UNIXhelp for Users

a reference and guide

The contents of this manual can be found on the WWW at URL:
http://www.ucs.ed.ac.uk/Unixhelp/TOP_.html

First edition
December 1993

Preface

This document and its accompanying quick reference card are just one part of *UNIXhelp for Users*. A hypertext help, based on this document, and computer based training materials are also available. My thanks to all those colleagues who have contributed to the development of these materials.

This work has been funded through the Information Technology Training Initiative (ITTI), a UK wide initiative to provide training materials about, and through, Information Technology.

For further information please contact:

Brian Shields
ITTI National Coordinator
University of St. Andrews
St. Andrews
Fife
Scotland
United Kingdom
+44 334 63014
dbs@uk.ac.st-andrews

John Smith
Edinburgh University Computing Services

Table of Contents

About this guide

Concepts

Getting started

Controlling access to files and directories

Using UNIX commands

Managing your working environment

Text editors

Table of Contents

Using electronic mail

Managing jobs and processes

The shell

File transfer utilities

Text file manipulation utilities

Programming utilities

Shell programming

1

About this guide

1 *About this guide*

This is a guide to a wide range of tasks using the UNIX™ operating system. Try consulting it whenever you are faced with a "how do I ..." situation.

While this guide assumes that you have had an initial introduction to using UNIX, it also appreciates that using a command line interface places severe demands on your memory and recall, making even simple tasks appear difficult and the more complex almost impossible.

You will find that the comprehensive table of contents and the index are good access aids. If the information you are looking for is in this guide then you should be able find it using one or the other of them.

Throughout this guide we use descriptive headings, clear text, and well documented examples, to help you to match your idea of what it is you are trying to do with the information you require. We hope that you find it useful.

Always remember, that if you find using UNIX trying and difficult, you are in good company!

What is in this guide?

Section 2 *Concepts*: Illustrates and explains some of the basic concepts that are represented by any operating system and by UNIX in particular. Reading this section will help you to understand why UNIX behaves in the way it does and give you a clearer idea of what it is you are doing.

Section 3 *Getting started*: Here you can find out how to start and stop using a UNIX system and how to use some of its many commands. It also explains how you can login and run commands on remote hosts.

Section 4 *Using UNIX Commands*: Explains how to make advanced use of the command line by, connecting commands together, redirecting command input and output and using regular expressions. This section also explains how to go about finding the entry for a command in the online manual.

Section 5 *Controlling access to your files and directories*: Every file and directory in your account can be protected from or made accessible to other users by changing the access permissions associated with it. This section shows you how to interpret this information and then explains how you can go about changing permissions for your own needs.

Section 6 *Working with files and directories*: Much of the time that you spend using UNIX involves you in working with files (and directories) in one way or another. This section gives you detailed instructions for the most frequent tasks you will find yourself using.

Section 7 *Managing your environment*: Explains how the environment provided by the shell program depends on the values of shell and environment variables. By changing the value of these variables you can redefine your user environment.

Section 8 *Text editors*: While using UNIX you will often want to create a text file and then change its content in some way. This section introduces you to the basic editing facilities of vi and EMACS, the two editors most often used in the UNIX environment.

Section 9 *Using electronic mail*: Every UNIX system allows you to exchange electronic mail with other users on your local system and on remote systems. Although there are many different mail programs available for UNIX, this section concentrates on explaining the use of mail (or mailx), the basic mail program available on any UNIX system.

Section 10 *Managing jobs and processes*: Every command which you enter starts a program, which, while running, is called a process. This section explains how you can monitor and control those processes which are under your control.

Section 11 *The shell*: Using the resources of the UNIX operating system is actually achieved through the user environment provided by a program know as the *shell*. As well as explaining how you can switch from using one shell to another, this section also introduces you to some of the most useful features provided by the standard UNIX shells.

Section 12 *File transfer utilities*: Many people find the task of transferring files from one computer system to another something of a black art. This section explains the use of several commands that can be used for file transfer. It also includes a section on a particular suite of commands which are still widely used for this purpose within the UK academic community.

Section 13 *File manipulation utilities*: UNIX provides an abundance of commands for manipulating the content of text files. This section introduces you to commands for comparing, sorting and searching file contents.

Section 14 *Programming utilities*: Describes the use of several utilities that are available to assist anyone wanting to develop programs under UNIX.

Section 15 *Shell programming*: As well as using the shell to run commands you can use its built-in programming language to write your own commands or programs. This section describes the shell scripting language features that are available with the Bourne shell.

Conventions used in this guide

Throughout this guide we use several conventions regarding items you type or particular keys you need to press, or information you see displayed on your computer screen.

`plain typewriter font`

Examples of output from the computer, which would normally appear on your terminal screen or line printer listing, are shown in plain typewriter font. The names of programs or commands that appear in paragraph text are also shown in this font.

`bold typewriter font`

Examples of input which you type in to the computer are shown in bold typewriter font to distinguish them from output from the computer. For example:

`filename:` **`cardiac.sys`**

This font is also used for the first occurrence of a program or command name in paragraph text.

`italic_typewriter_font`

At some points you will find examples containing words in an italic typewriter font. In such cases you do not type in the words shown, but information of that type. For example if the following is shown in the document:

`GET FILE=`*`filename`*

you might actually type

`GET FILE='cardiac.sys'`

An ordinary italic font is used to *emphasise* words and phrases to draw them to your attention.

`<RETURN>` means that you press the carriage return key usually marked Return or Enter.

`^` means that you hold down the control key usually marked `Ctrl` or `ctrl` while you are typing another key.

For example

```
^X
```

means hold down the control key, press and release X, then release the control key.

`M-` means that you press the key marked `Esc` or `esc` and then press another key. For example

```
M-Z
```

means press the `Esc` key and then press the `Z` key. Notice that you do *not* hold the `Esc` key down.

▼
This symbol is used to draws your attention to a tip or hint which may be of use or interest.

✖
This symbol draws your attention to a warning containing important information.

