** statements, but that it may declare a number of variables.

The following example declares a function called *a*. It accepts three values, representing the height, width and depth of a regular object, measured in inches. The function uses the automatic variable *v* to calculate the volume in cubic inches, and to translate the result into cubic centimetres. The metric volume of the object is returned to **bc** via this variable. The conversion factor for translating cubic inches into cubic centimetres is 16.38706.

After the function is entered, it is called assigning the values 11.4, 3.75 and 8.1 to *x*, *y* and *z* as the dimensions

in inches of the object.

```
>bc CR
scale=5 CR
define a(x,y,z){ CR
     auto v CR
     v=(x*y*z)*16.38706 CR
     return(v) CR
} CR
a(11.4,3.75,8.1) CR
5674.42920

>
```

Functions, once they have been entered, can be used in other calculations. For example, function *a*, as defined above, could be used in the following calculation, which determines how much free space remains after the object has been placed in a storage container measuring 100 centimetres in all three dimensions. The amount of free space is returned by the variable *p*.

```
p=(100^3)-a(11.4,3.75,8.1) CR
p CR
994325.65274
quit CR
```

Note that functions are maintained by **bc** only for the duration of the current session, unless they are loaded into **bc** using the *file* argument on the command line. For more details, see below.

SUBSCRIPTED VARIABLES

Bc allows the use of subscripted variables, to create one-dimensional arrays. The permitted range of values for a subscript is 0 to 2047, giving a maximum of 2048 possible elements. Any fractional part of the subscript is discarded before use. Such variables may be used in expressions, in function calls, and in return statements.

Subscripted variables are identified by a single letter (a-z). The name given to an array variable may be the same as that of a simple variable or a function. The array may then be used as an argument to a function call, or it may be declared as an automatic variable within a function, by using square brackets. The square brackets may contain an expression defining the range of the indexing. These statements take the form

```
f(a[])
```

and

```
define f(a[])
     auto a[]
```

where *f* is the function name, and *a[]* is the subscripted variable name.

CONTROL STATEMENTS

Functions may contain a variety of control statements which alter flow within the program or cause iteration. These statements take the following forms:

if (*relation*) *statement*

while (*relation*) *statement*

for (*expr1*;*relation*;*expr2*) *statement*

or

if (*relation*) {*statements*}

while (*relation*) {*statements*}

for (*expr1*;*relation*;*expr2*) {*statements*}

A *relation* takes the form of two expressions linked by one of the following six relational operators:

< less than

> greater than

<= less than or equal to

>= greater than or equal to

== equal to

!= not equal to

An example of a valid relation would be *x<y*.

The **if** statement causes execution of *statement(s)* only if *relation* is true. **While** causes repeated execution of *statement(s)* for as long as *relation* is true, otherwise control passes to the next statement in the program. **For** begins by executing *expr1*, then tests the *relation*. If true, it executes *statement(s)*. Then *expr2* is executed, and *relation* is again tested. This continues until *relation* is found to be false.

An illustation of the way these statements can be used is given in the next example. It defines a function, called *b*, which accepts a single number, and uses the **for** construction to calculate its factorial.

```
>bc CR
define b(x){ CR
     auto n,t CR
     t=x CR
     n=x-1 CR
     for(n=x-1;n>0;n=n-1)t=t*n CR
     return(t) CR
} CR
b(5) CR
120
b(12) CR
479001600
quit CR

>
```

Note that **for** and **while** statements can be terminated early using the **break** statement.

LOADING COMMAND FILES

Bc will lose functions typed in during a session as soon as the calculator is terminated. Commonly used or long function programs that require editing during entry, should be created using an editor, then loaded into **bc** when required.

The following example uses the **cat** command to display the contents of a file called *factorial*. It contains a copy of function *b* developed above. The second command line then loads the program into **bc**, and calls up the function in the usual way.

```
>cat factorial CR
define b(x){
     auto n,t
     t=x
     n=x-1
     for(n=x-1;n>0;n=n-1)t=t*n
     return(t)
}

>bc factorial CR
b(5) CR
120
b(12) CR
479001600
quit CR

>
```

Note that a program file can contain more than one function definition.

COMMENTS

In common with C programs, **bc** programs allow the use of comment lines. These must begin with /* and end with */, and may appear on a new line, or as part of an existing line.

RESERVED WORDS

The **bc** *reserved words* which have an in-built meaning are:

RESERVED WORDS			
auto	break	define	for
ibase	if	length	obase
quit	return	scale	sqrt
while			

DC: interactive desk calculator

INTRODUCTION

This section is a tutorial introduction to the **dc** interactive desk calculator utility. It is used to perform arbitrary-precision integer arithmetic, and provides facilities for the handling of scaled fixed-point numbers and number bases other than decimal. Input and output bases, and the number of fractional digits to be handled can be separately specified. The size of numbers that can be handled depends on the availability of core memory.

The **dc** utility handles input using a reverse Polish stacking system. As an user interface language to **dc**, **bc** is available. This is described in its own tutorial in this manual. Operations are compiled by **bc** before being passed to **dc**. Accordingly, some of the facilities described in this tutorial were not originally designed for direct use.

SYNTAX

```
dc [file]
```

DESCRIPTION

The **dc** calculator operates a pushdown stack system, in which numbers are taken from the top of the stack, and used in some operation. The result is pushed on to the stack. If a *file* argument is given on the command line, input is taken from the specified source until the file is empty. Input is thereafter taken from the standard input.

The rest of this section describes the functioning of **dc** in terms of the way it represents numbers, and uses its internal registers.

THE DC COMMANDS

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following construction is recognised:

number (e.g. 244)

The value of a number is pushed onto the stack. A number is an unbroken string of digits 0 through 9 and uppercase letters A through F (treated as digits with values 10 through 15, respectively). The number may be preceded by an underscore (_) to input a negative number and numbers may contain decimal points.

The top two values on the stack can be added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^).

The two entries are popped off the stack, and the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. An

exponent must not have any digits after the decimal point.

```
sx
```

The top of the main stack is popped and stored in a register named *x* (where *x* may be any character). If **s** is uppercase, *x* is treated as a stack; and the value is pushed onto it. Any character, even blank or newline, is a valid register name.

The value of register x is pushed onto the stack. Register x is not altered. If the l in

```
lx
```

is uppercase, register x is treated as a stack, and its top value is popped onto the main stack. All registers start with empty value which is treated as a zero by the command **l** and is treated as an error by the command **L**.

The following characters perform the stated tasks:

d the top value on the stack is duplicated.

p the top value on the stack is printed. The top value remains unchanged. Uppercase **P** interprets the top value on the stack as an ASCII string.

f all values on the stack and in registers are printed.

x treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of **dc** commands.

[...]
puts the bracketed character string onto the top of the stack.

q exits the program. If executing a string, the recursion level is popped by two. If **q** is uppercase, the top value on the stack is popped; and the string execution level is popped by that value.

<x *>x* *=x* *!<x* *!>x* *!=x*
the top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

v replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer.

! interprets the rest of the line as an X/OS software command. Control returns to **dc** when the command terminates.

c all values on the stack are popped; the stack becomes empty.

i the top value on the stack is popped and used as the number radix for further input. If **i** is uppercase, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o the top value on the stack is popped and used as the number radix for further output. If **o** is uppercase, the value of the output base is pushed onto the stack.

k the top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The

scale factor must be greater than or equal to zero and less than 100. If **k** is uppercase, the value of the scale factor is pushed onto the stack.

z the value of the stack level is pushed onto the stack. Uppercase **Z** replaces the top number on the stack with its length.

? a line of input is taken from the input source (usually the console) and executed.

;: used in **bc** array operations.

INTERNAL REPRESENTATIONS OF NUMBERS

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0 to 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100s complement notation, which is analogous to twos complement notation for binary numbers. The high-order digit of a negative number is always -1 and all other digits are in the range 0 to 99. The digit preceding the high-order -1 digit is never a 99. The representation of -157 is 43,98,-1. This is called the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when it is convenient.

An additional byte is stored with each number beyond the high-order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high-order digit. The value of this extra byte is called the *scale factor* of the number.

THE ALLOCATOR

The **dc** program uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is through the allocator. Associated with each string in the allocator is a 4-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and **dc** is via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of two. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Leftover strings are put on the free list. If there are no larger strings, the allocator tries to combine smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long.

If a string of the proper length cannot be found, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in **dc**. If the allocator runs out of headers at any time in the process of trying to allocate a string, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end of string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

INTERNAL ARITHMETIC

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. The **scale** register limits the number of decimal places retained in arithmetic

computations. The **scale** register may be set to the number on the top of the stack truncated to an integer with the **k** command. The **K** command may be used to push the value of **scale** on the stack. The value of **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations includes the exact effect of **scale** on the computations.

ADDITION AND SUBTRACTION

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

The addition is performed digit by digit from the low-order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers, replacing the high-order configuration 99,-1 by the digit -1. In any case, digits that are not in the range 0 through 99 must be brought into that range, propagating any carries or borrows that result.

MULTIPLICATION

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly follows the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low-order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

DIVISION

The scales are removed from the two operands. Zeros are appended, or digits are removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise, the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. If it turns out to be one unit too low, the next trial quotient is larger than 99; and this is adjusted at the end of the process. The trial digit is multiplied by the divisor, the result subtracted from the dividend, and the process is repeated to get additional quotient digits until the remaining dividend

is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form with propagation of carry as needed. The sign is set from the sign of the operands.

REMAINDER

The division routine is called, and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

SQUARE ROOT

The scale is removed from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand. The method used to compute the square root is Newton's method with successive approximations by the rule.

```
sub n+1 = (X sub n + Y/X sub n)
```

The initial guess is found by taking the integer square root of the top two digits.

EXPONENTIATION

Only exponents with 0 scale factor are handled. If the exponent is 0, then the result is 1. If the exponent is negative, then it is made positive; and the base is divided into 1. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared, and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

INPUT CONVERSION AND BASE

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with an underscore (_). The hexadecimal digits A through F correspond to the numbers 10 through 15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base (**ibase**) is initialized to 10 (decimal) but may, for example, be changed to 8 or 16 for octal or hexadecimal to decimal conversions. The command I pushes the value of the input base on the stack.

OUTPUT COMMANDS

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers are output by typing the command **f**. The **o** command is used to change the output base (**obase**). This command uses the top of the stack truncated to an integer as the base for all further output. The output base in initialized to 10 (decimal). It works correctly for any base. The command **O** pushes the value of the output base on the stack.

OUTPUT FORMAT AND BASE

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 are used for decimal-octal or decimal-hexadecimal conversions.

INTERNAL REGISTERS

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register *x*. The *x* can be any character. The command **l***x* puts the contents of register *x* on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

STACK COMMANDS

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack onto the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

SUBROUTINE DEFINITIONS AND CALLS

Enclosing a string in brackets, [], pushes the ASCII string on the stack. The **q** command quits or (in executing a string) pops the recursion levels by two.

INTERNAL REGISTERS - PROGRAMMING DC

The load and store commands, together with [] to store strings, the **x** command to execute, and the testing commands (**<**, **>**, **=**, **!<**, **!>**, **!=**), can be used to program **dc**. The **x** command assumes the top of the stack is a string of **dc** commands and executes it. The testing commands compare the top two elements on the stack and, if the relation holds, execute the register that follows the relation. For example, to print the numbers 0 through 9,

```
[lipl+ si  li10>a]sa
0si  lax
```

PUSHDOWN REGISTERS AND ARRAYS

These commands are designed for use by a compiler, not directly by programmers. They involve pushdown registers and arrays. In addition to the stack that commands work on, **dc** can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **L***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers but not as pushdown stacks. The command **l** does not affect the top of the register stack, but **s** destroys what was there before.

The commands to work on arrays are **:** and **;**. The command **:***x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. The command **;***x* loads the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

MISCELLANEOUS COMMANDS

The command **!** interprets the rest of the line as a X/OS software command and passes it to the X/OS operating system to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

EXAMPLES

This section contains a number of sample calculations which illustrate the more commonly used features of **dc**. The **>** symbol in bold type represents the X/OS system prompt, while **CR** indicates that the carriage return key should be pressed in order to enter the command line.

SIMPLE ARITHMETICAL OPERATIONS

The simplest operations are those involving two numbers and a single operator. The operators are respectively **+**, **-**, ***** and **/**. The following example adds the numbers 12 and 7, and pushes the result on to the stack. It then quits from **dc**.

```
>dc CR
12 CR
7 CR
+ CR
p CR
19
q CR

>
```

CALCULATIONS WITH NEGATIVE NUMBERS

To carry out operations with negative numbers, the negative number must be preceded by the underscore (_). In the next example, negative numbers are used to further illustrate the arithmetical operators.

```
>dc CR
_12 CR
5 CR
+ CR
p CR
-7
q CR

>
```

DETERMINING THE REMAINDER OF A DIVISION OPERATION

The division operator (/) rounds off the remainder. To find the remainder of a division operation, the % notation is used. The following example starts off by dividing 12 by 7, to return the answer 1. The % symbol is then used to return the remainder.

```
>dc CR
12 CR
7 CR
/ CR
p CR
1
12 CR
7 CR
% CR
p CR
5
q CR

>
```

RAISING NUMBERS TO A POWER

To raise a number to a power, **dc** uses the ^ operator. The following example illustrates this in use.

```
>dc CR
12 CR
7 CR
^ CR
p CR
35831808
q CR

>
```

FINDING THE SQUARE ROOT

To calculate the square root of a number, **dc** uses the **v** function.

```
>dc CR
64 CR
v CR
p CR
8
q CR

>
```

Note that results are rounded off to the nearest whole number. There is, however, a way of increasing the accuracy of such calculations, using the **k** command, which is described in the next section.

CHANGING THE ACCURACY OF AN OPERATION

In the preceding examples, there have been several examples of calculations where the accuracy of the answer may leave much to be desired. This can be corrected using the **k** command, which sets the number of digits after the decimal point. **Dc** will accept **k** settings of up to 99. Note that **dc**'s default scale is 0. The following example shows a series of operations which use the **k** command.

```
>dc CR
66 CR
v CR
p CR
8
5 CR
k CR
66 CR
v CR
p CR
8.12403
q CR

>
```

COMBINING CALCULATIONS

Dc allows the operations to be combined. The result is not printed until the **p** command is used. The following example shows a calculation using multiple operations.

```
>dc CR
12 CR
7 CR
* CR
32 CR
```

```
- CR
15CR
2 CR
^ CR
+ CR
p CR
277
q CR

>
```

This calculation is the **dc** equivalent of (12*7)-32+(15^2).

CONTINUOUS OPERATIONS

Different types of calculation can be carried out without quitting from **dc**. This involves using the **c** command, which clears the stack of existing values. The next example shows this in use:

```
>dc CR
15 CR
2 CR
^ CR
p CR
225
c CR
5 CR
k CR
17.2 CR
3 CR
^ CR
p CR
5088.448
q CR
```

CHANGING THE INPUT BASE

The default base for **bc** calculations is 10 (decimal). However, calculations can also be carried out in base 8 (octal) or 16 (hexadecimal) by re-setting the base of the numbers used. In the following example, the input base is changed, using the **i** command, so that numbers can be input in different bases.

```
>dc CR
8 CR
i CR
644 CR
420 CR
+ CR
p CR
692
q CR
```

CHANGING THE OUTPUT BASE

While **dc**'s default output base is decimal, it can be changed to octal or hexadecimal, using the **o** command. The following example begins with setting the output base to hexadecimal.

```
>dc CR
16 CR
o CR
64A CR
2C1 CR
+ CR
p CR
347
q CR
```

FACTOR: find prime factors of a number

INTRODUCTION

This chapter is a short tutorial-style introduction to the **factor** utility which calculates the prime factors of any whole number. Output consists of a list of the prime factors of the number entered, with the exception of 1.

SYNTAX

---

factor number

factor
number

DESCRIPTION

If a number is entered as part of the command line, the prime factors of that number are listed. If the command **factor** is entered on its own, a succession of numbers can be entered.

EXAMPLES

The first example finds the prime factors of a single number, 12. The **>** symbol in bold type represents the X/OS system prompt, while the symbol **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>factor 12 CR
12
    2
    2
    3

>
```

The second example illustrates how to enter a sequence of numbers, 12 and 4. Note the end-of-input marker, **q**, standing for quit, on a new line. **Factor** will continue to accept numbers until it reads an end-of-input character.

```
>factor CR
12 CR
    2
    2
    3
4 CR
    2
    2
q CR

>
```

UNITS: unit conversion system

INTRODUCTION

This section is a tutorial introduction to the **units** conversion utility. **Units** supplies the conversion factors (divisors and multipliers) for converting from one measurement system to another. A wide range of length, weight, mass, currency, electrical, fluid and temperature units may be converted.

The available conversion factors are stored in the file */usr/lib/unittab*.

SYNTAX

---

units

DESCRIPTION

The command has no options. The arguments which specify the units to be converted are entered in response to screen prompts printed by the utility. These take the form of

```
you have:
you want:
```

The first prompt, *you have*, asks for the units that are being used as a reference. The second, *you want*, asks for the type of unit into which the reference is to be converted.

System response takes the form of two numbers. The first, preceded by an asterisk, is the multiplier: the second, preceded by a slash, is the divisor.

Units will continue to prompt with the **you have:** message until the **CTRL-d** end-of-input character is typed.

EXAMPLES

The first example illustrates the successful use of **units** to obtain the conversion factor needed to obtain miles from inches. 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.

```
>units CR
you have:  inch CR
you want:  mile CR
        *  1.578283e-05
        /  6.336000e+04

you have:  CTRL-d

>
```

The response means that a mile represents 63360 inches, and an inch represents 0.0000157283 miles. The next example enters an invalid unit type.

```
>units CR
you have:  amp CR
you want:  crzzt CR
Cannot recognise crzzt

you have:  CTRL-d
```

The last example illustrates what happens when two incompatible units are entered. The user here is trying to convert amps into inches:

```
>units CR
you have:  amp CR
you want:  inch CR
conformability
1.000000e+00 coul/sec
2.540000e-02 m

you have:  CTRL-d

>
```

Where incompatible units are entered, **units** responds with the message **conformability** and displays the most sensible conversion factors for the two units entered. In this case, an inch is given to be 0.0254 of a meter, and that an amp is equal to 1 coul/sec.

INTRODUCTION

This chapter of the *Advanced Utilities User Guide* covers the more advanced editors. The essential X/OS editors, **ed** and **vi** were covered in the *User Guide*. The tutorials included in the present chapter are as follows:

BFS a big file scanner

EDIT a text editor

EX a text editor

The tutorials are presented in alphabetical order.

BFS: big file scanner editor

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **bfs** utility which allows the user to scan the contents of file without making any changes. **Bfs** uses **ed** conventions, with the exception of the commands that alter the contents of a file.

SYNTAX

```
bfs [-] name
```

DESCRIPTION

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

- suppresses the files size display.

name identifies the file to be scanned.

Bfs operates directly on the file, rather than on a copy, which is how **ed** operates. Files of up to 1024 kbytes can be scanned, where the maximum number of lines that can be handled is 32000, each line being up to 255 characters. It is often useful where large files are to be divided into smaller sections for later editing. The splitting of a file can be done using the **split** or **csplit** utilities. Note that **csplit** has its own tutorial in this manual, while **split** is described in the *User Guide*.

EXAMPLES

This section describes the features of the **bfs** scanner by way of a number of simple examples. Remember that 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.

STARTING UP

Bfs cannot be used to create new text, so existing files only can be used. However, parts of the contents of an existing file can be copied into a new file. This operation is described below.

The examples in this section will use a file called *shelley*, which has the following contents:

```
>cat shelley 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;
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.