2

Concepts

2 Concepts

This section illustrates and explains some of the basic concepts that are represented by any operating system and by UNIX in particular. You do not need to read it to be able to use UNIX, but doing so will help you to understand why UNIX behaves in the way it does and give you a clearer idea of what it is you are doing. It explains:

- what an operating system is
- the structure if the UNIX operating system
- about commands and processes
- the UNIX file system
- the different shells available for UNIX

Operating system

An operating system is the program that controls all the other parts of a computer system, both the hardware - the bits and pieces that the computer is made from-, and the software - the programs that organise the hardware. Most important, it allows you to make use of the facilities provided by the system. Every computer has an operating system.

The UNIX operating system has three important features; a kernel, the shell and a file system.

The kernel

As its name implies, the kernel is at the core of each UNIX system and is loaded in whenever the system is started up; known as “booting” the system.

It manages the entire resources of the system, presenting them to you and every other user as a coherent system. You do not need to know anything about the kernel in order to use a UNIX system. Amongst the functions performed by the kernel are:

- managing the machine's memory and allocating it to each process.
- scheduling the work done by the CPU so that the work of each user is carried out as efficiently as is possible.
- organising the transfer of data from one part of the machine to another.
- accepting instructions from the shell and carrying them out.
- enforcing the access permissions that are in force on the file system.

Location of the kernel file

The root directory contains both the boot program and the file containing the kernel for the system. The name of this kernel file varies from one manufacturer's machine to another, but will usually include the letters `nix` so you can search for it with wildcard characters. For example:

```
ls /*nix*
rwxr----- 1 root 1558734 Sep 24 1992 /vmunix
```

This lists details of any files in the root directory with the characters `nix` in their name. In this example it lists `vmunix` -a common name for the kernel file.

The shell

Whenever you login to a UNIX system you are placed in a program called the shell. You can see its prompt at the bottom left of your screen. To get your work done, you enter commands at this prompt.

The shell acts as a command interpreter; it takes each command that you type and passes it to the operating system kernel to be acted upon, and it then displays the result from this operation on your screen. So actually, when you are “using UNIX” you are in fact interacting with a shell program which is providing you with an interface to the operating system. By changing to a different shell program you can alter the way in which you interact with the UNIX operating system. For more information see "*Changing your shell" on page 11-4.*

UNIX file system

A file system is a logical method for organising and storing large amounts of information in a way which makes it easy manage. The file is the smallest unit in which information is stored. The UNIX file system has several important features.

Different types of file

To you, the user it appears as though there is only one type of file - the file which is used to hold your information. In fact, the UNIX file system contains several types of file.

Ordinary files

This type of file is used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.

Files which you create belong to you, you are said to “own” them and you can set access permissions to control which other users can have access to them. Any file is always contained within a directory.

Directories

A directory is a file that holds other files and other directories. You can create directories in your home directory to hold files and other sub-directories.

Having your own directory structure gives you a familiar place to work from and allows you to structure your information in a way that makes best sense to you.

Directories which you create belong to you. You are said to “own” them, and can set access permissions to control which other users can have access to the information they contain.

Files do not actually reside inside directories. A directory is a file that contains references to other files. The directory holds two pieces of information about each file:

- its filename.
- an inode number which acts as a pointer to where the operating system can find the information it needs about this file.

Filenames are only used by the operating system to locate a file and its corresponding inode number. This correspondence is called a link.

To the operating system, the file *is* the inode number. Multiple filenames can be used to refer to the same file by creating a link between an inode and each of the filenames. For more information see "*Linking files" on page 6-21.*

Special files

This type of file is used to represent a real physical device such as a printer, tape drive or terminal. You need not be concerned about this type of file unless you harbour ambitions of becoming a system administrator or systems programmer!

It may seem unusual to think of a physical device as a file, but it allows you to send the output of a command to a device in the same way that you send it to a file. For example:

```
cat meow.au  /dev/audio
```

This sends the contents of the sound file `meow.au` to the file `/dev/audio` which represents the audio device attached to the system. Can you guess what sound this makes?

The directory `/dev` contains the special files which are used to represent devices on a UNIX system.

There is a special file know as `/dev/null` to which you can redirect unwanted output. This is a null (non-existent) device represented by the file `null` in the directory `/dev`.

Pipes

UNIX allows you to link commands together using a *pipe*. For more information see "*Connecting commands together" on page 5-14.*The pipe acts a temporary file which only exists while it is required to hold data from one command until it is read by another.

Structure of the file system

The UNIX file system is organised as a hierarchy of directories starting from a single directory called the *root* which is represented by a `/` (forward slash). Imagine it as being similar to the root system of a plant or as an inverted tree structure.

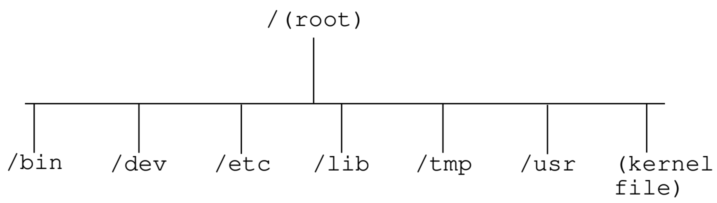
Immediately below the root directory are several system directories that contain information required by the operating system. The file holding the UNIX kernel is also here.

The system directories

The standard system directories are shown below. Each one contains specific types of file.

/bin

This directory contains the commands and utilities that you use day to day. These are executable binary files -hence the directory name 'bin'.



/dev

This directory contains special files used to represent real physical devices such as printers and terminals. One of these files represents a null (no-existent) device.

/etc

This directory contains various commands and files which are used for system administration. One of these files `motd` contains a ’message of the day’ which is displayed whenever you login to the system.

/lib

This directory contains libraries that are used by various programs and languages.

/tmp

This directory acts as a “scratch” area in which any user can store files on a temporary basis.

/usr

This directory contains other directories, amongst these are games, the home directories for each user and the source text for the on-line manual pages.

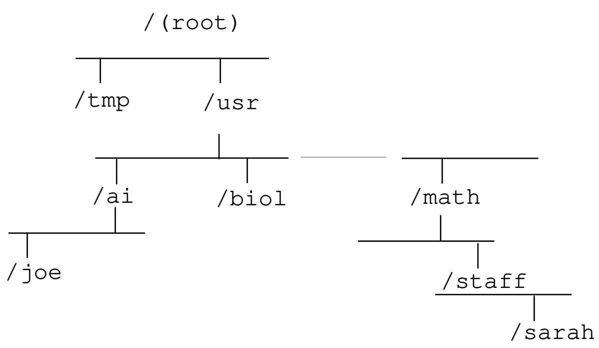
Home directory

Any UNIX system can have many users on it at any one time. As a user you are given a home directory in which you are placed whenever you log on to the system.