>
```

Scanning the file called *shelley* with **bfs** involves typing

```
>bfs shelley CR
621
```

The figure printed by **bfs** immediately after the command line is entered, is a size count of the file, in this case, the number of characters contained by *shelley*. Using the - on the command line suppresses the size count.

```
>bfs - shelley CR
```

USING A PROMPT

Bfs will accept commands, but will not display a prompt on the screen. If this is confusing, the **bfs** prompt can be forced, by typing the command **P**. All further commands will be prompted, with the * character. The examples in this section all use this prompt, displayed in bold type. Note that entering **P** a second time will switch the prompt off again.

Note that using the prompt has an impact on the **bfs** error message system. If the prompt has not been turned on, errors will be flagged only by **?**. Self-explanatory messages are available only when the prompt is used.

```
P CR
*
```

ACCESSING THE CONTENTS OF A FILE

In order to display the contents of the file, a line identifier must be typed. Typing **1** causes **bfs** to display the current file from the first line. The first line then becomes the *current* line, and all commands entered are executed with reference to this line. The current line can be displayed at any time, using the **p** command.

```
*1 CR
I met a traveller from an antique land
*p CR
I met a traveller from an antique land
*
```

FILE DATA

Two commands exist that return information about the file currently being scanned. The = command returns the number of lines in the file, while **f** returns the name of the file. The following example uses these:

```
*= CR
14
*f CR
shelley
*
```

QUITTING FROM BFS

To leave **bfs**, the **q** command is used. Because no changes can be made to a file using **bfs**, no warning is given when quitting. Control is returned to the X/OS shell:

```
*q CR

>
```

DISPLAYING MORE LINES

By typing a single line number, the line identified by that number will be displayed on the screen. However, if more than one line is required, a range can be specified, as in the following example:

```
*1,5p 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
*
```

From this, it can be seen that the **p** command actually stands for **print**. When entered with an argument, that is, one or more line numbers, it will display the specified lines. When entered on its own, it displays the current line only.

The last line of a file is identified using the dollar character (**$**). To display the last five lines of *shelley*, the following command would be used:

```
*9,$ CR
`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.
*
```

Remember that the number of lines in the files can be discovered using the = command.

It will be seen later that the **p** command can be used with pattern searching and marking, to display lines that satisfy a condition, such as the presence of a particular string of characters.

MOVING THROUGH THE FILE

It is possible to move both forward and backward through a file. This is done using the + and - commands. Used on their own, they move one line. Used with a number, they move forward or backward by that number of lines. The next example starts at line 1.

```
*p CR
I met a traveller from an antique land
*+ CR
*p CR
Who said: Two vast and trunkless legs of stone
*+10 CR
*p CR
Nothing beside remains. Round the decay
*-5 CR
Which yet survive, stamp'd on these lifeless things,
*$ CR
The lone and level sands stretch far away.
```

Note that the last command entered caused the last line of the file to be displayed.

It is also possible to specify an absolute line position, by simply entering the required line number:

```
*6 CR
Tell that its sculptor well those passions read
*
```

SEARCHING FOR TEXT

Where a line is to be displayed, but its position within the file is not known, the pattern search facilities can be used. There are a number of these, depending on the type of search to be carried out. A general rule of pattern searching is that the pattern to be located must be on a single line.

SEARCH WITH WRAP-AROUND

This form of the pattern search mechanism has two versions. The first involves enclosing the pattern in slashes (/). This causes **bfs** to search forward through the file for the pattern. If it reaches the end of the file without finding the pattern, it *wraps around* to the beginning of the file, and continues searching until either the pattern is found, or until the current line is again encountered. If the pattern is found, the line containing the pattern becomes the current line, and is displayed. If the pattern is not found, a message is displayed saying so, and the current line remains unchanged.

```
*p CR
Which yet survive, stamp'd on these lifeless things,
*/shatter'd/ CR
Half sunk, a shatter'd visage lies, whose frown
*= CR
4
*/Ozymndias/ CR
Ozymndias not found
*/decay Of that/ CR
decay Of that not found
*
```

The first command displayed the current line. The second requested a search for the pattern **shatter'd**, which was found in line 4 of the file. The second search operation requested a string that does not exist. The last command requested a search for text that does exist, but which includes a line break in the original.

The second version of the wrap-around search mechanism involves enclosing the pattern to be located within question marks (?). This version works in the same way, except that it searches backwards through the file from the current line. If it reaches the beginning of the file without locating the pattern, it wraps around to the end of the file, and continues until the pattern is found, or until the current line is again encountered.

```
*p CR
Half sunk, a shatter'd visage lies, whose frown
*?Nothing beside? CR
Nothing beside remains. Round the decay
*= CR
12
*
```

SEARCH WITHOUT WRAP-AROUND

A second system of searching is available that does not wrap around to the beginning or end of the file. The first version seraches forward through the file, and either displays the line containing the pattern, or displays a message. This involves enclosing the pattern to be located within greater-than angle brackets (>). If the pattern exists before the current line, it will not be found, because **bfs** stops at the end of the file.

```
*p CR
Nothing beside remains. Round the decay
*>colossal wreck> CR
Of that colossal wreck, boundless and bare,
*= CR
13
*>Stand in the desert> CR
Stand in the desert not found
*
```

The last command failed to locate its pattern because the pattern appears before the current line.

The second version of the search mechanism without wrap-around involves enclosing the pattern to be located within less-than angle brackets (<). This causes **bfs** to serach backward through the file until the pattern is located, or until the beginning of the file is encountered. If the pattern does not exist, or if it occurs afterthe current line, it will not be located.

```
*p CR
Which yet survive, stamp'd on these lifeless things,
*<trunkless< CR
Who said: Two vast and trunkless legs of stone
*= CR
2
```

```
*<Stand in the desert< CR
Stand in the desert not found
*
```

The first search located the pattern, and accordingly, line 2 became the current line. The second search tried to locate a pattern occuring after the current line, and **bfs** failed to locate it.

REPEATING A SEARCH OPERATION

Where a pattern occurs more than once within a file, the first occurence may not be the one required. Accordingly, **bfs** supplies a way of repeating the last search executed. This involves entering the relevent command symbols, without repeating the pattern to be located. Entering **>>** will search for the last pattern entered, from the current line to the end of the file. Entering **<<**, **??** or **//** will search for the last pattern in the appropriate fashion.

```
*p CR
Which yet survive, stamp'd on these lifeless things,
*>and> CR
The hand that mock'd them and the heart that fed;
*>> CR
The hand that mock'd them and the heart that fed;
*//
`My name is Ozymandias, king of kings:
*
```

Note that from the initial current line, the first search operation located the pattern **and** in line 8. In fact, it

found the pattern in the word **hand**. Repeating the search displays the same line, this time for the word **and**. The third search operation uses different notation, but has the same effect, that of locating the next incidence of **and**, this time in the word **Ozymandias**. Note that the intervening line, beginning with **And**, was not displayed, because **And** and **and** are treated as different patterns.

GLOBAL SEARCHES

It is also possible to request **bfs** to display all the lines containing a specified pattern. This involves the **g** (global) command. The **v** command displays all lines that do not contain the specified pattern. Patterns are specified using the the same four types of notation described above, that is **//**, **??**, **>>** and **<<**. In this case, the **g** and **v** commands appear before the pattern, and can be combined with other commands:

```
*1 CR
I met a traveller from an antique land
*g/and/p
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,
And wrinkled lip and sneer of cold command
The hand that mock'd them and the heart that fed;
`My name is Ozymandias, king of kings:
Look on my works, ye Mighty, and despair!'
Of that colossal wreck, boundless and bare,
The lone and level sands stretch far away.
*= CR
14
*v/and/p CR
Half sunk, a shatter'd visage lies, whose frown
Tell that its sculptor well those passions read
Which yet survive, stamp'd on these lifeless things,
And on the pedestal these words appear:
Nothing beside remains. Round the decay
```

```
*= CR
12
*
```

The first operation used the **g** and **p** commands to locate and print all lines containing the pattern **and**. The second used **v** and **p** to locate and print all lines not containing the pattern. Note that after each operation, the current line was the last line displayed.

SPECIAL SEARCH NOTATIONS

Bfs provides six special search notations, which allow searches for ranges of characters and characters in specific positions within a line, and which turn off the special meaning of certain characters. These are as follows:

- a single dot matches any single unknown character except the new line character. The next example performs a global search and print (as explained above), for any three-character pattern beginning with **t** and ending with **a**.

```
*1 CR
I met a traveller from an antique land
*g/t.a/p CR
I met a traveller from an antique land
Tell that its sculptor well those passions read
The hand that mock'd them and the heart that fed;
Of that colossal wreck, boundless and bare,
*
```

In the first displayed line, the pattern **tra** from **traveller** was located. In the second, third and last lines, the pattern **tha** from **that** appears. Both of these match the **t.a** notation.

* an asterisk matches any sequence of unknown characters, except the first ., \, [or ~ in a sequence. In the following example, a global search and print is performed with the notation **m*a**.

```
*g/m*a/p CR
I met a traveller from an antique land
Stand in the desert. Near them on the sand,
And wrinkled lip and sneer of cold command
The hand that mock'd them and the heart that fed;
`My name is Ozymandias, king of kings:
Look on my works, ye Mighty, and despair!'
Nothing beside remains. Round the decay
*
```

Seven lines of the poem contain one or more occurences of the pattern. The first line contains the sequences **met a** and **m a**, both of which consist of an **m** and an **a**, separated by zero or more other characters.

\ the backslash is used to release the special meaning of certain characters. For example, the dollar sign (**$**) is the **bfs** notation for the last line of a file. In order to search for a literal occurence of the dollar sign, it is necessary to tell **bfs** that the special meaning is not intended. This involves placing a \ immediately before the **$**.

```
*/\$/p CR
$ not found
*
```

The full list of characters whose special meaning can be released in this way is **.**, *****, **[**, **]**, **$**, **%** and **^**.

[] square brackets are used to enclose a range of characters. In the example, a global search and print is performed on the pattern **th[ae]**. This means that any occurence of **tha** or **the** will be displayed.

```
*g/th[ae]/p CR
Stand in the desert. Near them on the sand,
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;
And on the pedestal these words appear:
Nothing beside remains. Round the decay
Of that colossal wreck, boundless and bare,
*
```

$** a dollar sign, when used as part of a character pattern, means the end of the line. The following example performs a global search and print on the pattern **d$. Only those lines whose last character is a **d** will be selected.

```
*g/d$/p CR
I met a traveller from an antique land
And wrinkled lip and sneer of cold command
Tell that its sculptor well those passions read
*
```

Three lines end in **d**. Note that two others are disqualified because the last character is actually a punctuation mark.

^ the circumflex acts in the same way as **$**, except that it signifies the beginning of the line. The following example performs a global search and print on the pattern **^A**.

```
*g/^A/p CR
And wrinkled lip and sneer of cold command
And on the pedestal these words appear:
*
```

MARKING LINES

Bfs supplies a facility for marking lines. These marks then allow lines to be found easily. The mark command is **k**, followed by a single lower case letter, to act as the identifier. This means that a maximum of 26 marks can be set. A marked line is made the current line with the **'** command, followed by the identifier letter. In the following example, two marks, called **a** and **b**, are set. These then become the current line, in turn. The first command prints the whole file, using the **$** notation to stand for the last line.

```
*1,$p 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;
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.
*4 CR
Half sunk, a shatter'd visage lies, whose frown
*ka CR
*10 CR
```

```
`My name is Ozymandias, king of kings:
*kb CR
*1 CR
I met a traveller from an antique land
*'a CR
Half sunk, a shatter'd visage lies, whose frown
*'b CR
`My name is Ozymandias, king of kings:
```

A list of the currently set marks can be obtained, using the **xn** command. The example uses the marks set above.

```
*xn CR
a
b
*
```

Note that the mark identifiers are listed, but not the lines.

Marks are lost when the **q** command is used to quit **bfs**. However, when a new file is selected for scanning, marks set in the previous files are retained. Selecting a new file is described below.

CREATING A NEW FILE

Although new text cannot be written using **bfs**, new files containing copies of all or part of existing files can be created. The **w** command, followed by a filename, copies the whole of the current file into a new file identified by the filename entered. In the next example, a copy of the whole poem is made, the new file being called *shelley2*.

```
*w shelley2 CR
```

```
621
*
```

As soon as the new file is created, a character count is displayed, confirming its size. Note that an existing file will be over-written if its name is unintentionally used.

To copy part of an existing file, the numbers of the first and last lines required are prepended to the **w** command. In the next example, a file called *extract1* is created that consists of the first five lines from the poem. The second command creates a file called *extract2*, comprising the last five lines. Note the character counts.

```
*1,5w extract1 CR
221
*10,$w extract2 CR
209
*
```

CHANGING FILES

When work on a file has been completed, the session can be ended using the **q** command. If another file is to scanned, the **e** command (edit) will change files without exiting from **bfs**. In the example, the current file is changed from *shelley* to *extract1*.

```
*e extract1 CR
*1,$p 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
*
```

The print command displayed the whole of *extract1*, that is, the first five lines of the original poem.

ACCESSING THE SHELL FROM BFS

It is sometimes necessary to execute a normal shell command while using **bfs**. This can be done by accessing the shell with the ! command. This is followed by the shell command to be executed. Once the command is executed, **bfs** regains control, and displays a !, before returning to the current line. Normal **bfd** editing can continue, or a further shell command can be issued. In the example, the **ls** shell command is used to list the files held in the current directory.

```
*!ls CR
shelley     shelley2   extract1    extract2
!
*
```

ESTABLISHING VARIABLES

Bfs supplies a means of establishing and using variables. These are identified by a single digit from 0 to 9. They are used to store expressions that may be in frequent use, and are established as follows, using the **xv** command:

```
xv [id] [value]
```

In the next example, a variable identified as **1** is assigned the value **8**. A second variable, called **2** is assigned the value **10,$p**. Note that the space between the identifier and the value is optional.

```
*xv1 8 CR
*xv210,$p CR
*
```

Once established, variables can be accessed by way of their identifiers, whicf follow the **%** command. In the example, the current file, *shelley*, is scanned using these two variables.

```
*5,%1p CR
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;
*%2 CR
`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.
*
```

Note that in order to use the **%** character as a literal percentage mark, it should be preceded by the backslash.

Variables can also be used to store the first line of output from a shell command. The example establishes a variable called **3**, and sets it to contain the date, as retruned by the X/OS **date** utility. Note that the **!** character must be used, in order to specify that the command is to be executed by the shell.

```
*xv3 !date CR
*
```

COMMAND FILES

Where a complex sequence of **bfs** commands is required, it is possible to enter these in the desired sequence, and use the resulting file as the input to **bfs**. This is done using the **xf** command, followed by the name of the file that contains the command sequence. In the example, a shell command, **cat** displays the contents of a simple command file called *script*. This is then applied to the current file, *shelley*.

```
*!cat script CR
g/th[ae]/p
!
*xf script CR
Stand in the desert. Near them on the sand,
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;
And on the pedestal these words appear:
Nothing beside remains. Round the decay
Of that colossal wreck, boundless and bare,
*
```

The command file facility allows the use of conditional statements, and command execution status testing. These are explained in the *bfs*(1) entry of the *Utilities Reference manual*.

EDIT: text editor

INTRODUCTION

This is a tutorial-style introduction to the X/OS **edit** utility, which is a simplified version of the **ex** editor, also described in this manual. Casual and new users are recommended to use **edit** rather than one of the more complex and powerful editors such as **ex** or **vi**.

SYNTAX

```
edit [filename]
```

DESCRIPTION

The *filename* argument to **edit** specifies the file to be created or changed. Before working on an existing file, **edit** creates a copy, on which all editing operations are performed. This copy is called the *buffer*, and until the buffer is saved in a permanent form, all text entered remains temporary. At the end of an **edit** session, the new text is written to a new version of the file, while the original is retained. This means that after quitting from **edit**, two versions of the file are available: the original as a back-up, and the new version.

User commands are entered on the **edit** command line, which is identified by a single colon (:). Commands can be used to affect the current line, a specified range of lines, or the entire file. Throughout this tutorial, the phrase *current line* is taken to mean the line currently being operated on. Most commands are English words, most of which have abbreviated forms for the more practised user. The next section illustrates the various functions

available to **edit**.

EXAMPLES

This section describes the **edit** utility with a series of example screens. These show the commands to be entered, and the effect they have on a short text file. This file is called *shelley*, and contains a poem. Throughout, the > symbol 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. The colon in bold type which appears at the beginning of a line is the **edit** prompt.

CREATING A NEW FILE

The first screen begins by creating a new file. Its name is *shelley*.

```
>edit shelley CR



"shelley" [New file]
:
```

Note that **edit** immediately printed a message indicating that *shelley* is a new file, and then displayed the **edit** prompt. These usually appear at the bottom of the screen.

It is possible to create a new file by typing only **edit** on the command line. In this case, the normal editing operations can be preformed, but on quitting, **edit** asks for a filename. This is described below, in the section entitled *More About The edit Command Line*.

ENTERING TEXT: APPEND

At this stage, an empty file called *shelley* has been created. The next stage is to enter some text. This is done using the **append** command, which can be shortened to **a**. The examples will use the long version of commands for the first few times before switching to the abbreviated forms. A list of the available commands and abbreviations is available at the end of the tutorial.

```
>edit shelley CR



"shelley" [New file]
:append CR
```

The **append** command puts **edit** into *insert* mode. As soon as **CR** is pressed, a blank line is displayed, and **edit** is ready to accept text lines. The next screen shows the first few lines being added. Note that input lines end with the **CR** key.

```
"shelley" [New file]
:append CR
I met a traveller from an antique land CR
Who said: Two vast and trunkless legs of stone CR
```

LEAVING INSERT MODE: THE DOT

The first two lines of the poem have been entered. To leave insert mode, a single full stop or period is typed on a new line. This causes **edit** to display its prompt on a new line, and to wait for a new command.

```
"shelley" [New file]
:append CR
I met a traveller from an antique land CR
Who said: Two vast and trunkless legs of stone CR
. CR
:
```

In the next screen, insert mode is entered again, and the rest of the poem is entered. Note that **append** always adds text after the current line. In order to add new text before the current line, **edit** supplies the **insert** command. This will be described below.

```
:append CR
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;
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.
. CR
:
```

FINDING OUT THE CURRENT LINE NUMBER

The current line is always the line pointed to by the editor. It is often known by the shorthand name ., and this name can be used to address the current line. It may be the case that a complex sequence of operations makes it difficult to remember the number of the current line. This can be discovered using the .= command. The next screen shows this in use.

```
:.= CR
14
:
```

The current line was found to be number 14.

DISPLAYING TEXT: PRINT

All 14 lines of the poem have now been entered. To display the contents of the file, the **print** command, abbreviated to **p**, is used. This command, entered without any arguments, displays the current line.