User's home directories are usually grouped together under the `/usr` system directory. A large UNIX system may have several hundred users, with their home directories grouped in sub-directories according to some schema such as their organisation or department.

Example of location of a home directory

Although both joe and sarah are registered as users on their local UNIX system, they belong to different departments; joe is a postgraduate student in artificial intelligence while sarah is Head of the Math Department. Their respective home directories might be:



The full pathname to joe's home directory is `/usr/ai/joe`; that to sarah's is `/usr/math/staff/sarah`.

Current directory

When you log on to the system you are always placed in your home directory this is also your current directory. If you then change to another directory this becomes your current directory. The command pwd displays the full pathname to your current directory.

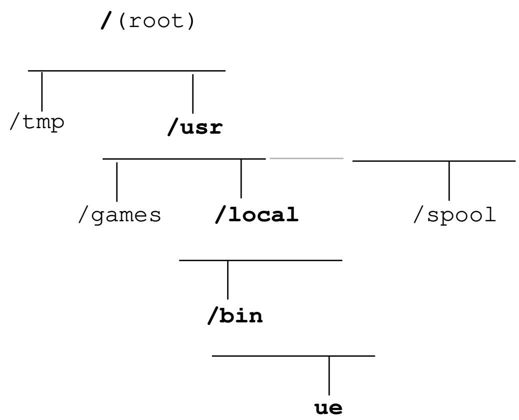
Pathnames: full and relative

Every file and directory in the file system can be identified by a complete list of the names of the directories that are on the route from the root directory to that file or directory. Each directory name on the route is separated by a `/` (forward slash). For example:

```
/usr/local/bin/ue
```

This gives the *full pathname* starting at the root directory and going down through the directories `usr`, `local` and `bin` to the file `ue` -the program for the MicroEMACS editor.

You can picture the full pathname as looking like this:



You can also define a file or directory by its location in *relation* to your current directory. The pathname is given as `/` (slash) separated list of the directories on the route to the file (or directory) from your current directory.

A `..` (dot dot) is used to represent the directory immediately above the current directory. In all shells except the Bourne shell, the ~ (tilde) character can be used as shorthand for the full pathname to your home directory.

Examples of using relative pathnames

1. To give a relative path to a file below your current directory:

```
pwd
# prints the current directory
/usr/eucs/jane
# full pathname to the current directory
ls -l presentation/docs/oview.html
# lists information on the file
```

This defines the file `oview.html` in the directory `docs` which is in the directory `presentation` which itself is a directory in the current directory.

2. To define a file that is in another directory:

```
pwd
# prints the users current directory)
/usr/eucs/jane/presentation/docs
# the full pathname
cd ../illustrations
# use relative pathname
pwd
# check the current directory
/usr/eucs/jane/presentation/illustrations
```

The user `jane` uses the relative pathname `../illustrations` to move up to the directory `presentation` and then down into the directory `illustrations`.

3. To give a pathname using the ~ character:

```
cp ~fran/tmp/for_mike.txt
```

This copies the file `for_mike.txt` from the directory `~fran/tmp`. The ~ (tilde) character is used to give the full pathname to fran's home directory.

Commands and processes

When you enter a command it invokes a program; while this program is running it is called a process. Although there is only one copy of a program held in the file system, any number of processes can be invoked which run this program. For more information see "*Using UNIX commands" on page 5-1.*

When the operating system is started after a boot, a single process is started. This process is the parent of all subsequent processes. Each process created on the system has a unique number known as its PID (Process ID) associated with it.

When you login to the system a process is started to run your shell program. Any processes that are started from within your shell - such as entering a command - are the children of this process. A process can have many children, but only one parent. For more information see "*Managing jobs and processes" on page 10-1.*

Finding out more about UNIX

These books provide a detailed technical description of the UNIX operating system. They are not intended for the general user.

System V variants

Bach, M.J.
The Design of the UNIX Operating System
Prentice-Hall ISBN

Berkeley Standard Distribution

Karels, Michael.J., Leffler, Samuel. J., McKusick, Marshall. K., and Quarterman, John.S.
The Design and Implementation of the 4.3BSD UNIX Operating System
Addison Wesley ISBN 0-201-06196-1

3

Getting started

3 *Getting started*

Here you can find out how to start using a UNIX system and how to use some of its many commands. If you have not used a UNIX system before look at the information in section 2 of this guide.

Logging on to the system

Before you can begin to use the system you will need to have been given a username and a password.

Obtaining a username and password

The procedure for obtaining a username and password is different for each company and organisation. Usually, it will involve you filling out an application form and then waiting until you are advised of your username and password. However, some organisations now issue these to you automatically as part of your joining process.

Example of the logging on procedure

1. To login to your account:

   `login:` **`helper`**

   `Password:`

   This logs in the person who has been given the username `helper` on this system. UNIX is case sensitive; if your username is **`earl`**, do not enter **`EARL`** or **`Earl`**.

   Your password is **never** displayed on the screen. If it were, other users might see it!

Problems with logging in

Message	Meaning
`login incorrect`	You have made a typing error while typing in your username or password. The <Caps Lock> key is on and everything that you type is being sent to the system in UPERCASE letters: UNIX is case sensitive. If your username is "helper", do not enter **HELPER** or **Helper**.

Forgotten your password?

Everyone does sooner or later. Contact your local helpdesk and explain your problem to them.

Special login files

When you login to your account the system carries out any commands that it finds in special "login" files.

These commands define your working environment; which editor you use, which printer to send your work to and where to deliver printed output to.

Making changes to these login files changes your working environment. For more information see "*Managing your working environment" on page 7-1.*

Your home directory

Once you are logged in to your account you are always placed in your home directory. This is your very own location in the file system: it never varies.

Logging out

To finish using a UNIX system you must go through a process known as "logging out".

1. To logout enter the command **`logout`** or **`exit`**.
2. If this does not work press Ctrl-d.

Problems with logging out

Message	Meaning
`There are stopped jobs`	`You have started a job by issuing a command and have then "stopped" the command from executing and placed it in the background using the key sequence Ctrl-z.`

	`1. Start the job again by typing fg and pressing the <Return> key.`
	`2. Quit the job or wait until it has finished.`
	`3. Try to logout again.`

`Not login shell`	`You are working in another "shell" that has been started from within your "login shell".`
`Not login shell`	`1. Enter the command exit.` `2. Try to logout again.`

About the logout file

When you logout of your account the system carries out any commands held in a "logout" file which is in your home directory.

Making changes to the commands in your logout file changes your working environment. For more information see "*Managing your working environment" on page 7-1.*

Who else is logged on?

To find out if there are other users logged on to the system use the command:

```
who
```

This lists the username of each person that is currently logged on to the system. It also gives details about the terminal they are using and the date and time that they logged on.

1. To find out who is logged on to the system:

```
who
erco02      ttyPt       Nov 3 08:01 (braid.ucs:0.0)
hwb         ttyPu       Oct 26 11:23 (dirac.math-
s.ed.a)
helper      ttyPv       Nov 5 09:07 (artws.ucs.ed.ac.)
djs         ttyPx       Nov 5 11:39 (129.215.72.75)
aipex       ttyPB       Nov 5 11:41 (owl.dai.ed.ac.uk)
billy       ttyPF       Nov 5 09:12 (diamond.epcc.ed.)
```

This displays the name of the user, the date and time that they logged on and the name of their terminal. Notice that a user might have been logged on for several days or even weeks.

Logging in more than once

You may have another account on the same system but under a different username. You can become this other user by either switching to another account or logging in to it from the account you are presently using.

▼
To help you remember which account you are using, set the shell prompt so that it displays your username. For more information see "*Setting the prompt variable: C shell" on page 7-10.*

Switching to another account

To switch to using another account use the command

```
/bin/su username
```

and enter your password when prompted.

You are still in the same working directory you were when you issued the command, but you now have access to the file system for your other account. To stop using this account enter the command **`exit`**.

Example of switching to another account

1. To switch to another account from your current account:

```
pwd
/u/usr/helper
/bin/su projX
Password:
whoami
projX
cd
pwd
/u/usr/projX
```

The user `helper` switches to the account they have under the username `projX` and confirms this with the **`whoami`** command.

They then use the **cd** command to place themselves in the home directory for the account they have switched to. They can now work with the files and directories in this account in the usual way.

Logging in to another account

To login to another account use the command

```
/bin/su - username
```

and enter your password when prompted. You are then logged into this account just like when you do your usual login.

To stop using this account enter the command **logout** or **exit**.

Example of logging in to another account

1. To login to another account that you own from your current account:

```
/bin/su - projX
Password:
You have new mail
Mon Jul 27 15:35:46 WET DST 1992
pwd
/u/user/projX
```

This logs the user in to another account that they own with the username `projX`.

▼

There may be several UNIX systems at your site connected together by a network. If you have accounts on these remote systems you can login to them from the system you are currently using. See "*Logging in to remote systems*" on page 8 of this section.

Changing your password

Your password is important; it stops other users from gaining access to your account. Never give your password to anyone. If you do you may have your access to the use of the system suspended or withdrawn

As an extra security measure change your password at regular intervals, using a password of 6 to 8 letters, with a mixture of upper and lower case characters and at least one non-alphanumeric character such as ! or &.

✗ Caution

Never do any of the following:

- choose a word from a dictionary.
- use personal information such as your name.
- tell anyone your password.
- write your password down anywhere.

Example of changing a password

1. To change your password:

```
passwd
Old password: enter your current password
New password: enter your new password
Retype new password: re-enter your new password
```

Enter the command `passwd` and then respond to the prompts by entering your old password followed by your new one. You are then asked to confirm your new password. If you make a mistake, the message

```
Mismatch - password unchanged.
```

is displayed and your password remains unchanged. Try again.

Logging in to remote systems

There may be several UNIX systems at your site connected together by a network. If you have accounts on these remote systems you can login to them from the system you are currently using.

Logging in using rlogin

To login to your account on a remote system use the command:

```
rlogin remote_system_name
```

To login to a remote system using a different username, use the command:

```
rlogin remote_system_name -l username
```

You are prompted for the password to your account on that system.

Preventing password prompting

You can avoid being prompted for your password whenever you rlogin to your account on a remote system. To do this:

1. Login to the remote system using rlogin.
2. Use an editor to create a file .rhosts.
3. Edit this file so that it contains entries for the other remote systems that you have accounts on.

   Each entry is a line consisting of the name of the remote system followed by a space and your username on that system.
4. Save the file to your home directory.
5. Check that the access permission for this file prevents other users from writing to it.

▼

To save time creating and maintaining different versions of the the .rhosts file on each remote system:

1. Create a .rhosts file with an entry for each system that you have an account on.

2. Copy this file to your home directory on each remote host.

Examples of using the rlogin *command*

1. To login to the remote system iona using the same username:

   ```
   rlogin iona Password:
   ```

   This logs the user into their account on the remote system named `iona`. Their username on this sytem is identical to their username on the system they are currently using.

2. To login to the remote system barra using the the username helper:

   ```
   rlogin barra -l helper barra
   Password:
   ```

   This logs the user in to the remote system named barra on which they have an account for the username helper. This username is different to their username on the system they are currently logged in to.

Logging in using telnet

`telnet` enables you to connect your terminal to another remote system. To login to your account on a remote system use the command:

```
telnet remote_system_name
```

You are prompted for your username and password for your account on the remote system. Once you are logged in, you can begin to issue commands on the remote system.

Examples of using the telnet ***command***

1. To connect to your account on the remote system known as `castor`:

```
telnet castor
Trying 129.215.112.8 ...
Connected to castor.ucs.ed.ac.uk.
Escape character is '^]'.

SunOS UNIX (castor)

login:
```

2. To start a telnet session and open a connection to the host "mstore-1":

```
telnet

telnet> open mstore-1
Trying 129.215.112.1 ...
Connected to mstore-1.ucs.ed.ac.uk.
Escape character is '^]'.

SunOS UNIX (mstore-1)

login:
```

Using telnet ***in session mode***

To login to several remote hosts, one after the other, use a telnet "session". To do this enter the command:

```
telnet
telnet>
```

The telnet prompt is displayed and you can enter commands to manage sessions on other hosts.

telnet ***access to the World Wide Web***

The WorldWideWeb (W3) is a wide area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents. To try out a simple interface to W3:

1. Open a telnet connection to the host

`info.cern.ch`

with the username

`www`

No password is required. Follow the on-screen instructions.