```
:print CR
The lone and level sands stretch far away.
:
```

The current line in this case is the last line entered. In order to display the whole file, a range of line numbers must be passed as an argument to the **print** command. There are several ways of doing this. The simplest argument is a single number telling **edit** to display a single line.

```
:5print CR
And wrinkled lip and sneer of cold command
:8,10print CR
The hand that mock'd them and the heart that fed;
And on the pedestal these words appear:
`My name is Ozymandias, king of kings:
:.,$print CR
`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 first command line told **edit** to display only the fifth line. The next command line specified a range of lines, 8 to 10, separated by a comma. The third command line used two code characters, **$**, which is shorthand for the last line in the **edit** buffer, and **.**, which is shorthand for the current line of the buffer. Accordingly, the command **.,$print** means *print the contents of the buffer from the current line to the end.* Whenever a **print** command is used, the last line to be displayed becomes the current line.

LISTING A FILE: LIST

In addition to the **print** command, **edit** supplies the **list** facility. This performs more or less the same function, but also displays special characters that **print** ignores. The next screen adds a new line to the buffer, containing a tab character. The first command re-enters insert mode. The notation **tab** indicates that the tab key was pressed at this point.

```
:a CR
here is another line containing a specialtab  character CR
. CR
:12,$p CR
Nothing beside remains. Round the decay
Of that colossal wreck, boundless and bare,
The lone and level sands stretch far away.
here is another line containing a special   character
:12,$list CR
Nothing beside remains. Round the decay$
Of that colossal wreck, boundless and bare,$
The lone and level sands stretch far away.$
here is another line containing a special^I character$
:
```

The **a** command was used to add a new line. The **p** command showed the effect of the **tab**, which was to insert spaces up to the next tab position. When **list** was used, all the special characters were displayed. The tab was indicated by the ^I notation, while all instances of the carriage return key being pressed were identified using the **$** symbol, which should not be confused with the user symbol which addresses the last line in the buffer.

DELETING LINES: DELETE

To remove a text line from the buffer, the **delete** or **d** command is available. This also takes line numbers as arguments. This means that specific lines or ranges of lines can be removed. The next screen shows the **p** command used without arguments, to identify the current line. It shows that the last line of the buffer is the current line. The **delete** command then removes it from the buffer. The last proper line of the poem then becomes the current line, and is displayed as confirmation.

```
:p CR
```

```
here is another line containing a special   character
:delete CR
The lone and level sands stretch far away.
:12,$delete CR
3 lines deleted
Look on my works, ye Mighty, and despair!'
:
```

The second use of **delete** removed the last three lines of the poem. Note that any **edit** command that affects more than one line of the buffer causes a warning message to be displayed.

In fact, deleting the last line of the buffer is a special case. In general, when a line is removed from the middle of the buffer, the next line becomes the current line. When the last line is removed, the last surviving line in the buffer becomes the current line.

Note that so long as no other command has been executed in the meantime, this **delete** command can be reversed, using the **undo** command.

REVERSING THE EFFECTS OF COMMANDS: UNDO

In fact, most **edit** commands can be reversed with **undo** or **u**. However, only the last command entered will be reversed, so it is important not to execute any other commands in between. The commands that cannot be affected by **undo** are the display commands such as **print** and **list**, and the **write** and **quit** commands. These two are described separately below.

The next screen begins by repeating the **delete** command entered above, then uses **undo** to rescue the last three lines of the poem.

```
:12,$delete CR
3 lines deleted
Look on my works, ye Mighty, and despair!'
:undo CR
3 more lines in file after undo
Nothing beside remains. Round the decay
:undo CR
3 fewer lines in file after undo
Look on my works, ye Mighty, and despair!'
:undo CR
3 more lines in file after undo
Nothing beside remains. Round the decay
:10,$p CR
'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 second time **undo** was executed, it reversed the effect of the first **undo**, and so re-deleted the three lines from the buffer. This is a useful facility when a complex operation has been carried out on the text, but it is not certain that its effects are desirable. The **p** command was used to verify that the poem had ultimately survived intact.

Note that throughout, **edit** displayed what had become the current line. Remember that the number of the current line can be found using the .= command.

MOVEMENT AROUND THE BUFFER

The basic movement commands are **CR** and **+**, which both move forward a line, and **-**, which moves back a line. Both **+** and **-** will take arguments, so that by combining one of these commands with a number, **edit** can be made to move forward or backward a specified number of lines. Both of these commands can be said to act *relative* to the current line.

Also available is the *absolute* notation, which involves specifying an actual line number. This involves typing a line number, or using the **$** or **.** notations, which have been encountered already. Used on its own, **$** tells **edit** to move to the last line of the buffer. These may be combined with numerical arguments, as follows: the address **$-5** points to the line located five lines before the end of the buffer. The address **.+10** points to the line located ten lines after the current line.

The next screen shows some of these in use.

```
:$ CR
The lone and level sands stretch far away.
:5 CR
And wrinkled lip and sneer of cold command
:+3 CR
The hand that mock'd them and the heart that fed;
:-7 CR
I met a traveller from an antique land
:
```

To view the next half screen of text, the sequence **CTRL-d** can be used. Pressing these keys together a second time will display another half screen of text, or the rest of the buffer, whichever is shortest. Pressing **CTRL-d** after the end of the buffer has been reached will cause **edit** to display the message

```
At EOF
```

which stands for End of File.

The text surrounding the current line can be displayed using the **z.** command. The last line displayed in this way becomes the current line. To get back to the line that was the current line before **z.** was entered, the '' command is used. The next screen shows some of these commands in use. The first command makes line 1 the current line.

```
:1 CR
I met a traveller from an antique land
:CTRL-d
I met a traveller from an antique land
Who said: Two vast and trunkless legs of stone
.
.
.
Look on my works, ye Mighty, and despair!'
Nothing beside remains. Round the decay
:p CR
Nothing beside remains. Round the decay
:z. CR
Who said: Two vast and trunkless legs of stone
Stand in the desert. Near them on the sand,
.
.
.
Of that colossal wreck, boundless and bare,
The lone and level sands stretch far away.
:p CR
The lone and level sands stretch far away.
:'' CR
Nothing beside remains. Round the decay
:
```

Note that the '' command restored the current line to the line that was current immediately before the z. command.

The z command will accept numerical arguments, for example z8 displays eight lines of text. For longer files than is used here, the z+ and z- commands are useful. The first displays a full 24 line screenful of text, starting at the current line. The second version displays a full 24 line screenful ending at the current line.

INSERTING TEXT: INSERT

The **append** command has already been used to add text to a file. This command always places the newly entered text after the current line. The **insert** command, abbreviated to i, adds text before the current line. The next screen adds a new line to the poem, between lines 6 and 7. The first command line makes line 7 the current line. The second command puts **edit** into insert mode.

```
:7 CR
Which yet survive, stamp'd on these lifeless things,
:i CR
Another line that doesn't rhyme CR
. CR
:1,$p
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
Another line that doesn't rhyme
Which yet survive, stamp'd on these lifeless things,
The hand that mock'd them and the heart that fed;
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.
:
```

Note that, as with **append**, insert mode was terminated by typing a single full stop or period on a line of its own. The new line was inserted immediately before the current line, that is, between lines 6 and 7. The last command line printed the whole poem.

The next command is **change**, which transforms existing text into new text.

CHANGING TEXT: CHANGE

The **change** command, abbreviated to **c**, will alter any specified lines. Entered without arguments, the current line is changed. When a line number or range of numbers is given, the change operation is performed on the lines specified. As with the other commands, changing more than one line causes **edit** to display a warning message.

Once the **change** or **c** command has been entered, **edit** supplies a blank line, as with **append** and **insert**. The new text is then typed in. Note that it is permitted to change a single existing line for more than one new lines. In this way, existing text can be altered and new text entered at the same time. Similarly, a number of existing lines can be changed into a single new line. In this case, a **delete** operation is subsumed.

The next screen alters the recently inserted line, changing it into three new lines.

```
:7 CR
Another line that doesn't rhyme
:change CR
ANOTHER LINE THAT DOESN'T RHYME
```

```
NOR DOES THIS
OR THIS
. CR
:6,10p CR
Tell that its sculptor well those passions read
ANOTHER LINE THAT DOESN'T RHYME
NOR DOES THIS
OR THIS
Which yet survive, stamp'd on these lifeless things,
:
```

In this way, a single line was turned into three lines. The next screen reverses the process, and translates the three new lines back into one.

```
:6,10p CR
Tell that its sculptor well those passions read
ANOTHER LINE THAT DOESN'T RHYME
NOR DOES THIS
OR THIS
Which yet survive, stamp'd on these lifeless things,
:7,9change CR
3 lines changed
. CR
:1,$p 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;
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 second screen changed the three recently added lines into nothing. This was done by specifying that the three lines were to be changed, and then specifying no text. The . command told **edit** that nothing should be put in the place of the existing three lines.

THE SEARCH FACILITIES

The first of the **edit** search systems locates a specified character string. This can be done by searching from the current line to the end of the file, or from the current line to the beginning of the file. Note that in all cases, the character string, called a *pattern*, must be on a single line. If a new line occurs within the pattern, it will not be located.

The commands for carrying out these simple searches are as follows

```
/pattern/p

?pattern?p
```

The first command, using the / characters searches for *pattern* forward through the file. The second command, using the ? characters, searches backward through the file. If the pattern does not exist, **edit** displays the message

```
Pattern not found
```

If the pattern is located, the line containing the pattern is displayed, and becomes the current line. Where the pattern is not found, the current line does not

change.

The pattern that is found is the first instance located. It may be required to repeat the search for further instances. In this case, the search can be repeated, using the commands // and ??. These tell **edit** to look for the next instance of the same pattern.

The next screen illustrates the use of these systems. The first command line makes line 7 of the poem the current line. The next command line attempts to search forward through the buffer (that is, through lines 7 to the end), for a pattern that in fact, does not exist. Next, the another search is carried out, this time for a pattern that does exist. Line 12 becomes the current line. The // command is used to repeat the search, but no further instances of the pattern exist.

```
:7 CR
Which yet survive, stamp'd on these lifeless things,
:/Ozymndias/p CR
Pattern not found
:/othi/p CR
Nothing beside remains. Round the decay
:.= CR
12
:// CR
Pattern not found
:
```

The next screen performs the same sort of sequence, this time with the ? command.

```
:7 CR
Which yet survive, stamp'd on these lifeless things,
:?xyz?p CR
Pattern not found
:?hatt?p CR
```

```
Half sunk, a shatter'd visage lies, whose frown
:.= CR
4
:?? CR
Pattern not found
:
```

Also available are the ^ and **$** characters. The first specifies a position at the beginning of a line, while **$** is used to specify the end of the line. The next screen shows these in use. The first command line makes line 1 the current line. The next uses the circumflex (^) to search for a pattern that must occur at the beginning of the line.

```
:1 CR
I met a traveller from an antique land
:/^T/p CR
Tell that its sculptor well those passions read
:?and$?p CR
And wrinkled lip and sneer of cold command
:?? CR
I met a traveller from an antique land
:
```

Note that the line containing the first instance of the pattern was printed, making line 6 the current line. The following command used the **$** character to search for a pattern occuring at the end of a line. The ? characters told **edit** to search backwards through the file. The first instance of **and** occuring at the end of a line occured in line 5, which became the current line. The next command repeated the search, and line 1 became the current line again. Note that line 3 does not match the conditions of the search: the line ends with the correct pattern followed by a single comma. This is enough to cause that line to be rejected.

The second type of search facility involves searching for all lines that contain or do not contain a specified pattern. These commands are **g** and **v** respectively.

The **g** command searches through the buffer for all lines containing the specified pattern. All lines found are displayed, and the current line becomes the last line displayed.

The **v** command performs the same operation, except that it seraches for all lines that do not contain the specified pattern. All of these lines are displayed.

The next screen illustrates these two commands in use.

```
:g/ar/p CR
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.
:v/ar/p CR
I met a traveller from an antique land
Who said: Two vast and trunkless legs of stone
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,
`My name is Ozymandias, king of kings:
Look on my works, ye Mighty, and despair!'
Nothing beside remains. Round the decay
:
```

THE SEARCH SPECIAL CHARACTERS

This section is a summary of the special search characters available to **edit** users. Some have already been encountered.

. the dot, used in this context, matches any *single* character, except the newline character. In the next screen, the search pattern **t.a** tells **edit** to search for an instance of the letter **t**, followed by any other character, then a letter **a**. Note that the global search found occurences of the unspecified middle character being a space. The character strings matching the pattern are highlighted for clarity.

```
:g/t.a/p CR
I met a traveller from an antique land
Who said: Two vast and trunkless legs of stone
Tell that its sculptor well those passions read
The hand that mock'd them and the heart that fed;
Of that colossal wreck, boundless and bare,
:
```

* the asterisk matches any group of repeated characters. For example, the search pattern **s*** would search the buffer for all lines containing the pattern **s**, followed by one or more other **s**'s. The following screen shows this command line in use.

```
:g/s*/p CR
Who said: Two vast and trunkless legs of stone
Tell that its sculptor well those passions read
Which yet survive, stamp'd on these lifeless things,
Of that colossal wreck, boundless and bare,
:
```

^ the circumflex searches the buffer for all lines *beginning* with the specified pattern. An example of this in use was given above.

$ the dollar sign searches the buffer for all lines *ending* with the specified pattern. An example of this has been given, above.

[] the square brackets are used to find a character that matches one of the characters in the brackets. In this way, the search pattern **[b-f]at** would match the any word beginning with the pattern **bat**, **cat**, **dat**, **eat** and **fat**. The following screen shows this sort of specification in use.

```
:g/[b-f]ar/p CR
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.
```

SUBSTITUTING TEXT: SUBSTITUTE

A varient of the **change** command is **substitute,** abbreviated to **s**. This has the effect of altering a number of specified characters rather than whole lines. The command takes the following form:

*s/old_text/new_text/***p**

where **s** is the abbreviated form of the **substitute** command, *old_text* is the text to be changed, in its existing form, and *new_text* is the new form of the text. The command ends with the **p** command, which tells **edit** to display the results of the substitution.

The next screen uses **substitute** to change the word **Ozymandias** into **Eric**. Note that the **change** command could have been used to perform this operation, but that it would have involved typing in the whole of the new line.

```
:10 CR
`My name is Ozymandias, king of kings:
:s/Ozymandias/Eric/p CR
`My name is Eric, king of kings:
:
```

As with the **change** command, **substitute** can be used to change existing text into nothing. The next screen illustrates this in use, but first, shows what happens if the first character string does not actually exist in the current line.

```
:s/Erix//p CR
Substitute pattern match failed
:s/Eric//p CR
`My name is , king of kings:
:
```

Due to a typing mistake, **edit** was asked to substitute a non-existant character string into nothing. An error message was displayed. Note that blank spaces are also characters, and can be manipulated by the **substitute** command in the same way as other characters. The next screen shows this in use.

```
:10 CR
`My name is , king of kings:
:s/ ,/    Ozymandias,/p
`My name is    Ozymandias, king of kings:
:
```

The last substitute operation changed a space followed by a comma into four spaces, a name and a comma. In order to reduce the spaces from four to one, the following command can be used.

```
:s/    / /p CR
`My name is Ozymandias, king of kings:
:
```

Note that a substitute operation implies a search operation, in that **edit** has to find the characters to be altered. In this way, the full range of the special search characters is available.

In addition, **edit** supplies the ampersand (**&**) as a special character. It can be used to add additional characters to an existing string of characters. The last line of the poem contains the phrase **far away**. It is required to change this to **far away into the distance**. The long way of doing this is as follows:

```
s/far away/far away into the distance/p
```

The command line in the next screen uses the ampersand to make this same change.

```
:$ CR
The lone and level sands stretch far away.
:s/far away/& into the distance/p CR
The lone and level sands stretch far away into the distance.
:
```

Where more than one case of the original character string exists in the current line, **edit** performs the substitution on the first. In the next screen, **substitute**

is used to change **a** into **A**. Although there are three **a**'s in this line, only the first is changed.

```
:10 CR
`My name is Ozymandias, king of kings:
:s/a/A/p CR
`My nAme is Ozymandias, king of kings:
:
```

In fact, **edit** supplies a version of the **substitute** command that will act upon all incidences of the character string, within a specified line or range of lines. This is the **global** substitution command, shortened to **g**. It has the following format:

g/*old_pattern*/**s**/*old_pattern*/*new_pattern*/**gp**

where the first part of the command, **g**/*old_pattern*/, tells **edit** to serach for each instance of the string *old_pattern*. The second part, **s**/*old_pattern*/*new_pattern*/, tells **edit** to change *old_pattern* into *new_pattern*. The third part, **gp**, tells **edit** to perform the substitution for every instance of the pattern, and to print the results. Without the final **g**, **edit** would change only the first instance of the pattern in each line.

```
:1 CR
I met a traveller from an antique land
:g/th/s/th/TH/gp CR
Stand in THe desert. Near THem on THe sand,
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;
And on THe pedestal THese words appear:
NoTHing beside remains. Round THe decay
```

```
Of THat colossal wreck, boundless and bare,
:
```

As a form of shorthand, it is not necessary to specify *old_pattern* twice, if the same pattern is to be used in the search and substitute operations. The following command line is the equivalent of that used in the last screen, above.

```
g/th/s//TH/gp
```

Global substitutions can be limited to a range of lines by entering the line numbers immediately before the command. The next screen turns the **TH** back into **th**, but only for the first seven lines of the poem.

```
:1,7g/TH/s//th/gp CR
Stand in the desert. Near them on the sand,
Tell that its sculptor well those passions read
Which yet survive, stamp'd on these lifeless things,
:
```

COPYING TEXT: COPY

The **copy** command, abbreviated to **co** (note that **c** is the abbreviation of **change**), can be used to make a copy an area of text, and to place it at a specified point within the buffer. Line numbers specified immediately before the **copy** command identify the lines to be copied, and a line number specified immediately after identify where the copy is to be put. The next screen shows lines 5 to 8 of the poem being copied to line 12.

```
:5,8copy12 CR
4 lines copied
The hand that mock'd them and the heart that fed;
:1,$p 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;
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
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;
Of that colossal wreck, boundless and bare,
The lone and level sands stretch far away.
:
```

Note that the end of the poem moved down, and that the text in lines 5 to 8 still exists. Line renumbering is carried out automatically by **edit**.

MOVING TEXT: MOVE

The **move** command, abbreviated to **m** is used to perform more or less the same operation as **copy**, except that the original text is deleted. Only the copy is left, in the new position.

MANIPULATING BUFFERS AND FILES

The **edit** buffer is used to store the work copy of the file until a permanent version is made. The **file** command, abbreviated to **f** is used to tell **edit** to report on how many lines are currently in the buffer, and on whether the contents of the buffer have been changed. In the next screen, line 5 is the current line.

```
:file CR
"shelley" line 5 of 14 --35%--
[Modified]
:
```

Up to 26 **edit** buffers can be used at a time. These are identifed by a single lower case letter, *a* through *z*. The next screen uses the **delete** command to remove 5 lines from the poem, and to place them in a buffer called *a*.

```
:1,5delete a CR
5 lines deleted
Tell that its sculptor well those passions read
:1,$p 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;
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.
:put a CR
```

```
5 lines puted
And wrinkled lip and sneer of cold command
1,$p CR
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;
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.
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
:
```

This sequence of commands takes the first five lines from the current buffer, and places them in *a*. They are then replaced, using the **put** command, after the current line. This means that the first five lines end up being the last five lines.