Summary of telnet commands

To ...	Do this ...
Connect to a remote system	telnet> open remote_system_name
Suspend a telnet connection and return to the telnet prompt	Ctrl)
Close a connection to a remote system	Ctrl) telnet> close
**Suspend telnet and work on the local system	telnet> z
**Resume the use of telnet	fg telnet>
End the telnet session	telnet> quit
To display a list of telnet commands	telnet> ?
To get an explanation of a command	telnet> help name_of_command

** These commands only work if you are using the C shell, Korn shell, TC shell or BASH shell.

4

Controlling access to files and directories

4 Controlling access to files and directories

Every file and directory in your account can be protected from, or made accessible to, other users by changing its access permissions.

You can only change the permissions for files and directories that you own.

Displaying access permissions

To display the access permissions of a file or directory use the command:

```
ls-l filename
     directory
```

This displays a one line summary for each file or directory. For example:

```
-rwxr-xr-x 1 erpl08 1995 Feb 22 15:51 my.html
```

This first item `-rwxr-xr-x` represents the access permissions on the file. The following items represent the number of links to it; the username of the person owning it; its size and the time and date it was last changed, and finally, its name.

Examples of displaying access permissions

1. To display the permissions on a single file:

```
ls -l file1
-rw-r--r-- 2 unixjohn 3287 Apr 8 12:10 file1
```

This displays the following information about the file `file1`.

`-rw-r--r--`	- access permissions
`2`	- number of links to this file
`unixjohn`	- owner
`3287`	- size in bytes
`Apr 8 12:10`	- date and time last modified

2. To display the permissions on the contents of a directory:

`ls -l prototype`

This displays the same information as in the previous example above, for each file and directory in the directory `prototype`.

3. To display the permissions on each file and directory in your current directory:

```
ls -l
-rw------- 1 erpl08 4307 Jun 17 10:00 FAQ.xdh
drwx------ 2 erpl08 1024 Jun 17 10:00 SCCS
-rw------- 1 erpl08 15119 Jun 17 10:00 commands.xdh
-rw------- 1 erpl08 266 Jun 17 10:00 concepts.xdh
```

This displays the following information about every file and directory.

`d`	- this is a directory
`rwx------`	- access permissions
`2`	- number of links
`erpl08`	- owner
`1024`	- size in bytes
`Jun 17 10:00`	- date and time last modified

Understanding access permissions

There are three types of permissions:

`r`	- read the file or list files in the directory
`w`	- write to the file or directory
`x`	- execute the file or search the directory

Each of these permissions can be set for any one of three types of user:

u	- the user who owns the file (usually you)
g	- members of the group the owner belongs to
o	- all other users

The access permissions for all three types of user can be given as a string of nine characters:

user	**group**	**others**
r w x	r w x	r w x

These permissions have different meanings for files and directories.

Summary of access permissions

Permission	**File**	**Directory**
r read	read a file	list files in ...
w write	write a file	create a file in rename file in delete file ...
x execute	execute a shell script	read a file in write to file in execute a file in execute a shell script in ...

Examples of access permissions

1. The owner of the file has read and write permission. Nobody else has any permissions.

```
ls -l file1
-rw------- 2 unixjohn 3287 Apr 8 12:10 file1
```

2. The owner has read and write permission. Everyone else -the group and all other users- can read the file.

```
ls -l file2
-rw-r--r-- 2 unixjohn 3287 Apr 8 12:11 file2
```

3. The user has read, write and execute permission. Everyone else -the group and all others- can execute the file.

```
ls -l myprog
-rwx--x--x 2 unixjohn 3287 Apr 8 12:10 myprog
```

4. This is a directory. The user has read, write and execute permission. The group has read and execute permission on the directory. Nobody else can get access to it.

```
ls -l
drwxr-x--- 2 erpl08 1024 Jun 17 10:00 SCCS
```

Default access permissions

When you create a file or directory its access permissions are set to a default value. These are usually:

`rw-------`

gives you read and write permission for your files; no access permissions for the group or others.

`rwx------`

gives you read write and execute permission for your directories; no access permissions for the group or others.

Access permissions for your home directory are usually set to `rwx--x--x` or to `rwxr-xr-x`.

The first set of permissions on your home directory allows members of your group and all other users to:

-read, write to or execute a file in your home directory for which you have given them permission to do so.

Other users cannot list the contents of your home directory.

The second set of permissions on your home directory would allow members of your group and all other users to:

-list the contents of your home directory

-read, write or execute a file in your home directory for which you have given them permission to do so.

These are the absolute maximum permissions that you should set on your home directory.

In both cases, you have full access permissions to your home directory.

Change group ownership of files and directories

Every user is a member of one or more groups. To find out which groups you belong to use the command:

```
groups
```

To find out which groups another user belongs to use the command:

```
groups username
```

Your files and directories are owned by the group (or one of the groups) that you belong to. This is known as their “group ownership”.

To list the group ownership of your files:

```
ls -gl
```

You can change the group ownership of a file or directory with the command:

```
chgrp group_name file/directory name
```

You must be a member of the group which you are giving ownership to.

Changing access permissions

To change the access permissions for a file or directory use the command

```
chmod mode filename
chmod mode directory_name
```

The "mode" consists of three parts: who the permissions apply to, how the permissions are set and the permissions to be set.

Who the permissions apply to

Permissions can be made to apply to any one of three types of user:

u	- the user who owns the file (usually you)
g	- members of the group the owner belongs to
o	- all other users

How the permissions are set

Permissions can be set for any type of user in the following ways:

+	add a permission
-	remove a permission
=	assign a permission and reset all other permissions

The permissions to be set

The permissions to be to be set can be any combination from:

r	read
w	write
x	execute

Permissions have different meanings for files and directories. See "*Summary of access permissions*" on page 3 of this section.

✕Caution

Never set write permission for all other users on a file or directory which is in your home directory. If you do other users will be able to change its content. This can represent a serious security risk.