INSERTING FILES: READ

The **read** command, abbreviated to **r** can be used to insert the contents of an existing file into the current buffer, without overwriting the contents of the buffer. The new file is inserted immediately after the current line. In the example, a two line file called *insert.txt* is read into *shelley*, between lines 6 and 7. Note that a line and character count is given for the new file when **read** is executed. The first command line makes line 6 the current line.

```
:6 CR
```

```
Tell that its sculptor well those passions read
:read insert.txt CR
"insert.txt" 2 lines, 39 characters
:6,10p CR
Tell that its sculptor well those passions read
THIS IS 2 LINES OF
NEWLY INSERTED TEXT
Which yet survive, stamp'd on these lifeless things,
The hand that mock'd them and the heart that fed;
:
```

SAVING FILES: WRITE

The buffer (all, or in part) can be written to a new file with the **write** command, abbreviated to **w**. Using the **write** command does not exit from **edit**, and can therefore be used to make a number of different copies of the file as it develops.

Used without arguments, **write** saves the current buffer in the original file. In the next example, the file called *shelley* is saved.

```
:w CR
"shelley" 14 lines, 621 characters
:
```

The next screen uses the **w** command to write the buffer to a new file called *shelley2*.

```
:write shelley2 CR
"shelley2" [New file] 14 lines 621 characters
:
```

Note that the last two command lines have collectively created two copies of the same text, one called *shelley*, and the other called *shelley2*.

The next screen uses **write** to create a new file that contains only the last five line of the buffer.

```
:10,$write shelley3 CR
"shelley3" [New file] 5 lines, 209 characters
:
```

A stronger version is **write!**, which forces **edit** to overwrite the contents of any existing file with the same name as that specified in the **write** command. In the next screen, a file called *test* already exists, so the first command line returns an error message. The second command line succeeds in making a copy of the buffer, and calling it *test*.

```
:w test CR
"test" File exists - use "w! test" to overwrite
:w! test CR
"test" 14 lines, 621 characters
:
```

QUITTING EDIT

Once an editing session has been completed, the **quit** command is used to return to the X/OS shell. If amendments have been made to the current file, but no **write** command has been executed, **edit** displays a warning:

```
:quit CR
No write since last change (:quit! overrides)
:quit! CR
```

This warning is a reminder that all modifications will be lost if the new version of the file is not saved. The **quit!** command, abbreviated to **q!**, forces **edit** to terminate, abandoning any work done on the file since the beginning of the current editing session. This can be useful if the file has been undergone a number of unwanted changes during the current session, and going back to the previous version is the quickest way of remedying the errors.

MORE ABOUT THE EDIT COMMAND LINE

In the section called *Creating a New File*, it was mentioned that it is possible to activate **edit** without specifying a filename on the command line. The next screen shows what happens when this is done.

```
>edit CR
:append CR
This is a couple of lines of text designed to CR
illustrate what happens when no filename is given. CR
. CR
:write CR
No current filename
:write testfile CR
"testfile" 2 lines, 97 characters
:
```

When **edit** was typed, a new buffer was created. Text was then entered, using the **append** command. The **write** command was used to save the new text, but **edit** was unable to create a file containing this material, because no filename had been specified. The message is a reminder that a filename is needed. Once the filename was supplied, **edit** was able to create the file.

RECOVERING LOST TEXT

In addition to recovering text by reversing a command using **undo**, **edit** supplies a means of recovering from system crashes. In the event of a system crash occuring during an **edit** session, an emergency file save process will attempt to retain as much text as possible. The **mail** command should indicate the name of the emergency save file during the next login routine. The next stage is to return to the directory containing the file to be edited, and type **edit** followed by the filename supplied by **mail**. The **recover** command is then used, using the name of the save file. Note that this command cannot be abbreviated. In the next example, the filename returned by **mail** was *save.file*.

```
>edit save.file CR
"save.file" 14 lines, 621 characters
:recover save.file CR
```

It is possible that **edit** was uanble to save all the text contained in the buffer when the system crash occured. Accordingly, a check should be made on how much work was recovered.

If the editor itself develops a fault during the editing session, it is vital not to quit from **edit**. Instead, the **preserve** command should be used. This command may be abbreviated to **pre**. This activates the same emergency routines that are used during a system crash, as described above.

USING X/OS COMMANDS

At any time during an **edit** session, X/OS commands can be used. The normal command line for the utility concerned is typed, after a single shriek character (!). The utility is executed as normal, and after its completion, **edit** prints an ! on a new line, before displaying its normal prompt (:). The example shows the **ls** utility being used from within **edit**. This utility is described in the *User Guide*, and in the *ls*(1) entry of the *Utilities Reference Manual*.

```
:!ls CR
shelley     shelley2     testfile     test
!
:
```

If it is required to execute more than one command in a single command line, the **sh** command will temporarily return control to the default X/OS shell. A number of commands may then be executed. When it is required to return to **edit**, the **CTRL-d** key sequence is used to terminate the shell. The **edit** prompt will reappear.

COMMAND LIST

This last section supplies a list of the **edit** commands, and their abbreviations, where available.

COMMAND	ABBREVIATION	COMMAND	ABBREVIATION
!	none	list	l
&	none	move	m
''	none	preserve	pre
+	none	print	p
-	none	put	none
.	none	quit	q
.=	none	quit!	q!
//	none	read	r
??	none	recover	none
append	a	substitute	s
change	c	undo	u
copy	co	v	none
delete	d	write	w
file	f	write!	w!
g	none	z	none
insert	i		

EX: text editor

INTRODUCTION

This is a tutorial-style introduction to the X/OS **ex** utility, which is the basis of a whole family of X/OS editors, including **edit** (described in this manual), **vi** and **ed** (both described in the *User Guide*). It is not intended to describe the basic features of **ex** here: for a description of these, see the **edit** tutorial. Instead, useful additional features will be highlighted. For a complete formal description of the editor, see the *ex*(1) entry in the *Utilities Reference Manual*.

Instead of directly working on a specified file, **ex** makes a copy which resides in a temporary memory area called a *buffer*, and uses this for the editing session. This is then copied to a permanent file, while the original is retained as a backup. This system is described in more detail in the **edit** tutorial.

Again, like **edit**, **ex** supplies commands in the form of English words, many of which have abbreviated forms. Commands are entered on the **ex** command line, which is identified by a single colon (:).

Unless line numbers are entered as arguments, commands affect the current line. Line addressing may be absolute, or relative to the current line.

SYNTAX

```
ex [-] [-v] [-rfile] [+cmd] [-l] name ...
```

DESCRIPTION

The command line options and arguments are as follows:

- suppresses all interactive responses from **ex**.

-v calls up the **vi** editor.

-r*file* recovers *file* after an editor or system crash. If *file* is not specified, a list of all the files saved is displayed.

+*cmd* begins **ex** processing with the search or addressing command specified by *cmd*. If this option is not used, the default operation is performed, which is to open a file, and make the last line the current line.

-l generates the correct indenting for LISP code.

The *name* argument identifies the file to be edited.

The major additional features offered by **ex** over **edit** are the enhanced terminal capabilities, error handling functions, and text viewing operations. For coverage of the basic editing facilities, reference should be made to the **edit** tutorial in this manual. This chapter will build upon the material presented there.

EXAMPLES

This section describes the **ex** editor with a series of example screens. These show the commands to be entered, and the effect they have on a short text file. This file is called *shelley*, and contains a poem. Throughout, the > symbol 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. The colon in bold type which appears at the beginning of a line is the **ex** prompt.

CREATING A NEW FILE

The first screen begins by creating a new file. Its name is *shelley*.

```
>ex shelley CR



"shelley" [New file]
:
```

Note that **ex** immediately printed a message indicating that *shelley* is a new file, and then displayed the **ex** prompt. These may appear at the bottom of the screen.

It is possible to create a new file by typing only **ex** on the command line. In this case, the normal editing operations can be preformed, but on quitting, **ex** asks for a filename. This is described in the section below called *More About The ex Command Line*.

ENTERING TEXT: APPEND

At this stage, an empty file called *shelley* has been created. The next stage is to enter some text. This is done using the **append** command, which can be shortened to **a**. The examples will use the long version of commands for the first few times before switching to the abbreviated forms. A full list of the numerous available commands and abbreviations is available in the *ex*(1) entry of the *Utilities Reference Manual*.

```
>ex shelley CR



"shelley" [New file]
:append CR
```

The **append** command puts **ex** into *insert* mode. As soon as **CR** is pressed, a blank line is displayed, and **ex** is ready to accept text lines. The next screen shows the first few lines being added. Note that input lines end with the **CR** key.

```
"shelley" [New file]
:append CR
I met a traveller from an antique land CR
Who said: Two vast and trunkless legs of stone CR
```

LEAVING INSERT MODE: THE DOT

The first two lines of the poem have been entered. To leave insert mode, a single full stop or period is typed on a new line. This causes **ex** to display its prompt on a new line, and to wait for a new command.

```
"shelley" [New file]
:append CR
I met a traveller from an antique land CR
Who said: Two vast and trunkless legs of stone CR
. CR
:
```

In the next screen, insert mode is entered again, and the rest of the poem is entered. Note that **append** always adds text after the current line.

```
:append CR
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;
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.
. CR
:
```

FINDING OUT THE CURRENT LINE NUMBER

The current line is always the line pointed to by the editor. It is often known by the shorthand name ., and this name can be used to address the current line. It may be the case that a complex sequence of operations makes it difficult to remember the number of the current line. This can be discovered using the .= command. The next screen shows this in use.

```
:.= CR
14
:
```

INSERTING FILES: READ

The **read** command, abbreviated to **r** can be used to insert the contents of an existing file into the current buffer, without overwriting the contents of the buffer. The new file is inserted immediately after the current line. In the example, a two line file called *insert.txt* is read into *shelley*, between lines 6 and 7. Note that a line and character count is given for the new file when **read** is executed. The first command line makes line 6 of *shelley* the current line.

```
:6 CR
Tell that its sculptor well those passions read
:read insert.txt CR
"insert.txt" 2 lines, 39 characters
:6,10p CR
Tell that its sculptor well those passions read
THIS IS 2 LINES OF
NEWLY INSERTED TEXT
Which yet survive, stamp'd on these lifeless things,
The hand that mock'd them and the heart that fed;
:
```

SAVING FILES: WRITE

The buffer (all, or in part) can be written to a new file using the **write** command (**w**). This command does not exit from **ex**, and can therefore be used to make a number of different copies of the file as it develops.

Used without arguments, **write** saves the current buffer in the original file. In the next example, the file called *shelley* is saved.

```
:w CR
"shelley" 14 lines, 621 characters
:
```

The next screen uses the **w** command to write the buffer to a new file called *shelley2*.

```
:write shelley2 CR
"shelley2" [New file] 14 lines 621 characters
:
```

Note that the last two command lines have collectively created two copies of the same text, one called *shelley*, and the other called *shelley2*.

The next screen uses **write** to create a new file that contains only the last five line of the buffer.

```
:10,$write shelley3 CR
"shelley3" [New file] 5 lines, 209 characters
:
```

A stronger version is **write!**, which forces **ex** to overwrite the contents of any existing file with the same name as that specified in the **write** command. In the next screen, a file called *test* already exists, so the first command line returns an error message. The second command line succeeds in making a copy of the buffer, and calling it *test*.

```
:w test CR
"test" File exists - use "w! test" to overwrite
:w! test CR
"test" 14 lines, 621 characters
:
```

QUITTING EX

Once an editing session has been completed, the **quit** command is used to return to the X/OS shell. If amendments have been made to the current file, but no **write** command has been executed, **ex** displays a warning:

```
:quit CR
No write since last change (:quit! overrides)
:quit! CR

>
```

This warning is a reminder that all modifications will be lost if the new version of the file is not saved. The **quit!** command, abbreviated to **q!**, forces **ex** to terminate, abandoning any work done on the file since the beginning of the current editing session. This can be useful if the file has been undergone a number of unwanted changes during the current session, and going back to the previous version is the quickest way of remedying the errors.

MORE ABOUT THE EX COMMAND LINE

In the section called *Creating a New File*, it was mentioned that it is possible to activate **ex** without specifying a filename on the command line. The next screen shows what happens when this is done.

```
>ex CR
:append CR
This is a couple of lines of text designed to CR
illustrate what happens when no filename is given. CR
. CR
:write CR
No current filename
:write testfile CR
"testfile" 2 lines, 97 characters
:
```

When **ex** was typed, a new buffer was created. Text was then entered, using the **append** command. The **write** command was used to save the new text, but **ex** was unable to create a file containing this material, because no filename had been specified. The message is a reminder that a filename is needed. Once the filename was supplied, **ex** was able to create the file.

Note that **ex** supplies a quick way of saving a file and quitting the editor in one command line. This takes the following form:

```
:wq CR
```

DISPLAYING THE FILE

The facilities supplied for displaying the contents of a file are the same as those described in the **edit** tutorial. Reference should be made to that chapter of this manual.

MOVING AROUND THE FILE

The facilities supplied for moving from one line to another are the same as those described in the **edit** tutorial. Reference should be made to that chapter of this manual.

CHANGING THE CONTENTS OF THE FILE

The facilities supplied for changing the contents of a file are the same as those described in the **edit** tutorial. Reference should be made to that chapter of this manual.

FILE AND BUFFER HANDLING

The facilities supplied for handling buffers and files are generally the same as those described in the **edit** tutorial. Reference should be made to that chapter of this manual. In addition are the facilities described in the next two sections.

EDITING MULTIPLE FILES

Access can be gained to more than one file during a single session. The first method involves specifying more than one filename on the command line. The first file specified becomes the current file, but this can be changed using the **next** command (abbreviated to **n**). A list of the filenames originally specified can be obtained using the **args** command (abbreviated to **ar**). In the example below, *file2* is the current file when the **args** command is executed. The current file is enclosed in square brackets on the **args** output line.

```
>ex file1 file2 file3 CR
3 files to edit
"file1" 27 lines, 784 characters
:next CR
"file2" 103 lines, 6649 characters
:args CR
file1 [file2] file3
:
```

The second method of switching to another file is to have specified only one filename on the command line, but to then use the **edit** command (abbreviated to **e**). The file being edited is the current file, while a second file called up is the alternative file. In the example, the file called *shelley* is being edited, and is therefore the current file. The name of the current file is displayed using the **file** command. The **edit** command is then used to edit a file called *test6*. This command makes *test6* the current file, and *shelley* the alternative file.

```
:file CR
"shelley" line 14 of 14 --100%--
[Modified]
:edit test6 CR
"test6" 270 lines, 9164 characters
```

```
:file CR
"test6" line 270 of 270 --100%--
:
```

To return to the first file, the notation **#** can be used. This is short-hand for the name of the alternative file.

```
:edit # CR
"shelley" 14 lines, 621 characters
:
```

In addition to the **#** notation, filenames may be specified using the full range of X/OS shell filename expansion metacharacters.

OPEN/VISUAL MODE

When it is required to display a file's contents without making changes, the **visual** (abbreviated to **vi**) or **open** (abbreviated to **o**) commands can be used. This mode may be terminated using the **Q** command or by typing ^.

USING X/OS COMMANDS

The facilities supplied for executing an X/OS shell command from within **ex** are the same as those described in the **edit** tutorial. Reference should be made to that chapter of this manual.

RECOVERING LOST TEXT

The facilities supplied for recovering the contents of a file after a system or editor crash are the same as those described in the **edit** tutorial. Reference should be made to that chapter of this manual.

THE MESSAGE AND NEWS SYSTEMS

INTRODUCTION

This chapter supplies tutorials to cover the various X/OS utilities for the transmission of messages and news items. The tutorials supplied are as follows:

MAILX interactive message processing

NEWS news printing system

WRITE writing to other users

The tutorials are presented in alphabetical order.

MAILX: interactive message processing utility

INTRODUCTION

This is a tutorial-style introduction to the X/OS **mailx** message processing utility, which is an enhanced version of the **mail** utility. This is covered in its own tutorial, in the *User Guide*.

Many of the options to **mailx** are not available to **mail**. For example, it is possible to define an alias for a single user or for a whole group of users. This will send mail to an individual using a name or word other than their login name, or send mail to a whole group of people using a single name or word. A range of facilities exist for moving, editing and deleting messages.

Incoming mail is stored in a standard user file called *mailbox*. Once read, messages are passed to a standard secondary user file called *mbox*. Once received, messages can be amended and passed to other users. They are retained in *mbox* until specifically removed.

The **mailx** environment variables can be used to develop a personalised user environment, to suit individual tastes.

If entered with one or more login names as arguments, **mailx** interprets this as a request to send mail to the named users. A prompt asks for a summary of the subject, and then waits for the message to be entered, or a command to be issued. The section entitled *How to Send Messages* describes the facilities that are available for editing, incorporating other files, adding names to copy lists, and more.

If you enter the **mailx** command with no arguments, **mailx** checks incoming mail for you in the file named */usr/mail/name*, where *name* is your login name. If there is mail for you in that file, you are shown a list of the items and given the opportunity to read, store, remove or

transfer each one to another file. The section entitled *How to Manage Incoming Mail* provides some examples and describes the options available.

If you choose to customize **mailx**, you should create a start-up file in your home directory called *.mailrc*. The section called *The .mailrc File* describes variables you can include in your start-up file.

mailx has two modes of functioning: input mode and command mode. You must be in input mode to create and send messages. Command mode is used to read incoming mail. You can use any of the following methods to control the way **mailx** works for you:

- by entering options on the command line. (See the *mailx*(1) entry in the *Utilities Reference Manual*.)

- by issuing commands when you are in input mode, for example, creating a message to send. These commands are always preceded by a ~ (tilde) and are referred to as tilde escapes. (See the *mailx*(1) manual page in the *Utilities Reference Manual*)

- by issuing commands when you are in command mode, for example, reading incoming mail.

- by storing commands and environment variables in a start-up file in your home directory called **$HOME**/*.mailrc*.

Tilde escapes are discussed in *Sending Messages*, command mode commands in *Managing Incoming Mail*, and the *.mailrc* file in *The .mailrc File*.

SYNTAX

```
mailx [options] [name...]
```

DESCRIPTION

The options are flags that control the action of the command, and *name* represents the intended recipients. This may be one or more login names or aliases.

Anything on the command line other than an option preceded by a hyphen is read by **mailx** as a *name*; that is, the login or alias of a person to whom you are sending a message.

The options available to **mailx** are as follows.

-e tests for the presence of mail. Nothing is printed, but if mail is present, **mailx** returns a successful exit code.

-f [*file*] allows you to read messages from *file* instead of your mailbox. If no *file* is specified, **mailx** reads from the secondary storage file, *mbox*.

-F records the outgoing message in a file with the name of the first recipient. Therefore, if the message is to be sent to a user called **spike**, the file will be called *spike*.

-H only prints the header summary of the message.

-n do not initialize from the system default file *mailx.rc*. If you have your own

.mailrc file (see *The .mailrc File*), **mailx** will not look through the system default file for specifications when you use the **-n** option, but will go directly to your *.mailrc* file. This results in faster initialization; substantially faster when the system is busy.

-N do not print the header summary of the message.

-r*address* pass *address* to the message transmission routines. All tilde escapes (see below) are disabled.

-s *subject* sets the subject part of the message's header field to *subject*.

-u *user* reads *user*'s *mailbox* file. This option will not succeed if *user*'s *mailbox* is read-protected.

There are, in fact, other options to **mailx**. For details of these, see the *mailx*(1) entry of the *Utilities Reference Manual*.

EXAMPLES

This section illustrates **mailx** in use. Throughout, sample screens will be given, showing the **mailx** command lines, and the responses generated. The screens use the normal manual conventions: the **>** symbol in bold type represents the X/OS system prompt, while **CR** indicates that the carriage return key should be pressed in order to enter the command line.

HOW TO SEND MESSAGES: THE TILDE ESCAPES

The first example shows begins with sending a message to a user called **spike**. how you can edit messages you are sending, incorporate existing text into your messages, change the header information, and perform other tasks that take advantage of the **mailx** command's capabilities. Each example is followed by an explanation of the key points illustrated in the sample screen.

The login name specified belongs to the person who is to receive the message. The system puts you into input mode and prompts you for the subject of the message. (You may have to wait a few seconds for the **Subject:** prompt if the system is very busy.) This is the simplest way to run the **mailx** command; it differs very little from the way you run the **mail** command. For details of **mail**, see the *User Guide*.

```
>mailx spike CR
Subject:
```

Whether to include a subject or not is optional. If you elect not to, press the **CR** key. The cursor moves to the next line and the program waits for you to enter the text of the message.

```
>mailx spike CR
Subject: meeting CR
We're having a meeting for novice mailx users in CR
the auditorium at 9:00 tomorrow. CR
Would you be willing to give a demonstration? CR
Bob CR
~. CR
cc: CR
```

There are two important things to notice about the above example:

- You break up the lines of your message by pressing the **CR** key at the end of each line. This makes it easier for the recipient to read the message, and prevents you from overflowing the line buffer.

- You end the text and send the message by entering a tilde and a period together (~.) at the beginning of a line. The system responds with an end-of-text notice (EOT) and a prompt.

The **cc:** prompt at the bottom is requesting the names of users to receive carbon copies. This is described later.

There are several commands available to you when you are in input mode (as we were in the example). Each of them consists of a tilde (~), followed by an alphabetic character, entered at the beginning of a line. Together they are known as tilde escapes. (See the *mailx*(1) manual page in the *Utilities Reference Manual* for a full list.) Most of them are used in the examples in this section.

You can include the subject of your message on the command line by using the **-s** option, see above. For example, the command line:

```
>mailx -s "meeting" spike CR
```

is equivalent to:

```
>mailx spike CR
Subject: meeting CR
```

The subject line will look the same to the recipient of the message. Notice that when putting the subject on the command line, you must enclose a subject that has more than one word in quotation marks.

EDITING THE MESSAGE

When you are in the input mode of **mailx**, you can invoke an editor by entering the ~e (tilde e) escape at the beginning of a line. The following example shows how to use tilde:

```
>mailx spike CR
Subject: Testing my tilde CR
When entering the text of a message CR
that has somehow gotten grabled CR
you may invoke your favorite editor CR
by means of a ~e (tilde e). CR
.
.
.
```

Notice that you have misspelled a word in your message. To correct the error, use ~e to invoke the editor, in this case the default editor, **ed**. The **ed** utility is described in the *User Guide*.

```
.
.
.
~e CR
12
/grabled/p CR
that has somehow gotten grabled
s/gra/gar/p CR
that has somehow gotten garbled
w CR
```

```
132
q CR
(continue)
What more can I tell you? CR
.
.
.
```

In this example the **ed** editor was used. Your *.profile* or a *.mailrc* file controls which editor will be invoked when you issue a ~**e** escape command. The ~**v** (tilde v) escape invokes an alternate editor (most commonly, **vi**).

When you exited from **ed** (by typing **q**), the **mailx** command returned you to input mode and prompted you to continue your message. At this point you may want to preview your corrected message by entering a ~**p** (tilde p) escape. The ~**p** escape prints out the entire message up to the point where the ~**p** was entered. Thus, at any time during text entry, you can review the current contents of your message.

```
~p CR
Message contains:
To: spike
Subject: Testing my tilde

When entering the text of a message
that has somehow gotten garbled
you may invoke your favorite editor
by means of a tilde e (~e).
What more can I tell you?
(continue)
~. CR
EOT

>
```

INCORPORATING EXISTING TEXT

mailx provides four ways to incorporate material from another source into the message you are creating. You can:

- read a file into your message
- read a message you have received into a reply
- incorporate the value of a named environment variable into a message
- execute a shell command and incorporate the output of the command into a message

The following examples show the first two of these functions. These are the most commonly used of these four functions. For information about the other two, see the *mailx*(1) manual page of the *Utilities Reference Manual*.

READING A FILE INTO A MESSAGE

```
>mailx spike CR
Subject: Work Schedule CR
As you can see from the following CR
~r letters/file1 CR
"letters/file1"   10/725
we have our work cut out for us. CR
Please give me your thoughts on this. CR
- Bob CR
~. CR
EOT

>
```

As the example shows, the ~r (tilde r) escape is followed by the name of the file you want to include. The system displays the file name and the number of lines and characters it contains. You are still in input mode and can continue with the rest of the message. When the recipient gets the message, the text of *letters/file1* is included. (You can, of course, use the ~p (tilde p) escape to preview the contents before sending your message.)

There is a varient of ~r that allows the output from an X/OS shell command to be inserted into the message. This takes the form of ~<!*cmd*, where the output from the shell command *cmd* is used. These are described in the *mailx*(1) entry of the *Utilities Reference Manual*.

INCORPORATING A MESSAGE FROM THE MAILBOX INTO A REPLY

```
>mailx CR
mailx version 2.14  2/9/85  Type ? for help.
"usr/mail/roberts": 2 messages 1 new
>N   1 abc        Tue May 1  08:09  8/155  Meeting Notice
     2 hqtrs      Mon Apr 30 16:57  4/127  Schedule
? m jones CR
Subject: Hq Schedule CR
Here is a copy of the schedule from headquarters... CR
~f 2 CR
Interpolating: 2
(continue)
As you can see, the boss will be visiting our district on CR
the 14th and 15th. CR
- Robert CR
~. CR
EOT
?
```

There are several important points illustrated in this example:

- The sequence begins in command mode, where you read and respond to your incoming mail. Then you switch into input mode by issuing the command **m jones** (meaning send a message to jones).

- The ~f escape is used in input mode to call in one of the messages in your mailbox and make it part of the outgoing message. The number 2 after the ~f means message 2 is to be interpolated (read in).

- **mailx** tells you that message 2 is being interpolated and then tells you to continue.

- When you finish creating and sending the message, you are back in command mode, as shown by the ? prompt. You may now do something else in command mode, or exit **mailx** by typing **q**.

An alternate command, the ~m (tilde m) escape, works the way that ~f does except the read-in message is indented one tab stop. Both the ~m and ~f commands work only if you start out in command mode and then enter a command that puts you into input mode. Other commands that work this way will be covered in the section *How to Manage Incoming Mail*.

CHANGING PARTS OF THE MESSAGE HEADER

The header of a **mailx** message has four components:

- subject

- recipient(s)

- copy-to list

~ blind-copy list (a list of intended recipients that is not shown on the copies sent to other recipients)

When you enter the **mailx** command followed by a login or an alias you are put into input mode and prompted for the subject of your message. Once you end the subject line by pressing the **CR** key, **mailx** expects you to type the text of the message. If, at any point in input mode, you want to change or supplement some of the header information, there are four tilde escapes that you can use: **~h**, **~t**, **~c**, and **~b**.

~h displays all the header fields: subject, recipient, copy-to list, and blind copy list, with their current values. You can change a current value, add to it, or, by pressing the **CR** key, accept it.

~t lets you add names to the list of recipients. Names can be either login names or aliases.

~c lets you create or add to a copy-to list for the message. Enter either login names or aliases of those to whom a copy of the message should be sent.

~b lets you create or add to a blind-copy list for the message.

All tilde escapes must be in the first position on a line. For the **~t**, **~c** or **~b**, any additional material on the line is taken to be input for the list in question. Any additional material on a line that begins with a **~h** is ignored.

ADDING A SIGNATURE

If you want, you can establish two different signatures with the **sign** and **Sign** environment variables. These can be invoked with the ~**a** (tilde a) or ~**A** (tilde A) escape, respectively. Assume you have set the value Supreme Commander to be called by the ~**A** escape. Here's how it would work:

```
>mailx -s orders all CR
Be ready to move out at 0400 hours. CR
~A CR
Supreme Commander
~. CR
EOT

>
```

Having both escapes (~**a** and ~**A**) allows you to set up two forms for your signature. However, because the sender's login automatically appears in the message header when the message is read, no signature is required to identify you.

KEEPING A RECORD OF MESSAGES SENT

The **mailx** command offers several ways to keep copies of outgoing messages. Two that you can use without setting any special environment variables are the ~**w** (tilde w) escape and the -F option on the command line.

The ~**w** followed by a file name causes the message to be written to the named file. For example:

```
>mailx bdr CR
Subject: Saving Copies CR
When you want to save a copy of CR
```

```
the text of a message, use the tilde w. CR
~w savemail CR
"savemail" 2/71
~. CR
EOT

>
```

If you now display the contents of *savemail*, you will see this:

```
>cat savemail CR
When you want to save a copy of
the text of a message, use the tilde w.

>
```

The drawback to this method, as you can see, is that none of the header information is saved.

Using the -F option on the command line does preserve the header information. It works as follows:

```
>mailx -F -s Savings bdr CR
This method appends this message to a CR
file in my current directory named bdr. CR
~. CR
EOT

>
```

We can check the results by looking at the file *bdr*.

```
>cat bdr CR
From: kol  Fri May 2  11:14:45  1986
```

```
To: bdr
Subject: Savings

This method appends this message to a
file in my current directory named bdr.

>
```

The -F option appends the text of the message to a file named after the first recipient. If you have used an alias for the recipient(s) the alias is first converted into the appropriate login(s) and the first login is used as the file name. As noted above, if you have a file by that name in your current directory, the text of the message is appended to it.

EXITING FROM MAILX

When you have finished composing your message, you can leave **mailx** by typing any of the following three commands:

~. tilde period (**~.**) is the standard way of leaving input mode. It also sends the message. If you entered input mode from the command mode of **mailx**, you now return to the command mode (as shown by the **?** prompt you receive after typing this command). If you started out in input mode, you now return to the shell (as shown by the shell prompt).

~q tilde q (**~q**) simulates an interrupt. It lets you exit the input mode of **mailx**. If you have entered text for a message, it will be saved in a file called *dead.letter* in your home directory.

~x tilde x (**~x**) simulates an interrupt. It lets you exit the input mode of **mailx** without saving anything.

In the preceding paragraphs we have described and shown examples of some of the tilde escape commands available when sending messages via the **mailx** command. (See the *mailx*(1) manual page in the *Utilities Reference Manual*.)

HOW TO MANAGE INCOMING MAIL

mailx has over fifty commands which help you manage your incoming mail. See the *mailx*(1) manual page in the *Utilities Reference Manual* for a list of all of them (and their synonyms) in alphabetic order. The most commonly used commands (and arguments) are described in the following subsections:

- the *msglist* argument
- commands for reading and deleting mail
- commands for saving mail
- commands for replying to mail
- commands for getting out of **mailx**

THE MSGLIST ARGUMENT

Many commands in **mailx** take a form of the *msglist* argument. This argument provides the command with a list of messages on which to operate. If a command expects a *msglist* argument and you do not provide one, the command is performed on the current message. Any of the following formats can be used for a *msglist*:

n	message number *n* the current message
^	the first undeleted message
$	the last message

`*` all messages

n-m an inclusive range of message numbers

user all messages from *user*

/string All messages with *string* in the subject line (case is ignored)

:c all messages of type *c* where *c* is:

d deleted messages

n new messages

o old messages

r read messages

u unread messages

The context of the command determines whether this type of specification makes sense.

Here are two examples (the ? is the command mode prompt):

```
?d 1-3 CR      [ Delete messages 1, 2 and 3 ]
?s bdr bdr CR  [ Save all messages from user bdr in a
                          file named bdr. ]
?
```

Additional examples may be found throughout the next three subsections.

COMMANDS FOR READING AND DELETING MAIL

When a message arrives in your mailbox the following notice appears on your screen:

```
you have mail
```

The notice appears when you log in or when you return to the shell from another procedure.

READING MAIL

To read your mail, enter the **mailx** command with or without arguments. Execution of the command places you in the command mode of **mailx**. The next thing that appears on your screen is a display that looks something like this:

```
mailx version 2.14  10/19/86    Type ? for help
"/usr/mail/bdr":   3 messages  3 new
> N 1 rbt          Thur Apr 30 14:20   8/190  Review Session
  N 2 admin        Thur Apr 30 15:56   5/84   New printer
  N 3 daves        Fri  May  1 08:39  64/1574 Reorganization
?
```

The first line identifies the version of **mailx** used on your system, displays the date, and reminds you that help is available by typing a question mark (**?**). The second line shows the path name of the file used as input to the display (the file name is normally the same as your login name) together with a count of the total number of messages and their status. The rest of the display is header information from the incoming messages. The messages are numbered in sequence with the last one received at the bottom of the list. To the left of the numbers there may be a status indicator; N for new, U for

unread. A greater than sign (>) points to the current message. Other fields in the header line show the login of the originator of the message, the day, date and time it was delivered, the number of lines and characters in the message, and the message subject. The last field may be blank.

When the header information is displayed on your screen, you can print messages either by pressing the **CR** key or entering a command followed by a *msglist* argument. If you enter a command with no *msglist* argument, the command acts on the message pointed at by the > sign. Pressing the **CR** key is the equivalent of a typing the **p** (for print) command without a *msglist* argument; the message displayed is the one pointed at by the > sign. To read some other message (or several others in succession), enter a **p** (for print) or **t** (for type) followed by the message number(s). Here are some examples:

```
? CR          [ Print the current message. ]
? p 2 CR      [ Print message number 2.    ]
? p spike CR  [ Print all messages from user spike. ]
```

The command **t** (for type) is a synonym of **p** (for print).

SCANNING THE MAILBOX

The **mailx** command lets you look through the messages in your mailbox while you decide which ones need your immediate attention.

When you first enter the **mailx** command mode, the banner tells you how many messages you have and displays the header line for twenty messages. (If you are dialed into the computer system, only the header lines for ten messages are displayed.) If the total number of messages exceeds one screenful, you can display the next screen by entering the **z** command. Typing **z-** causes a previous

screen (if there is one) to be displayed. If you want to see the header information for a specific group of messages, enter the **f** (for from) command followed by the *msglist* argument.

Here are examples of those commands:

```
? z CR          [ Scroll forward one screenful of header lines. ]
? z- CR         [ Scroll backward one screenful. ]
? f spike CR    [ Display headers of all messages from user spike. ]
```

SWITCHING TO OTHER MAIL FILES

When you enter **mailx** by issuing the command:

```
>mailx CR
```

you are looking at the file */usr/mail/your_login*.

mailx lets you switch to other mail files and use any of the **mailx** commands on their contents. (You can even switch to a non-mail file, but if you try to use **mailx** commands you are told **No applicable messages**.) The switch to another file is done with the **fi** or **fold** command (they are synonyms) followed by the *filename*. The following special characters work in place of the *filename* argument:

%	the current mailbox
%*login*	the mailbox of the owner of *login* (if you have the required permissions)
#	the previous file

& the current mbox

Here is an example of how this might look on your screen:

```
>mailx CR

mailx version 2.14 10/19/86  Type ? for help.
"usr/mail/spike":  3 messages 2 new 3 unread
  U 1 jaf         Sat May 9 07:55   7/137   test25
> N 2 todd        Sat May 9 08:59   9/377   UNITS requirements
  N 3 has         Sat May 9 11:08  29/1214  access to bailey

? fi &          [ Enter this command to transfer to your mbox. ]

Held 3 messages in /usr/mail/spike
"/fs1/spike/mbox":  74 messages 10 unread
  .
  .
  .
? q CR

>
```

DELETING MAIL

To delete a message, enter a **d** followed by a *msglist* argument. If the *msglist* argument is omitted, the current message is deleted. The messages are not deleted until you leave the mailbox file you are processing. Prior to that, the **u** (for undelete) gives you the opportunity to change your mind. Once you have issued the quit command (**q**) or switched to another file, however, the deleted messages are gone.

mailx permits you to combine the delete and print command and enter a **dp**. This is like saying, *Delete the message I just read and show me the next one.* Here are some examples of the delete command:

```
? d * CR     [ Delete all my messages. ]
? d r CR     [ Delete all messages that have been read. ]
? dp CR      [ Delete the current message and print the next one. ]
? d 2-5 CR   [ Delete messages 2 through 5. ]
```

COMMANDS FOR SAVING MAIL

All messages not specifically deleted are saved when you quit **mailx**. Messages that have been read are saved in a file in your home directory called *mbox*. Messages that have not been read are held in your mailbox (*/usr/mail/your_login*).

The command to save messages comes in two forms: with an upper case or a lower case **s**. The syntax for the upper case version is:

S [*msglist*]

Messages specified by the *msglist* argument are saved in a file in the current directory named for the author of the first message in the list.

The syntax for the lower case version is:

s [*msglist*] [*filename*]

Messages specified by the *msglist* argument are saved in the file named in the *filename* argument. If you omit the *msglist* argument, the current message is saved. If you are using logins for file names, this can lead to some ambiguity. If **mailx** is puzzled, you will get an error message.

COMMANDS FOR REPLYING TO MAIL

The command for replying to mail comes in two forms: with an upper case or a lower case **r**. The principal difference between the two forms is that the upper case form (**R**) causes your response to be sent only to the originator of the message, while the lower case form (**r**) causes your response to be sent not only to the originator but also to all other recipients. (There are other differences between these two forms. For details, see the *mailx*(1) manual page in the *Utilities Reference Manual*.)

When you reply to a message, the original subject line is picked up and used as the subject of your reply. Here's an example of the way it looks:

```
>mailx CR

mailx version 2.14 10/19/83  Type ? for help.
"usr/mail/spike":  3 messages 2 new 3 unread
  U 1 jaf         Wed May 9 07:55   7/137   test25
> N 2 todd        Wed May 9 08:59   9/377   UNITS requirements
  N 3 has         Wed May 9 11:08  29/1214  access to bailey

? R 2 CR
To: todd
Subject: Re: UNITS requirements
```

Assuming the message about UNITS requirements had been sent to some additional people, and the lower case **r** had been used, the header might have appeared like this:

```
? r 2 CR
To: todd eg has jcb bdr
Subject: Re: UNITS requirements
```

COMMANDS FOR GETTING OUT OF MAILX

There are two standard ways of leaving **mailx**: with a **q** or with an **x**. If you leave **mailx** with a **q**, you see messages that summarize what you did with your mail. They look like this:

```
? q CR
Saved 1 message in /fsl/bdr/mbox
Held 1 message in /usr/mail/bdr

CR
```

From the example we can surmise that user *bdr* had at least two messages, read one and either left the other unread or issued a command asking that it be held in */usr/mail/bdr*. If there were more than two messages, the others were deleted or saved in other files. **mailx** does not issue a message about those.