You can use the **`umask`** command to set default access permissions. See "*Changing default access permissions*" on page 8 of this section. This will ensure that any files and directories you create have consistent permissions.

Changing access permissions with the chmod command

1. To give yourself permission to execute a file that you own:

```
chmod u+x file1
```

This gives you execute permission for the file file1.

2. To give members of your group permission to read a file:

```
chmod g+r file2
```

This gives the group permission to read the file file2.

3. To give read permission to everyone for a particular type of file:

```
chmod a+r *.pub
```

This gives everyone permission to read all files with the extension `.pub`.

4. To give the group write and execute permission:

```
chmod g+wx $HOME/SCCS
```

This gives all members of the group permission to place files in the directory `SCCS` in your home directory. They can also list (`ls`) the contents of this directory.

Setting access permissions numerically

There is a shorthand way of setting permissions by using octal numbers. Read permission is given the value 4, write permission the value 2 and execute permission 1.

```
r w x
4 2 1
```

These values are added together for any one user category:

1	= execute only
2	= write only
3	= write and execute (1+2)
4	= read only
5	= read and execute (4+1)
6	= read and write (4+2)
7	= read and write and execute (4+2+1)

So any access permission can be expressed as three digits. For example:

	user	**group**	**others**
`chmod 640 file1`	rw-	r--	---
`chmod 754 file1`	rwx	r-x	r--
`chmod 664 file1`	rw-	rw-	r--

Changing default access permissions

You will need to understand the previous section on how to set file access permissions numerically before reading this section.

Every time you create a file or a directory, its access permissions are set to a predetermined value. This value is defined by the *file-creation mode mask*; a number, consisting of three octal digits which is subtracted from a similar number used by the system to create files and directories.

To display the value of this mask enter the command:

```
umask
```

This will produce something like:

```
077
```

Sometimes the `0` (zero) is omitted. The values may be different on your system.

Without the mask the system would create files with permissions set to `666` and directories to `777`.

The values in the mask are subtracted from these values to give a default value for access permissions for the files and directories created by you. For example:

```
 777        (system value for directories)
-077        (value of the umask)
 -----
 700        (default access permission of rwx------)
```

To change your default access permissions until you logout or exit from your current shell use the command:

```
umask nnn
```

Each "*n*" is number from 0 to 7.

To make changes to the default access permissions last until you next change them place the `umask` command in your shell login file.For more information see "*Managing your working environment*" on page 7-1.

Examples of using the umask *command*

1. To give yourself full permissions for both files and directories and prevent the group and other users from having access:

```
umask 077
```

This subtracts `077` from the system defaults for files and directories `666` and `777`. Giving a default access permissions for your files of `600` (`rw-------`) and for directories of `700` (`rwx------`).

2. To give all access permissions to the group and allow other users read and execute permission:

```
umask 002
```

This subtracts `002` from the system defaults to give a default access permission for your files of `664` (`rw-rw-r--`) and for your directories of `775` (`rwxrwxr-x`).

3. To give the group and other users all access except write access:

```
umask 022
```

This subtracts `022` from the system defaults to give a default access permission for your files of `644` (`rw-r--r--`) and for your directories of `755` (`rwxr-xr-x`).

4. To give the group read and execute access, but deny other users access:

```
umask 027
```

This subtracts 027 from the system defaults to give a default access permission for your files of 640 (`rw-r-----`) and for your directories of 750 (`rwxr-x---`).

5

Using UNIX™ commands

5 Using UNIX commands

You interact with the UNIX operating system by entering a command at the shell prompt. You can look up the command in the On-line Manual for a complete description of it. See "*Look-up online manual*" on page 18 of this section.

When you enter a command it invokes a program. While this program is running it is called a process. Although there is only one copy of a program, any number of processes can be invoked which run this program.

About commands

The basic form of any UNIX command is:

```
command_name options argument(s)
```

Most descriptions for commands such as those given in the On-line Manual use a much more precise syntax. For example:

```
cp [-iprR] filename ... directory
```

This syntax has a few simple rules. Apply them to the command description and you can understand how the command is to be used.

Command syntax rules

To understand the command syntax apply the following simple rules:

1. Any options or arguments enclosed in [] square brackets are optional.

2. Anything not enclosed in [] square brackets must be entered.

3. **Boldface** words are considered to be literals and must be typed exactly as they appear. This usually applies to the com-

mand name and command options.

4. Arguments shown in `italics` must be replaced by whatever it is that they represent. This is usually the name of a file or directory.
5. Ellipses '...' mean that the previous argument can be repeated any number of times.

Command options

Options modify the way that a command works. They usually consist of a hyphen followed by a single letter.

For example the wc command counts the number of words characters and lines in a file. By using a different option you can choose what is counted.

wc -w file1	counts the words
wc -c file1	counts the characters
wc -l file1	counts the lines

Using multiple options

Some commands accept multiple options. These can be grouped together after a single hyphen:

```
command -abc argument
```

A small number of commands require each option to be given separately. Check the command synopsis for a command to see how a command requires its options to be given.

An option that takes an argument, such as the name of a printer or a filename is given separately.

Examples of using multiple options

1. Options can be grouped together:

```
ls -FI
```

This lists the contents of the current directory. Directories are shown with a trailing '/'; files which have execute permission are shown with a trailing '*'.

The inode number of each file is also displayed.

2. Some options need to be given separately:

```
lpr -Plp2 -#4rm report.ps
```

This prints four copies of the file report.ps, removes the file and sends mail to the user on completion.

The option `-Plp2` which specifies the printer to be use is given separately from the other options. If the command were given as

```
lpr -Plp2#4rm report.ps
```

the command would try to find a printer with the name `lpr2#4rm`.

Command arguments

Most commands are used together with one or more arguments. For some commands these arguments are optional; others require you to use them with an argument.

If the argument contains an empty space then you must quote the argument.

Commands become far more effective when used together with regular expressions

Examples of using command arguments

1. To use a command with no argument:

`date`

Entered on its own this command displays the current time and date.

2. To use a command with a single argument:

`cd newdir`

The `cd` command (used to change from one directory to another) takes the name of the directory `newdir` as its argument.