If you leave **mailx** with an **x**, it is almost as if you had never entered. Mail read and messages deleted are retained in your mailbox. However, if you have saved messages in other files, that action has already taken place and is not undone by the **x**.

MAILX COMMAND SUMMARY

In the preceding subsections we have described some of the most frequently used **mailx** commands. (See the *mailx*(1) manual page in the *Utilities Reference Manual* for a complete list.) If you need help while you are in the command mode of **mailx**, type either a **?** or **help** after the **?** prompt. A list of **mailx** commands and what they do will be displayed on your terminal screen.

THE .mailrc FILE

The *.mailrc* file contains commands to be executed when you invoke **mailx**.

There may be a system-wide start-up file (*/usr/lib/mailx/mailx.rc*) on your system. If it exists it is used by the system administrator to set common variables. Variables set in your *.mailrc* file take precedence over those in *mailx.rc*.

Most **mailx** commands are legal in the *.mailrc* file. However, the following commands are NOT legal entries:

! or **shell**	escape to the shell
Copy	save messages in *msglist* in a file whose name is derived from the author
edit	invoke the editor
visual	invoke the **vi** editor. This is described in the *User Guide*, and in the *vi*(1) entry of the *Utilities Reference Manual*.
followup	respond to a message
Followup	respond to a message, sending a copy to *msglist*
mail	switch into input mode
reply	respond to a message
Reply	respond to the author of each message in *msglist*

You can create your own *.mailrc* with any editor, or copy a friend's. A sample *.mailrc* file is shown in the diagram. *.mailrc* file:

```
if r
        cd $HOME/mail
endif
set allnet append asksub askcc autoprint dot
set metoo quiet save showto header hold keep keepsave
set outfolder
set folder='mail'
set record='outbox'
set crt=24
set EDITOR='/bin/ed'
set sign='Roberts'
set Sign='Jackson Roberts, Supervisor'
set toplines=10
alias fred      fjs
alias bob       rcm
alias alice     ap
alias mark      mct
alias donna     dr
alias pat       pat
group robertsgrp        fred bob alice pat mark
group accounts  robertsgrp donna
```

The example in the diagram includes the commands you are most likely to find useful: the **set** command and the **alias** or **group** commands.

The **set** command is used to establish values for environment variables. The command syntax is:

set
set *name*
set *name* = *string*
set *name* = *number*

When you issue the **set** command without any arguments, **set** produces a list of all defined variables and their values. The argument *name* refers to an environmental

variable. More than one *name* can be entered after the **set** command. Some variables take a string or numeric value. String values are enclosed in single quotes.

When you put a value in an environment variable by making an assignment such as **HOME**=*my_login*, you are telling the shell how to interpret that variable. However, this type of assigment in the shell does not make the value of the variable accessible to other X/OS system programs that need to reference environment variables. To make it accessible, you must export the variable, as shown in the following example:

```
>TERM=5425 CR

>export TERM CR
```

When you export variables from the shell in this way, programs that reference environment variables are said to import them. Some of these variables (such as **EDITOR** and **VISUAL**) are not peculiar to **mailx**, but may be specified as general environment variables and imported from your execution environment. If a value is set in *.mailrc* for an imported variable it overrides the imported value. There is an **unset** command, but it works only against variables set in *.mailrc*; it has no effect on imported variables.

There are forty-one environment variables that can be defined in your *.mailrc*; too many to be fully described in this document. For complete information, consult the *mailx*(1) manual page in the *Utilities Reference Manual*.

Three variables used in the diagram deserve special attention because they demonstrate how to organize the filing of messages. These variables are: **folder**, **record**, and **outfolder**. All three are interrelated and control the directories and files in which copies of messages are kept.

To put a value into the **folder** variable, use the following format:

set folder=*directory*

This specifies the directory in which you want to save standard mail files. If the directory name specified does not begin with a / (slash), it is presumed to be relative to **$HOME**. If **folder** is an exported shell variable, you can specify file names (in commands that call for a *filename* argument) with a / before the name; the name will be expanded so that the file is put into the **folder** directory.

To put a value in the **record** variable, use the following format:

set record=*filename*

This directs **mailx** to save a copy of all outgoing messages in the specified file. The header information is saved along with the text of the message. By default, this variable is disabled.

The **outfolder** variable causes the file in which you store copies of outgoing messages (enabled by the variable **record**=) to be located in the *folder* directory. It is established by being named in a **set** command. The default is **nooutfolder**.

The **alias** and **group** commands are synonyms. In the diagram, the **alias** command is used to associate a name with a single login; the **group** command is used to specify multiple names that can be called in with one pseudonym. This is a nice way to distinguish between single and group aliases, but if you want, you can treat the commands as exact equivalents. Notice, too, that aliases

can be nested.

In the *.mailrc* file shown in the diagram the alias **robertsgroup** represents five users; three of them are specified by previously defined aliases and one is specified by a login. The fifth user, **pat**, is specified by both a login and an alias. The next group command in the example, **accounts**, uses the alias **robertsgroup** plus the alias **donna**. It expands to twelve logins.

The *.mailrc* file in the diagram includes an **if-endif** command. The full syntax of that command is:

```
if s|r mail_commands

    else mail_commands

endif
```

The **s** and **r** stand for send and receive, so you can cause some initializing commands to be executed according to whether **mailx** is entered in input mode (send) or command mode (receive). In the preceding example, the command is issued to change directory to **$HOME**/*mail* if reading mail. The user in this case had elected to set up a subdirectory for handling incoming mail.

The environment variables shown in this section are those most commonly included in the *.mailrc* file. You can, however, specify any of them for one session only whenever you are in command mode. For a complete list of the environment variables you can set in **mailx** see the *mailx*(1) manual page in the *Utilities Reference Manual*.

NEWS: news print utility

INTRODUCTION

This is a tutorial-style introduction to the X/OS **news** utility, which is used to broadcast general interest messages. Message files are stored in the directory */usr/news*.

SYNTAX

---

news [-a] [-n] [-s] [items]

DESCRIPTION

Whenever messages are read, **news** creates an empty file called *.news_time*, in the user's home directory. Whenever **news** is executed without arguments, all messages more recent than the modification time of *.news_time* are printed. Messages are printed in order of increasing age, most recent first. Each is preceded by an identifying header. After the messages have been read, *.news_time* is updated, and its new modification date again determines whether messages are current.

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

-a prints out all messages stored in */usr/news*, irrespective of currency. The modification date *.news_time* is not updated.

-n prints out the names of the current messages, without printing their contents, and without changing the modification date of *.news_time*.

-s prints out the number of current messages stored in */usr/news*, without printing their contents, and without changing the modification date of *.news_time*. Note that it may be useful to include a **news** command line using this option in the *.profile* file.

items specifies the names of message files stored in */usr/news*, to be read by the user.

Messages can be skipped by pressing the **DELETE** key once. The next message is displayed. Pressing **DELETE** twice in quick succession terminates the **news** program.

WRITE: write to another user

INTRODUCTION

This is a tutorial-style introduction to the X/OS **write** utility, which communicates between users. When invoked, **write** accepts lines of text or input files which are then sent to the specified user. The receiving user establishes a two-way connection by responding with the **write** command. Communications continue until the file is completed, or until one user types **CTRL-d** for End of Transmission.

SYNTAX

```
write user [line]
```

DESCRIPTION

The *user* argument takes the form of a login name to identify the user to recive the message. Where that user is logged on to more than one terminal, the optional *line* argument is used to identify which terminal should receive the message. A line specification takes the form of the line identifier or the terminal number (for example, **tty32**). If this argument is not given, **write** checks the entries in a file called */etc/utmp*, and the first entry relating to the destination user is used, and the following is displayed on the screen:

user is logged on more than one place.
You are connected to *terminal*.
Other locations are:
terminal

When connection is made, the following appears on the receiving user's terminal:

```
Message from source_user (ttyxx) [date] ...
```

The terminal bell sounds as an additional warning. The receiving user would then type

```
write source_user
```

and the connection would be made. The convention when talking, is to use the symbol **o** for *over* at the end of an input session, and **oo** for *over and out* at the end of a conversation. **CTRL-d** is then pressed, and the message **EOT** for End of Transmission appears.

If the receiving user is not logged on, **write** displays the message

```
User not logged in
```

EXAMPLES

This section illustates **write** in action, by setting up a dialogue between two users, called **spike** and **sue**. The **>** symbol in bold type represents the X/OS system prompt, while **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>write sue CR
```

User **sue** receives the warning message, and responds:

```
>
Message from spike on tty12 at 13:44 ...
CR
>write spike CR
Good morning! CR
o CR
```

User **spike**'s screen will display a similar message, and the first two lines of **sue**'s response:

```
>write sue tty15 CR
Message from sue on tty15 at 13:44 ...
Good morning!
o
Good morning to you too! CR
oo CR
CTRL-d
EOT

>
```

User **spike**'s reply is followed by *over and out*, then **CTRL-d**. This returns **spike** to the X/OS system prompt, and terminates the connection.

INTRODUCTION

This chapter covers the various X/OS networking utilities, with especial reference to the **UUCP** system. The utilities covered are as follows:

CU calls a remote terminal

UUCP system to system copy

UUNAME lists system names

UUSTAT reports system status

UUTO file transmission system

UUX inter-system command execution

The tutorials are presented in alphabetical order.

CU: call a remote computer

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **cu** networking utility, which tries to establish a connection between a specified remote computer system. Once the connection has been made, a user can log on to both computer systems at once, with access to command execution and data transfer procedures. Note that access to these facilities may be restricted for security reasons. See the **uux** tutorial in this manual for details.

The destination computer can be specified in one of two ways. The first method is to specify the telephone number of the destination computer's modem. This number is then passed to the automatic dialling modem. The second method involves specifying the system name of the destination computer. The telephone number of this computer is then looked up in the source computer's */usr/lib/uucp/L.sys* file, which contains details of transmission lines, system names, and the appropriate baud rates. Where a destination computer has more than one entry, **cu** tries each in turn until a connection is made.

Once a connection is made, it should be noted that checks should be made as to the integrity of files created on the remote system, or transfered to the remote system from the local system. This is because **cu** does not provide data integrity checks.

SYNTAX

---

cu [-sspeed] [-lline] [options] telno | sysname | dir

DESCRIPTION

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

-sspeed specifies the data transmission speed, measured in baud. Available speeds are 110, 150, 300, 600, 1200, 4800 and 9600. 300 is the default.

-lline specifies the device name of the line to be used. This option can be used to over-ride **cu**'s choice of the first available suitable line. If used without the **-s** option, above, **cu** reads the appropriate transmission speed from the file */usr/lib/uucp/L-devices*. If both **-s** and **-l** are used, **cu** checks *L-devices* to ensure that the specified line and speed are compatible. If they are, **cu** makes the connection: if not, an error message is printed. If the specified device is an auto-dialer, the *telno* argument must be supplied.

options the following options are available:

-h sets **cu** to emulate the behaviour of terminals operating in half-duplex mode.

-t sets **cu** to handle auto-answer ASCII terminals.

-d causes diagnostic messages to be printed.

-e specifies even parity data transmission to the remote system.

-o specifies odd parity data transmission to the remote system.

-m specifies a direct line operating under modem control.

-n specifies that **cu** should request a telephone number from the user, rather than accepting it from the command line.

To specify the destination system, one of the following arguments is used:

telno specifies the telephone number to be used by an auto-dialer system. Where long distance dialling requires delays within the number, the hyphen is used to specify the delay positions. The equals sign represents secondary dial tone.

sysname specifies the system name of the remote computer or terminal. The appropriate direct line or telephone number is obtained automatically by **cu** from the file */usr/lib/uucp/L.sys*, which also provides the appropriate transmission speed. If more than one entry is available for *sysname*, each line or number in turn is tried until a connection is made.

dir specifies the direct line given by the **-l** option, above. If this option is used, **cu** runs as two processes. The *transmit* process reads lines from the standard input, and, except for lines beginning with a tilde (~), passes them *to* the remote system. The *receive* process accepts lines *from* the remote system, and, except for those beginning with a tilde, passes them to the standard output. The meaning of the tilde is described below.

Once the connection is made, the remote system displays a **login:** prompt. Once the user has logged on, the *transmit* process begins. Note that the user has access to both the local and remote systems. The *transmit* process interprets the following special character strings:

~. log out.

~! escape to a shell on the local system.

~!*cmd...* executes *cmd* on the local shell (using the **-c** option of the Bourne Shell; see the *sh*(1) entry in the *Utilities Reference Manual* for details).

~**$***cmd...* executes *cmd* on the local system, but sends the output to the remote system.

~**%cd** changes current directory on the local system.

~**%take** *remfile* [*locfile*]
copies a file called *remfile* from the remote system to the local system. The *locfile* argument optionally specifies the destination of the file on the local system. The filename specified in *remfile* may be changed during the transfer.

~**%put** *locfile* [*remfile*]
copies a file called *locfile* from the local system to the remote system. The *locfile* argument optionally specifies the destination of the file on the local system. The filename specified in *remfile* may be changed during the transfer.

~~*...* sends the line ~*...* to the remote system. That is, to send a line that includes a literal incidence of the tilde character, precede the literal tilde with a second, in order to release the special meaning of the literal tilde.

~%**break** transits a **BREAK** signal to the remote system.

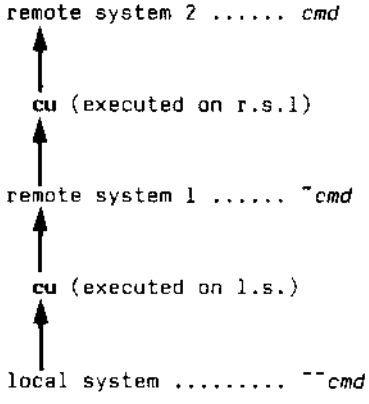
In case of protocol problems during the *transmit* process, see the **cu**(1C) entry in the *Utilities Reference Manual*.

The *receive* process interprets the following:

~>:*file* diverts incoming data recived from the remote system away from the standard output into a file. Once entered, the remote user is free to type as many lines as needed: these will be diverted into the local file. To terminate the operation, ~> is typed.

~>>:*file* acts as the above, except that lines are appended to the end of the specified file. To terminate, ~> is typed.

Where a chain of **cu** commands is used to connect a number of systems, the following use of the ~ character applies:



In the above diagram, **cu** was executed on the local system to connect with the first remote system. It was then executed again, this time on the first remote system, to

connect to a second remote system. This second remote system becomes the top level, and using it, commands can be entered as usual. From the top level, executing a command on the next level down requires that command to be preceeded by a single tilde. A command executed on the bottom level requires two tildes.

EXAMPLES

In the following examples, **cu** is used to establish a link between a local computer system and a remote system. The first example uses the **-s** option to set the speed to 1200 baud, and the *telno* argument to identify the remote system. Because the call is long distance, delays (the - characters) are entered in the number to assist dialling. The > symbol 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.

```
>cu -s1200 00-44-1-409-6621 CR
connected
login:
```

The next example establishes a link with a remote system attached to the local system via a direct line. The line has the device name */dev/tty88*. The device name is specified using the **-l** option, while **dir** forces **cu** to recognise the device name as the identifier of the destination system.

```
>cu -l/dev/tty8d dir CR
connected
login:
```

In the third example, a system name only is used (XYZ43). This forces **cu** to look up the appropriate telephone number or direct line device name. The baud rate is also looked up. The example continues with a user called **spike** logging in to the remote system, and making a copy of a remote system file called *planX*. The next command line sends a copy of the local file *planY* to the remote system, and renames it during the transfer. The next command line adds two lines of text to the file just transferred. The ~> signals the end of the text to be appended. The lasy command line consists of the string ~., which logs **spike** off the remote system.

```
>cu XYZ43 CR
connected
login: spike
>~%take planX CR

>~%put planY plan2 CR

>~>>plan2 CR
Update: the meeting time-tabled for Tuesday CR
has been brought forward to 2.00 CR
~> CR

>~. CR
logout
```

UUCP: system to system copy

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **uucp** networking utility, which copies files from one X/OS system to another, once a connection has been established.

Files and directories can be transferred from the source computer to any location on the destination computer, so long as the correct access permissions are set. Because **uucp** is not interactive, the transfer process occurs in the background, once the command line has been entered. Other commands can be executed while **uucp** is running.

Running **uucp** causes a number of steps to be carried out. First, a command file is created, that contains the instructions for the transfer. These instructions are taken from the options and arguments passed to **uucp** on the command line. A copy of the file to be transferred may then be made. These files are created in a dedicated directory called the *spool* directory, identified as */usr/spool/uucp*. Once activated, **uucp** calls the **uucico** daemon to queue the file, and to carry out the transfer as soon as the system load permits. (Daemons are processes that run in the background.) All these steps are carried out automatically.

The pre-requisites for using **uucp** are therefore knowledge of the name of the remote computer system, and, where appropriate, the login name of the recipient user. The full pathnames of the source file and the destination directory must also be known. Note that when specifying pathnames and filenames, the full range of shell filename expansion notations is available. Note that the **uuname** utility lists the system names of the rmote systems known to the local system. This utility is also described in this manual.

The pathname of the source file may consist of a full absolute path from *root*, or a relative pathname from the current working directory. If the file is held in the current directory, the filename alone is needed.

The pathname of the destination directory consists of the system name of the destination computer system, followed by the pathname of the directory, expressed from the *root* location of the destination system. These two elements are separated by a shriek mark (!).