3. To use a command with several arguments:

`cat file1 file2 file3`

The `cat` command takes the names of three files as arguments. It prints `file1` followed by `file2` and then `file3` on the screen. This explains the name of the command - it concatenates any files given to it as arguments.

4. To 'quote' an argument:

`ls 'part 1'`

This lists information about the file `part 1`. The quotation marks ensure that the shell interprets the command line as the `ls` command followed by a single argument '`part 1`'.

If the quotation marks were left out, the shell would interpret the command line as an instruction to list information about the file `part` and then the file `1`.

Using regular expressions

A regular expression is a concise way of expressing any pattern of characters. You construct a regular expression by combining ordinary characters with one or more metacharacters: characters that have special meaning for the shell.

You can use the following metacharacters within any shell to create regular expressions that match file names.

Metacharacter	Action
`?`	`matches any single character`
`*`	`matches any number of any characters`
`[nnn]`	`matches any of the enclosed characters`
`[!nnn]`	`matches any character that is not enclosed`
`[n-n]`	`matches any character in this range`

Examples of using regular expressions in filenames

1. To list all files with a particular extension:

`ls *.fm`

This will list all the files in the current directory that have the extension `.fm`.

2. To remove a range of files:

`rm foo.?`

This removes all the files in the current directory with a single character extension to their name. Files `foo.c` and `foo.o` would be removed as would file `foo.3`.

3. To copy all the files containing the same pattern:

```
cp [pP]art[0-9] ../book
```

This copies all files whose name contains a pattern that starts with either `p` or `P` followed by the letters `art` and ending in any number from 0 through to 9.

So the file `part1` would be copied to the parent directory book as would the file `Part7`.

Characters with special meaning for the shell

The characters

; & ( ) | ^ ~ % { } $ # ′ " @@ * ? [ ] - !

a newline (<Return>)

a space (<Space>)

a tab(<Tab>)

are all shell metacharacters. They represent something other than the character or symbol that they are. You must quote any command argument or regular expression containing one or more of these, so that it is passed to the shell unchanged.

Guidelines for quoting command arguments

Character	Action
`\`	`prevent the shell from expanding any single special character.`
`‘.....’`	`prevent the shell from expanding all special characters enclosed in this string except !`
`“.....”`	`prevent the shell from expanding any enclosed special character except $, ‘ and !.`

Examples of quoting special characters

1. To remove the special meaning from a single metacharacter:

```
rcp ubik:/usr/home/roger/recipe.ps .;\<Return>
lpr -Pps3 #6m recipe.ps
```

This prevents the shell from expanding the newline character generated by pressing the key. You can then continue entering the command on the next line.

2. To make the shell interpret a metacharacter literally:

```
echo &
[1] 1463
[1] Done echo
echo \&
&
```

The shell interprets the `&` character at the end of a command line to mean that the command is to be run in the background. So the command

```
echo &
```

simply runs the `echo` command in the background and since the command has nothing to echo it immediately exits. When the `&` character is preceded by a `\` (backslash) the `echo` command interprets this literally and simply echoes (displays) this character on the screen.

3. To prevent the shell from expanding an argument containing several special characters:

```
$ sed -e 1,$s/V[0-9].*: //p /etc/motd
sed: command garbled: 1,/V[0-9].*:
$ sed -e '1,$s/V[0-9].*: //p' /etc/motd
DYNIX/ptx(R) Tue Feb 25 12:04:15 GMT 1993
```

First, the shell tries to expand the command line and changes its meaning in the process. When the arguments to the command are protected by enclosing them in single quotes the `sed` command is successful. It displays the operating system version number contained in the message of the day.

4. To prevent the shell from expanding special characters but allow command substitution:

```
echo 'My working directory is `pwd`'
My working directory is`pwd`
echo "My working directory is `pwd`"
My working directory is /home/jane/docs
```

When enclosed within single quotes the characters '`pwd`' are treated literally as part of the string to be echoed. Placed within double quotes, the command within the backquotes is executed and its output substituted in its place.

Where are commands located?

Commands are executable binary files located in directories with the name bin (for binary). Many of the commands that you use are located in the directory `/usr/bin`.

When you enter the name of a command the shell checks to see if it is a built-in command. If it is not, it looks for the binary file that the command name refers to in each of the directories that are defined in the `PATH` environment variable.

A typical value for the `PATH` variable might be:

```
/bin:/usr/bin:/usr/local/bin:$HOME/bin
```

This is a list of colon separated pathnames of the directories that contain the executable binary files for each command.

The shell will first look for the command that has been entered in the directory `/bin`. If it is not found there it will then look in the directory `/usr/bin` and so on until the command is found.

If the command is not found on your path the message

```
command_name: command not found
```

is displayed.

Entering commands

To enter a command type its name followed by any options and arguments that you want the command to have. Leave one or more spaces between the command name, its options and arguments. If necessary you can continue the command onto another line.

Correct mistakes by using the <Delete> or <Backspace> key to backup along the command line and correct it.

Continuing a command onto another line

To continue a command onto another line enter a '\' (backspace) character and then press <Return>. You can then continue entering the command. For example:

```
rcp ubik:/usr/home/roger/recipe.ps .;\
lpr -Pps3 #6m recipe.ps
```

This copies the file recipe.ps from the home directory of the user roger on the remote host `ubik` to the users current directory. It then sends this job to the printer ps3 with the request that 6 copies are printed.

This person ought to use a Xerox!

If you are using the Bourne, BASH or Korn shell a character appears as a prompt at the start of the second command line. For example:

```
ls -l | grep '^d.........x'  dirs;\

> less dirs
```

Users of the C or Tcsh shell do not receive this prompt.

Examples of commands

1. To enter a simple command:

   ```
   ls
   ```

   This command lists the contents of the current directory. In this example, no options or arguments are used.

2. To enter a command together with one of its options:

   ```
   ls -l
   ```

   The -l option changes the way the ls command works. This command produces a listing of the contents of the current directory in a long format.

3. To enter a command together with several options and an argument:

   ```
   lp -dps5 -n3 -m foo.c
   ```

   The option `-dps5` specifies that the file `foo.c` is to be printed on printer `ps5`. The option `-n3` specifies that 3 copies of the file are to be printed. The option `-m` requests that a mail message is sent to the user once the command has been completed.

4. To enter a command with several arguments:

```
lp part1 part2 part3
```

This will print the files `part1`, `part2` and `part3` on the default printer.