```
system_name!pathname
```

It may be the case that no direct route exists between the local system and the desired destination. In this case, it may be necessary to include a number of intermediate remote systems in the route. These intermediate remote systems are called *nodes*. The notation used to specify a multiple-node route is as follows:

```
system_name!system_name!...!pathname
```

If *pathname* takes the form of *~/user*, **uucp** will copy the tranferred file to the public **uucp** reception area. The tilde (~) used in this context is interpreted as shorthand for */usr/spool/uucppublic*. In this way, a file can be sent to a user called **spike** on a system called *big_brother* by specifying the pathname as

```
big_brother!~/spike/file
```

A copy of *file* will be sent to */usr/spool/uucppublic/spike*. The public reception area is often referred to as *PUBDIR*.

Note that **uucp** can also send a copy of a file from a remote system to the local system. The command line for doing this is the same, except that the pathname of the source file has the remote system name appended to it, using the !.

In the event of the destination pathname being incorrectly specified, the transfer fails.

SYNTAX

```
uucp [options] s_file d_file
```

DESCRIPTION

The following options and arguments are available to **uucp**:

options these are as follows:

-d where the destination pathname includes intermediate directories that do not currently exist, this option tells **uucp** to create them. This is the default behaviour.

-f where the destination pathname includes intermediate directories that do not currently exist, this option tells **uucp** not to create them.

-c when this argument is specified, **uucp** uses the original of the file to be copied, rather than making a copy in the spool directory. This is the default behaviour.

-C when this argument is specified, **uucp** makes a copy of the original file, placing it in the spool directory. This then used in the transfer.

-m*name* requests **uucp** to send a status message about the transfer to the file identified by *name*. If **-m** is used, but no *name* is given, **uucp** reports by **mail** when the copy is complete.

-n*user* notifies the recipient on the remote system, identified by *user*, when the transfer is complete.

-e*sys* causes the **uucp** command to be executed on the remote system identified by *sys*.

-r causes **uucp** to queue the job, but not to start the transfer. The default behaviour is to initiate transfer as soon as possible after **uucp** is activated.

-j controls the printing of the job number associated with each transfer. This option is explained in more detail in the *uucp*(1C) entry of the *Utilities Reference manual*.

s_file specifies the file or files to be copied.

d_file specifies the destination of the file or files to be copied.

EXAMPLES

In the following example, **uucp** is used to transfer a file called *chptl* from the current directory of the local system to a remote system called *big_brother*. The pathname of the destination directory is */usr/bird/projectl*. Use of the **-n** option ensures that user **bird** on the remote system will be notified of the arrival of the file. The **-m** option will alert the current user that the transfer was successful.

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

```
>uucp -m -nbird chptl big_brother!~bird/projectl CR
```

Note that the ~ metanotation was used to specify the *home* directory of user **bird**, instead of specifying the full pathname. A detailed explanation of the shells' filename expansion metacharcters is given in the *Shell / C Shell X/OS Command Language User Guide*.

UUNAME: lists system names

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **uuname** networking utility, which lists the **uucp** system names of computer systems attached to the network. The known computer systems are listed in the file */usr/lib/uucp/ADMIN*, and descriptions have the format

*sysname***TAB***description***TAB**

SYNTAX

---

uuname [-l] [-v]

DESCRIPTION

Used without options, **uuname** lists the names of all systems connected to the local system by network. This behaviour may be modified using the options available to **uuname**, which are as follows:

-l prints the system name of the local computer.

-v prints additional information about each of the remote computers (*nodes*) attached the network.

EXAMPLES

The following example uses **uuname** to print the names of the remote systems attached to the local computer system currently in use by the user. The second command line uses the **-l** option to print out the name of the current computer system.

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

```
>uuname CR
big_brother
big_sister
asgard

>uuname -l CR
dog_star

>
```

UUSTAT: status enquiry utility

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **uustat** networking utility, which displays the status of previously entered **uucp** commands, and returns information on network connections.

Where no options are given on the command line, **uustat** prints the status of all **uucp** requests issued by the user.

SYNTAX

```
uustat [options]
```

DESCRIPTION

The options available to a normal user of **uustat** are as follows:

-j*job* requests the status of the **uucp** request identified by the job number *job*.

-k*job* kills the **uucp** request identified by the job number *job*. Note that normal users can only kill their own jobs.

-r*job* updates the modification of the job identified by the job number *job*. This is useful where some of the **uucp** administration facilities are in use. See the *uustat*(1C) entry in the *Utilities Reference Manual* for further details.

-u*user* reports on the status of all **uucp** requests issued by the user identified by the name *user*.

-s*sys* reports on the status of all **uucp** requests communicating with the remote computer system identified by *sys*.

-o*hour* reports on the status of all **uucp** requests older than *hour*.

-y*sys* reports on the status of all **uucp** requests younger than *hour*.

-m*mch* reports on the availability of the machine identified by the name *mch*. The word **all** is understood by **uustat** to mean all the machines known to the **uucp** system. The format of the reply is as follows:

system_name time status

where *system_name* identifes the machine, *time* identifies the time of the last status request, and *status* gives the current accessability status of the machine.

-M*mch* produces the same information as the **-m** option, with the addition of the time that the status of the specified machine was last requested, and the time of the last communication with that machine.

-O reports statuses using the octal coded form of output. The codes take the form of a 6 digit number. The meanings of these codes is listed in the *uustat*(1C) entry of the *Utilities Reference manual*. An example is 004000, which indicates that the specified job has been successfully queued.

-q gives the number of **uucp** requests queued for each machine, and the times of the youngest and oldest entries.

Note that special users, such as those logged on as **uucp** and the system administrator have access to other options. These are described in the *uustat*(1C) entry in the *Utilities Reference Manual*.

As the above option descriptions indicate, there are two formats for **uustat** *status* reporting. The octal coded form is only used when the **-O** option is specified. The default format is a verbose description.

Status reporting takes the following form:

```
status
job_number user remote_sys command_time status_time
```

where *status* is octal coded or verbose, and *job_number* identifies the request. The *user* field identifies the user who initiated the request, *command_time* indicates when the request was made, and *status_time* indicates when the last status query was made.

Note that options can be combined on the **uustat** command line, but where this is done, only one of **-j**, **-m**, **-M**, **-k**, **-c** or **-r** can be used.

EXAMPLES

This section illustrates **uustat** in use. The first example shows the command required to obtain a report on all **uucp** requests issued to system **big_brother** by user **spike** in the last 36 hours. The current time is noon on 21st July, and the jobs requested were file transfers. Remember that the **>** in bold type represents the X/OS system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>uustat -uspike -sbig_brother -y36 CR
big_brotherJ55f12  07/20-13:06  spike  12651  memo2
big_brotherJ64f12  07/21-09:43  spike   4429  memo6

>
```

In the second example, the current user is called **spike**. The example begins with **uustat** used without options to obtain a list of all the requests issued by user **spike**. It then goes on to issue the **-k** option to delete job number **asgardA59c6**.

```
>uustat CR
big_brotherJ64f12  07/21-09:43  spike   4429  memo6
asgardA54c6  07/21-11:38  spike   87603  chapter3
asgardA59c6  07/21-11:53  spike   35397  chapter4

>uustat -kasgardA59c6 CR
Job: asgardA59c6 successfully killed

>
```

UUTO: file transmission utility

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **uuto** networking utility, which tranmits a file from the local computer system to a remote system. This utility uses the **uucp** facilities, the difference between **uucp** and **uuto** being that **uuto** always sends the file to a specific area of the remote systems's public reception area. This is */usr/spool/uucppublic/receive/user/system*, where *user* is the login name of the recipient and *system* is the system name of the sending machine. Sub-directories and files appended to this system retain their names. The **uucp** public reception area is also known by the shorthand name *PUBDIR*, and is described in more detail in the **uucp** tutorial in this manual. The recipient of the transfer is notified by **mail**.

SYNTAX

```
uuto [options] source dest
```

DESCRIPTION

Used without options, **uuto** sends the files identified by *source* to the remote system identified by *dest*. The *source* argument takes the form of the pathname that specifies the file or files on the local system. Note that whole directories and sub-directories can be sent. The *dest* argument has two elements, as follows:

system!user

The *system* element is the system name of the remote computer, while *user* is the login name of the intended recipient. Note that *system* has to be a recognised **uucp** name. See the **uuname** tutorial in this manual for details.

The command line options to **uuto** are as follows:

-p tells **uuto** to make a copy of the source file in the spool directory before executing the transfer. The spool directory is located in */usr/spool/uucp*.

-m tells **uuto** to notify the sender by **mail** when the transfer is complete.

Note that if intermediate computer systems (called nodes) are involved, that is, where there is no direct route to the destination system, care should be taken that all nodes in the route are known to **uucp**. This can be checked using **uuname**. The **uucp** tutorial in this manual describes how a multi-node route is specified.

EXAMPLES

In the example, a user called **spike** uses **uuto** to send a file called */usr2/spike/project4/chapt1* to a remote system called **big_brother**. The local system as used by **spike** is called **agentX**. The tilde (~) is shell notation for the home directory of the user. Filename expansion metanotation is described in full in the *Shell / C Shell X/OS Command Language User Guide*. The recipient user on **big_brother** is called **sue**. The **-m** option is used to force notification by **mail** when the transfer is complete.

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

```
>uuto -m ~/project4/chap1 big_brother!sue CR
```

This command line will send *chap1* from the source computer to that area of *big_brother*'s public reception area that is set aside for **uuto** transmissions. The full pathname of the file when it arrives in the *PUBDIR* area of **big_brother** will be */usr/spool/uucppublic/spike/agentX/project4/chap1*.

UUX: inter-system command execution

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **uux** networking utility, which is used to execute a command on one computer system and transmit the output to a file on another system.

Note that for reasons of security, many remote systems restrict the range of commands executable by **uux**, in some cases permitting use of the **mail** system only. If a command is attempted using **uux** that is disallowed on the specified remote system, the user is notified by **mail**. A list of permitted **uux**-run commands is maintained in the file */usr/lib/uucp/L.cmds*. The format of this file is as follows:

cmd,sys1,sys2,...

The first field identifies the command, and subsequent fields identify the machines on which this command may be executed via **uux**. Where no *sys* fields are present, the command may be run on all machines. If the required command is not listed, that command may not be run on any of the systems.

A further restriction often applied by remote systems limits the sending of command output to the public reception area, called *PUBDIR*. This is located in */usr/spool/uucppublic*, and is described in more detail in the **uucp** tutorial in this manual.

SYNTAX

```
uux [options] command_string
```

DESCRIPTION

The *command_string* argument specifies the command to be executed via **uux**, and takes the following format:

```
"![sysname] command [args]"
```

The *command_string* consists of a shell command, using its normal combination of arguments and options, preceded by the name of the remote system involved. Where special characters recognised by the shell are used, for example redirection operators or filename expansion metacharacters, they, or the whole command line should be enclosed in double quotes. Where an argument takes the form of a filename, the appropriate system name is prepended to the pathname. Where the local system in involved, the ! must still appear, but no system name is given. In this way, a **uux** command string can consist of a command executed on the local system, using files taken from one or more remote systems. Otherwise, a command can be executed on a remote system, using files located on a variety of systems. Illustrations are given in the next section.

Filenames are preceded by the appropriate system name, and may be one of the following:

- a full pathname
- a pathname preceded by *~user*, where the tilde points to the home directory of *user*, which is a user's

login name.

The options available to **uux** are:

- when this option is used, the standard input to **uux** acts as the standard input to the *command_string*.

-n the user is not notified when the command has executed.

-m*file* the status of the command execution is reported in *file*. If **-m** is used on its own, notification is sent by **mail** when execution has been completed.

-j controls the job number returned by **uux** whenever a request is queued. For further details of the job number facility, see the *uux*(1C) entry in the *Utilities Reference Manual*.

Note that when files from more than one system are to be used, **uux** attempts to make copies of all necessary files on the system executing the command. When specifying input files, filenames are given as described above. Output filenames must be enclosed inside **\(** and **\)**. This tells **uux** not to search for files that do not exist on any of the systems mentioned.

EXAMPLES

The following examples illustrate the use of **uux**. In the first example, the command line passed to **uux** uses the local system to run the **diff** command, comparing files held on two separate remote systems. The first remote file is stored on a system called **sysX**, and is identified by the pathname */usr/sue/chap1*. The second remote file is stored on system **sysZ**, and is identified by */usr/alan/chap1*.

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

```
>uux "!diff sysX!~sue/chap1 sysZ!~alan/chap1 > !diff.out" CR
```

Note that because the host system of all command string elements must be identified, the **diff** command name was also preceded by the ! identifier, although its host system is the local machine. Because of this, the ! was prepended to the command name, with no system name. In this context, ! is short-hand for the local system.

The two input files are stored on different remote systems, and so their system names were prepended to the pathnames. Remember that ~ is a shell notation that points to the home directory of the specified user. In this case, *~sue* is short-hand for */usr/sue*. The output was piped to a file called *diff.out*, in the current working directory on the local system.

In the second example, the **uux** command line is a **uucp** file transfer, executed on remote system **sysX**, using */usr/sue/memo1* on remote system **sysY**, and transferring a copy of it to the directory */usr/chris* on **sysZ**. The copy is renamed to *output* in the process. Because this file does not already exist on **sysZ**, its name should be enclosed in parentheses on the command line.

```
>uux sysX!uucp sysY!~sue/memo1 \(sysZ!~chris/output\) CR
```

THE LINE PRINTER UTILITES

INTRODUCTION

This seventh chapter contains three tutorials covering the X/OS line printer control utilities. The utilities are as follows:

CANCEL cancels a line printer request

LP sends a request to a line printer

LPSTAT reports line printer status

The tutorials are presented in alphabetical order.

CANCEL: cancel a request to a line printer

INTRODUCTION

This is a tutorial-style introduction to the X/OS **cancel** utility, which is used to delete a job from the print queue. One job at a time can be cancelled, and cancellation may be performed before or after the job begins printing. Any user can cancel a print job, irrespective of ownership. This is so that problems with unattended printers can be remedied. The owner of a cancelled job is then notified by **mail**, and the next job in the queue begins printing.

SYNTAX

```
cancel [ids] [printers]
```

DESCRIPTION

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

ids this is a list of requets IDs, as returned by **lp**. The specified jobs are removed from the print queue, even if printing has begun. The list takes the same forms as are available to **lpstat**, that is

"item1, item2, item*n*"

item1,item2,item*n*

printers this is a list of printer names. The list format is as above. Specifying a printer removes the job currently printing from the print queue.

EXAMPLES

The first example of **cancel** in use begins with the **lpstat** command, used to check on the current state of the print queue. Job 236 is then removed from the print queue. Remember that **>** in bold type represents the X/OS system prompt, and that **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>lpstat CR
fpr_A_235     spike     17209  July 21  14:04
fpr_A_236     spike      4099  July 21  14:05
fpr_A_237     spike     32996  July 21  14:07

>cancel fpr_A_236 CR
request "fpr_A_236" cancelled

>
```

LP: send a request to a line printer

INTRODUCTION

This is a short tutorial-style introduction to the X/OS **lp** utility, which arranges for a named file to be sent to the line printer according to the arguments given on the command line.

SYNTAX

```
lp [-c] [-ddest] [-m] [-nnumber] [-ooption] [-s]
[-ttitle] [-w] files
```

DESCRIPTION

The named files and associated information specified on the command line is collectively called a *request*. If no filenames are given, **lp** uses the standard input. If the standard input is to be printed along with a number of files, it is specified using the hyphen (-). Files are printed in the order that they are specified on the command line.

Each request, irrespective of the number of files to be printed, is assigned a unique *id*. This identifier is used by the **cancel** and **lpstat** commands, also described in this manual.

Lp has the following options:

-c The files are copied before printing. This means that any changes made to the files between queuing and actual printing, will not

be printed. The default arrangement is that files are linked, but not copied, so that changes do appear. Note also that files can be queued using this option, then deleted, but will still be printed.

-d*dest* Specifies the printer or class of printers to be used. If *dest* is a specific printer, then that printer is used. If *dest* is a class of printers, the first available printer is used. The default value of *dest* can be set using the environment variable **LPDEST**.

-m Sends mail to the user (see the *mail*(1) entry in the *Utilities Reference Manual* for details), signalling that printing is complete.

-n*number* Specifies the number of copies of the file to be printed.

-o*option* Specifies the printer- or class-dependent options. The full list of these is available in the *lpadmin*(1M) entry in the *System Administration Utilities Reference Manual*.

-s Suppresses messages from **lp**.

-t*title* Prints the title of the first file in the list on the first page of the output.

-w Writes a message to the user's terminal, signalling that printing is complete. If the user is not logged in, mail is sent instead.

files Identifies the files to be printed. Note that the full range of filename metacharacters may be used, and a filename may include the pathname.

EXAMPLES

In the following example, two files named *chapt1.txt* and *chapt2.doc* are sent to a printer identified by the name *130f*. 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.

```
>lp -c -d130f -n2 -tManual1 chapt*.* CR
request id is 236 (2 files)

>
```

In this example, a number of arguments were specified in addition to the filenames. The files were copied before printing so that further editing could be done while the user waited for the job to finish. Two copies of each file were requested, and the output included a title page reading *Manual1*. Note that the two files were specified using the wildcard characters. For details of how to use these, see *csh*(1) in the *Utilities Reference Manual*.

LPSTAT: returns printer status

INTRODUCTION

This is a short tutorial-style intoduction to the X/OS **lpstat** utility, which is used to obtain the following items of information:

- the print jobs currently queued
- the currently printing job
- the currently active and inactive printers
- the system's default printer
- the status of the job scheduler. Note that this is described in the *lpsched*(1M) entry of the *System Administration Utilities Reference Manual*.

SYNTAX

```
lpstat [options]
```

DESCRIPTION

When used without options, **lpstat** displays details of all requests made to **lp** by the user. In addition, the following options are available:

-a[*list*] prints the acceptance status of the printers identified by the printer and class names held in *list*. The acceptance status indicates whether jobs are being accepted by the specified printers, or not. The absence of

list causes all printers to be reported.

-c[*list*] prints the class names and thr printers included in each class. Individual classes may be specified in *list*, otherwise all classes are reported. For this option, *list* is a list of class names.

-d prints the identity of the system's default printer.

-o[*list*] prints the status of the output requests. The absence of *list* causes all output requests to be printed. For this option, *list* takes the form of printer names, class names, and request IDs.

-p[*list*] prints the status of the printers. The absence of *list* causes all printers to be reported. For this option, *list* takes the form of a list of printer names.

-r prints the status of the scheduler. See the *lpsched*(1M) entry in the *System Administration Utilities Reference Manual* for details.

-s prints a summary of the print scheduler and its status, the default printer destination, the class names and class members, and the printers and associated devices.

-t prints all status information.

-u[*list*] prints a list of the output requests for specified user. For this option, *list* is a list of user login names.

-v[*list*] prints a list of printer names, and the pathnames of the associated special files. See sections 1M and 7 in the *System*

Administration Utilities Reference Manual for details of the I/O commands, application programs, and special files. For this option, *list* is a list of printer names.

request ID any argument to **lpstat** that is not one of the above, is assumed to be a request ID, as returned by **lp**. The status of any specified request is returned.

Note that the *list* arguments, irrespective of content, can take the following forms:

```
"item1, item2, itemn"

item1,item2,itemn
```

EXAMPLES

The following examples illustrate the use of **lpstat**. In the sample screens, 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.

The first command consists of **lpstat** without arguments. Its output comprises the ID number for the print request, the user's login name, the size of the file, and the date and time that the request was made.

```
>lpstat CR
fpr_A_235     spike     17209  July 21  14:04
fpr_A_236     spike      4099  July 21  14:05
fpr_A_237     spike     32996  July 21  14:07

>
```

The next example requests information about a list of printers. This is done using the **-p** option. Note that the second list format is used, and that there are no spaces between the list items.

```
>lpstat -pfpr_A,fpr_B1 CR
printer fpr_A is active. enabled since July 21 08:51
printer fpr_B1 is idle. enabled since July 21 09:12

>
```

The next example uses the **-a** option to determine whether certain printers are currently accepting requests. Note that the first list format is used, and that spaces are permitted between the list items, so long as there are enclosing double quotes.

```
>lpstat -a"fpr_D1, fpr_B2" CR
printer fpr_D1 disabled since July 20 14:31
fpr_B2 accepting requests since July 21 14:02

>
```

The last example uses the **-t** option to print all the status information available.

```
>lpstat -t CR
scheduler is running
system default destination: fpr_D1
device for fpr_D1: /dev/tty10
device for fpr_A: /dev/tty14
device for fpr_B1: /dev/tty19
device for fpr_B2: /dev/tty20
printer fpr_D1 disabled since July 20 14:31
      bad print wheel
fpr_B1 idle since July 21 13:26
```

```
fpr_B2 accepting requests since July 21 14:02
fpr_A_235     spike     17209  July 21  14:04
fpr_A_236     spike      4099  July 21  14:05
fpr_A_237     spike     32996  July 21  14:07

>
```

INTRODUCTION

This chapter contains tutorial-style coverage of the X/OS terminal filters. The filters covered are as follows;

300 handles special functions for the DASI 300 terminal

300s handles special functions for the DASI 300s terminal

4014 handles pagination for the TEKTRONIX 4014 terminal

450 handles special functions for the DASI 450 terminal

GREEK re-interprets the **nroff** character set for various terminals

The tutorials are presented in alphabetical order.

GENERAL

All these filters operate by passing the input text through a number of processes to ensure that it is compatible with the relevant terminal. For example, some of the special control character sequences used by **nroff** may already have a special meaning to a terminal. Also, different terminals may respond differently to an **nroff** character sequence. A tab set command on one terminal may act as the line feed command on another. It is the job of these filters to clear up such incompatibilities. This is done by equating the intention of the input command sequence with the appropriate terminal routine.

Another function of these filters is the production of special characters where they are not, by default, supported by the terminal. For example, the bullet character (●) may be produced by overtyping the characters **o** and **+** to produce ●. By way of these filters, special character handling produces consistent output across the range of terminals supported.

300, 300s: terminal filters

INTRODUCTION

This is a short tutorial-style introduction to the **300** and **300s** terminal filter utilities, which handle special functions for the DASI 300 and 300s terminals respectively. consistent output across the range of terminals supported.

SYNTAX

---

300 [+12][-n]

300s [+12][-n]

DESCRIPTION

The **300** utility re-interprets special characters for the DASI 300 (GSI 300 and DTC 300) terminal; **300s** performs the same functions for the DASI 300s (GSI 300s and DTC 300s) terminal. Half-line forward, half-line reverse, and full-line reverse motions are converted to the correct vertical motions. They also attempt to produce the Greek alphabet and other special characters, and permit easy use of text in 12-pitch. Printing times can be reduced substantially.

Note that if your terminal has a PLOT switch, it must be set to ON before these utilities are used.

The command line options can be used in any order. They have the following effects:

+12 This option sets text to 12-pitch, 6 lines per inch. To use this option, the PITCH switch should be set to 12.

-n This option handles half-line spacing, allowing flexible control of subscripts and superscripts. By default, a half-line is set to four vertical increments.

EXAMPLES

The **300** and **300s** filters can be used as options on the **nroff** command line, or as separate commands into which **nroff** output is piped. The following examples show how to format a file called *testfile* for a DASI 300 terminal. 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.

```
>nroff -T300 testfile CR

>nroff testfile | 300 CR

>
```

For details of the **nroff** options see the *nroff*(1) entry in the *Utilities Reference Manual*. For details on use of the pipe facility, see the *Shell / C Shell X/OS Command Language User Guide*.

The following examples show how *testfile* can be formatted on a DASI 300s terminal using 12-pitch, 6 lines per inch text:

```
>nroff -T300s-12 testfile CR

>nroff testfile | 300s +12 CR

>
```

The following examples show how to format *testfile* on a DASI 300 using quarter-lines rather than half-lines. Note that a half-line is, by default, equal to four vertical increments.

```
>nroff -T300 -2 testfile CR

>nroff testfile | 300 -2 CR

>
```

4014: terminal filter

INTRODUCTION

This is a short tutorial-style introduction to the **4014** terminal filter utility, which handles text pagination for the TEKTRONIX 4014 terminal. A general introduction to the terminal filters may be found at the beginning of the **300** tutorial, in this manual.

SYNTAX

```
4014 [-t][-cn][-pl][file]
```

DESCRIPTION

The **4014** filter establishes the TEKTRONIX 4014 screen format according to the values given to the command line options:

-t if this option is set, output does not wait for the new line signal between pages. This can be useful when formatting into another file.

-c*n* this option sets the screen width, where *n* is the number of columns.

-p*L* this option sets the page length to *L* is the number of lines. The argument *L* is a number representing lines (with the letter *l*), or inches (*i*). If neither is specified, **4014** assumes lines.

file gives the name of the file to be re-interpreted for screen display.

EXAMPLES

The **4014** filter is used as an option on the **nroff** command line, or as a separate command into which **nroff** output is piped.

The following examples show how to format a file called *testfile* for a TEKTRONIX 4014 terminal:

```
>nroff -T4014 testfile CR

>nroff testfile | 4014 CR

>
```

The next examples illustrate how to format *testfile* without waiting for new-line signals between the pages:

```
>nroff -T4014 -t testfile CR

>nroff testfile | 4014 -t CR

>
```

The last pair of examples shows how to set page length to 35 lines, with a two-column format:

```
>nroff -T4014 -c2 -p35 testfile CR

>nroff testfile | 4014 -c2 -p35 CR

>
```

450: terminal filter

INTRODUCTION

This is a short tutorial-style introduction to the **450** terminal filter utility, which handles special functions for the DASI 450 terminal. A general introduction to the terminal filters may be found at the beginning of the **300** tutorial, in this manual.

SYNTAX

```
450 [+12]
```

DESCRIPTION

The **450** filter re-interprets **nroff** output for use on the DASI 450 terminal. It also attempts to produce the Greek alphabet and other special characters.

Note that the terminal's PLOT switch must be set to ON before this utility is used. The SPACING switch should be set to the desired position: either 10-pitch or 12-pitch.

The options function as follows:

+12 This option sets text to 12-pitch. To use this option, the SPACING switch must be set to 12.

EXAMPLES

The **450** filter can be used as an option in the **nroff** command line, or as a separate command into which **nroff** output is piped.

The first set of examples shows how to format a file called *testfile* for the DASI 450 terminal:

```
>nroff -T450 testfile CR

>nroff testfile | 450 CR

>
```

The next two examples show how to format *testfile* on a DASI 450 terminal, using 12-pitch text:

```
>nroff -T450-12 testfile CR

>nroff testfile | 450 +12 CR

>
```

GREEK: terminal filter

INTRODUCTION

This is a short tutorial-style introduction to the **greek** terminal filter utility, which handles special characters for a range of terminals. A general introduction to the terminal filters may be found at the beginning of the **300** tutorial, in this manual.

SYNTAX

```
greek [-Tcode]
```

DESCRIPTION

The *code* argument is the identifier of the terminal. The allowable codes are:

300 DASI 300

300-12 DASI 300 (12-pitch)

300s DASI 300s

300s-12 ASI 300s (12-pitch)

450 DASI 450

450-12 DASI 450 (12-pitch)

4014 TEKTRONIX 4014

The **greek** filter re-interprets the **nroff** character set for a variety of terminals. Special characters are

simulated, where possible, by over-striking. If the terminal code is not specified, **greek** attempts to use the **$term** variable set in your environment. The **$term** variable and its use is described in the *term*(5) entry in the *System Interfaces and Libraries Reference Manual.*

EXAMPLES

The following example illustrate how to format a file called *testfile* on a DASI 300s terminal, using 12-pitch text:

```
>nroff testfile | greek -T300s-12 CR

>
```

THE SYSTEM HANDLING UTILITIES

INTRODUCTION

This last chapter covers a range of utilities for managing terminals, checking on who is using the system, and archiving files. The utilities covered are as follows:

LOGNAME prints login names

TABS sets tab stops for terminals or printers

TAR archives files onto magnetic tape

TTY prints terminal names

WHO reports on who is using the system

The tutorials are presented in alphabetical order.

LOGNAME: print login name

INTRODUCTION

This chapter is a brief tutorial-style description of the X/OS **logname** utility, which prints the contents of the environment variable **$LOGNAME**. This variable is set when the user logs into the system.

SYNTAX

```
logname
```

DESCRIPTION

This command has no arguments or options.

EXAMPLES

The first example illustrates the effect of the **root** user using this command. The **#** symbol in bold type represents **root**'s X/OS system prompt and **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
#logname CR
root

#
```

The second example shows the output obtained by a normal user logged into the system as **spike**. The > symbol in

bold represents a normal user's X/OS system prompt.

```
>logname CR
spike

>
```

TABS: set tab stops on a terminal or printer

INTRODUCTION

This section is a brief tutorial explanation of the **tabs** command, which is used to set tab stops on a terminal or printer. Note that in order to set tabs on a printer, the printer device must be enabled. It is also possible that problems may arise due to not all terminals being compatible with the escape sequences used by the **tabs** command to clear existing tabs, and set new ones.

If entered without arguments or options, **tabs** uses the default tab interval setting of 8 spaces. This is because many programs for high speed output use this setting. Particular tab settings can be specified using the options, and these settings can be saved in a file.

SYNTAX

```
tabs [specs] [+mn] [-Ttype]
```

DESCRIPTION

The arguments ands options are as follows:

specs specifies the placement of tab positions. A wide range of individual specifications are recognised by the **tabs** command, for example:

-n allows the user to set tabs stops every *n* spaces.

--file specifies the file containing the tab settings.

-f sets tab stops required by programmers writing in FORTRAN.

-a sets tab stops suitable for certain Assembler programs.

Note that the full range of values for the *specs* argument is available in the *tabs*(1) entry of the *Utilities Reference Manual*.

+m*n* specifies the position of the left hand margin, where *n* is the number of spaces that the margin is to be indented. If this option is not set, the left margin is given the default indentation, which is 10 spaces.

-T*type* specifies the terminal type. This terminal type must be set in the **$term** environment variable. For details of this variable, see the *term*(5) entry in the *System Interfaces and Libraries Reference Manual*.

EXAMPLES

The following example uses the **tabs** command to set up a terminal for use by a FORTRAN programmer. Remember that the **>** character in bold type represents the system prompt, and that the **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>tabs -f CR

>
```

The next example sets tab stops every 10 spaces:

```
>tabs -10 CR

>
```

TAR: tape file archiver

INTRODUCTION

This is a tutorial introduction to the **tar** tape archive utility. This system down-loads and restores files onto cartridge tape according to options and arguments entered on the command line.

SYNTAX

```
tar [key] [device] [file ...]
```

DESCRIPTION

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

key specifies the actions to be taken by **tar**. It must consist of a single *function identifier* followed by at least one *function modifier*, but more are permitted if necessary. The function identifiers are as follows:

- r the specified file or files are added to the end of the tape. Existing files are not over-written.

- x the specified files are extracted from the tape and restored to the system. If a directory is specified, the entire directory is restored, including sub-directories. If no files are specified, the whole tape is restored. If several copies of the same file exist on the tape, restoring the whole tape in this way will cause earlier copies to be

over-written by later copies.

t the specified files are listed as they are restored. If no files are specified on the command line, all the files and directories on the tape are listed.

u the specified files are added to the tape only if they do not already exist there, or if they have been modified since the last time they were archived.

c creates a new tape by down-loading files onto the start instead of the end of the tape. Existing files are over-written.

The function modifiers are as follows:

0 - 7 specifies the drive on which the tape is mounted. The default drive is 1.

v prints the name of each file as it is processed, preceded by its function letter, where **a** stands for *archive*, and **x** stands for *extract*. This modifier is often used with the **t** function in order to obtain more information about the down-loading operation.

w prints the action to be taken, followed by the filename, then waits for confirmation from the user. Any string beginning with **y** specified acceptance of the operation to be taken. Any other input indicates rejection.

f specifies the name of the device driver to use, instead of the default, */dev/mt?*. This option can be used to select the cartridge tape driver

/dev/rSA/ctape1. If the name of the driver is specified as -, **tar** reads from the standard input or writes to the standard output, as appropriate. In this way, **tar** can be used in a pipeline to move directory systems, see below.

b causes **tar** to use the next element on the command line as the blocking factor for tape records. The permitted range is 1, the default, to 20. When reading tapes, the blocking factor is determined automatically.

l prints error messages if the links between files being down-loaded cannot be resolved.

m does not restore the modification times as they stood at the time of archiving. The new modification time will be the time of extraction from the tape.

o causes extracted files to take on the user and group identifier of the user running *tar*, rather than those of the archived files. This modifier is valid only with the **x** option.

file specifies the file to be down-loaded or restored. Specifying a directory causes all files and sub-directories to be processed.

EXAMPLES

The first example illustrates how **tar** can be used to down-load a series of files selected from the current directory, where the filename extension is *.txt*. The tape driver */dev/rSA/ctape1* is used throughout these examples. Note that the name of each file is printed as it is processed, and that the function letter **a** precedes the filename, indicating that the file is being archived. A new tape is created by down-loading all files from the beginning of the tape. Remember that the symbol **>** in bold type represents the X/OS system prompt, and that **CR** indicates that the carriage return key is to be pressed in order to enter the command line.

```
>tar cvf /dev/rSA/ctape1 *.txt CR
a chapt1.txt 8 blocks
a chapt2.txt 7 blocks
a chapt3.txt 7 blocks

>
```

The next example illustrates how to add the file called *contents.cts* to the end of the tape.

```
>tar rvf /dev/rSA/ctape1 contents.cts CR
a contents.cts 2 blocks

>
```

The third example checks the contents of the tape, using the **t** function.

```
>tar tvf /dev/rSA/ctape1 CR
Tar: blocksize = 20
rw------- 0/1  3527 July 21 14:09 1987 chapt1.txt
```

```
rw------- 0/1  3310 July 21 15:43 1987 chapt2.txt
rw------- 0/1  3266 July 22 09:32 1987 chapt3.txt
rw------- 0/1   886 July 23 12:38 1987 contents.cts

>
```

The next example shows how to restore archived files from a tape. They are to be restored into a new directory, called */usr2/spike/oldvers*. The first two commands create this directory and make it the current location. The last command prints a listing of the contents of the current directory after the files have been restored.

```
>mkdir /usr2/spike/oldvers CR

>cd /usr2/spike/oldvers CR

>tar xvf /dev/rSA/ctape1 CR
Tar: blocksize = 20
x chapt1.txt 8 blocks
x chapt2.txt 7 blocks
x chapt3.txt 7 blocks
x contents.cts 2 blocks

>ls -l CR
rw------- 0/1  3527 July 21 14:09 1987 chapt1.txt
rw------- 0/1  3310 July 21 15:43 1987 chapt2.txt
rw------- 0/1  3266 July 22 09:32 1987 chapt3.txt
rw------- 0/1   886 July 23 12:38 1987 contents.cts

>
```

The last example illustrates how **tar** can be used to move a hierarchy of files and sub-directories. The first element makes */usr2/spike/oldvers* the current directory. The first **tar** command downloads the four files held there (specified by the . which stands for *current directory*, by writing to the standard output. This is then piped

into another sequence of commands which begins by making the current directory */usr2/spike/new* using the .. notation, and extracting the files by reading from the standard input. The files are restored into the new current directory. Note that the standard ouput and input were specified using the **f** function modifier with the - argument. Note also that the default tape driver, *dev/mt1* was used in this example, and that both the **tar** commands use the **v** option.

```
>cd /usr2/spike/oldvers; tar cvf - . | (cd ../new; tar xvf -) CR
a chapt1.txt 8 blocks
a chapt2.txt 7 blocks
x chapt1.txt 8 blocks
a chapt3.txt 7 blocks
x chapt2.txt 7 blocks
x chapt3.txt 7 blocks
a contents.cts 2 blocks
x contents.cnf 2 blocks

>
```

TTY: print the terminal name

INTRODUCTION

This chapter is a brief tutorial introduction to the **tty** command, which prints the pathname of the special file representing the user's terminal.

SYNTAX

```
tty [-l] [-s]
```

DESCRIPTION

The options to the **tty** command are as follows:

-l prints the identification number of the synchronous line attached to the user's terminal (if any)

-s suppresses printing of the path name of the terminal's special file, allowing the user to test the exit code of the **tty** command. Exit codes for **tty** are as follows:

0 standard input is a terminal

1 standard input is from a source other than the terminal

2 invalid option(s) have been selected

EXAMPLES

The first example shown here uses **tty** without options. In this form, **tty** prints the pathname of the terminal's special file. Remember that the > character in bold type 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.

```
>tty CR
/dev/tty21

>
```

The next example uses the **-s** option to test the exit code. The **?** symbol shown here is the special shell variable that contains the exit code. The contents of **?** are printed using the **echo** command. Details of **echo** can be found in the **echo** tutorial in the *User Guide*.

```
>tty -s CR

>echo ? CR
0

>
```

In this case, the exit code was found to be 0.

WHO: who is using the system

INTRODUCTION

This section is a short tutorial-style introduction to the **who** utility, which lists the login name and other information for all users currently using the system. This information is taken from the system file */etc/utmp*, and takes the following layout for all forms of **who** except for the -s option, see below.

name [*state*] *line time activity pid* [*comment*] [*exit*]

where *name* is a user login name, *state* specifies whether the terminal currently used by that user can be written to by other users, and *line* is the name of that terminal's line as found in the directory */dev*. The *time* field specifies when the user logged on, and *activity* specifies how long has elapsed since the terminal was last active. If this field returns a dot, it means that the terminal has been active in the last minute, and is therefore *current*. If the word **old** appears, it means that at least 24 hours has elapsed since the last activity. The process ID of the user's shell is returned by *pid*. Comments relating to the terminal line may be stored in the file */etc/inittab*: if comments are present, they are returned by the *comment* field. The contents of the *exit* field are explained in the *who*(1) entry of the *Utilities Reference Manual*.

Also available is the system file */etc/wtmp*. This contains a history of all terminal processes created whenever a user logs on to the system.

The login name and other information is returned only for the current user when the **who am i** form of the command is used.

SYNTAX

```
who [options] [file]

who am i
```

DESCRIPTION

The *options* to **who** are as follows:

-u prints all the output fields except *state* and *exit*.

-T same as **-u**, except that the *state* field is also output. Where the + character appears, the terminal is writable by anyone: - means that writing to the terminal is restricted to the current user. A bad line is specified by ?.

-l lists all lines not logged on. The same fields are printed as for **-u**, but *name* always takes the value **LOGIN**.

-H same as **-u**, except that column headers are printed.

-q prints out only the names and number of users currently logged in. This option will suppress any other options entered on the command line.

-b prints the date and time of the last system reboot.

-s prints only the *name*, *line* and *time* fields. This is the default.

Note that there are other options available to **who**. These are described in the *who*(1) entry of the *Utilities Reference Manual*.

EXAMPLES

This section illustrates **who** in use. The first example uses the **-u** option to print a full list of the users currently using a sample X/OS system. The **>** symbol in bold type represents the X/OS system prompt, while **CR** indicates that the carriage return key should be pressed in order to enter the command line.

```
>who -u CR
spike    tty18      Sep 29 14:36  .      36
sue      tty21      Sep 28 12:09  old    22

>
```

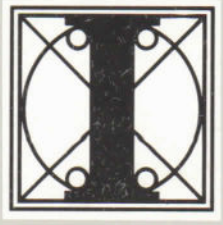
Two users are logged on. The first terminal in the list has been used within the last minute, and is current, hence the **.** notation. The second terminal has not been active for more than 24 hours, and is therefore listed as **old**. The last field gives the process IDs of the users' shells.

The second example uses the **who am i** form of the command.

```
>who am i CR
spike    tty18      Sep 29 14:36

>
```

Code 4043620 D (0)
Printed in Italy



olivetti