5. To enter a command using a regular expression as an argument:

```
rm -i [pP]*.ps
```

This will prompt the user if they want to remove any file whose name matches the regular expression `[pP]*.ps`. This will remove files with the names `part1.ps` and `Personnel.ps`.

Command substitution

To include the output from one command within the command line for another command, enclose the command whose output is to be included within 'backquotes'. For example:

```
echo The date today is `date +%d/%m/%y`
The date today is 22/03/93
```

The output from the command `date +%d/%m/%y` is substituted at the appropriate location within a command line that uses the `echo` command.

Entering more than one command

To enter several commands on the one command line, use a `;` (semicolon) to separate each command from the next. For example:

```
cd ..; ls -l
```

This command line contains two commands. If necessary you can continue the commands onto another line.

Redirecting standard input and output

UNIX considers any device attached to the system to be a file, including your terminal! By default, a command treats your terminal as the standard input file from which to read in information. Your terminal is also treated as the standard output file to which information is sent from the command.

This action can be changed by redirecting standard input and standard output from and to any other file.

Redirecting standard input

To redirect the standard input for a command use the < (less than) character followed by the name of the input file. For example:

```
mail tony < memo
```

This redirects the standard input to the `mail` command so that it comes from the file memo. The effect of this is to mail the contents of this file to the user `tony`.

Redirecting standard output

To redirect the standard output from a command use the > (greater than) symbol followed by the name of the output file. If the file that you redirect standard output to does not already exist it will be created. For example:

```
grep Smith /etc/passwd > popular
```

This redirects the standard output from the grep command so that it goes to the file `popular`. This file will contain all occurrences of the string Smith that were found in the `/etc/passwd` file.

✗ Caution

Redirecting standard output to a file that already exists overwrites its contents with the standard output. You can append output to an existing file. See "*Appending standard output to a file*" on page 13 of this section.

Command output

When a command is carried out successfully, the results are sent to the standard output file. But if the command fails to execute successfully the results are displayed on the standard error file; by default, this is your terminal. For example:

```
ls -l part1.txt
part1.txt not found
```

The **ls** command fails because the file `part1.txt` does not exist in the current directory. A message is sent to the standard error -your terminal- informing you of this.

The display of standard output from a command can be controlled by connecting it to a pager program. See "*Examples of connecting commands with pipelines*" on page 14 of this section.

You can redirect the standard error away from your terminal to another file but this is generally discouraged. The method for doing this also varies slightly with different shells. Consult the manual page for the shell that you are using for further details.

Appending standard output to a file

To append the standard output from a command to a file use two `>>` (greater than) symbols followed by the name of the file. If the file does not exist it is created.

For example:

```
cat part1 part2 part3 part4 >>chpt2
```

This reads files `part1` through `part4` in turn and appends them to the file `chapt2`. The contents of `part2` are appended to `part1` and so on.

Connecting commands together

UNIX allows you to link two or more commands together using a pipe. The pipe takes the standard output from one command and uses it as the standard input to another command.

```
command1 | command2 | command3
```

The | (vertical bar) character is used to represent the pipeline connecting the commands.

With practise you can use pipes to create complex commands by combining several simpler commands together.

Examples of connecting commands with pipelines

1. To pipe the output from one command into another command:

```
who | wc -l
342
```

This command tells you how many users are currently logged in on the system: a lot!

The standard output from the `who` command is piped into the `wc` command as its standard input. Used with the -l option this command counts the numbers of lines in the standard input and displays the result on the standard output.

2. To connect several commands together:

```
ps -aux|grep joe|sort +5 -6|less
```

The first command `ps -aux` outputs information about the processes currently running. This information becomes the input to the command `grep joe` which searches for any lines that contain the username "`joe`". The output from this command is then sorted on the sixth field of each line, with the output being displayed one screen at a time using the `less` pager.

Locating a command

To find out if a command or utility is available on your system enter the command:

```
which command_name
```

If the command is found its pathname is displayed. The message

```
command_name: Command not found
```

or similar, is displayed when the command cannot be found.

Examples of locating commands with the which *command*

1. To find out if a command name is located in your command path:

```
which ue
/usr/local/bin/ue
```

This returns the information that the MicroEMACS editor (`ue`) is available on this system as the file `/usr/local/bin/ue`.

2. To find out which (if any) of several commands are available:

```
which delete talk ps
No delete in /usr/local/utils/bin:/bin:/usr/bin:

/usr/local/ucb/bin:/usr/local/gnu/bin: /usr/local/
public/bin:/usr/bin/X11:. /usr/bin/talk /bin/ps
```

This returns the information that the file for the command **`delete`** does not exist in any of the directories that are specified in this users PATH environment variable.

Pathnames for the files of the commands talk and ps are displayed so these commands are available on this system.

Making other commands available

The value of the PATH variable is set so that you have access to the commands available on your system.

You can use commands that are not on your path list by setting a new value for the PATH environment variable. For more information see "*Changing environment variables" on page 7-7.*

Suspending commands with the sleep command

To suspend the execution of a command for a specific amount of time use the command:

```
sleep time_in_seconds; command
```

The second command will not be carried out until the amount of time specified as an argument to the `sleep` command has elapsed. The semicolon (;) is used to allow the two commands to be entered on the same command line.

This command is very useful if you need to do something else before a command is executed.

Example of using the sleep *command*

1. To set an interval before a command is executed:

```
sleep 20; ls -l &
cd $HOME/Mail
-rw------- 1 erpl08 4761 Jul 17 1992 IUSC_Confence
drwx------ 2 erpl08 512 Jun 5 1992 SGML
drwx------ 2 erpl08 512 Mar 15 1991 drafts
```

This gives the user 20 seconds to change to another directory before the command `ls -l` is executed to list the contents of the current directory `$HOME/Mail`. The first command line is run as a background job to allow the user to continue entering commands.

Running commands at a specific time

To execute a command at a specific time use the **at** command.It is usual to expect your UNIX system to allow you to use this command, although it may be that you are prevented from using it.

Unfortunately, the way that `at` takes commands differs from one UNIX system to another. To get around this, place the comand(s) you want carried out in a shell script, and redirect this as input to the at command. For more information see "*Shell programming" on page 15-1.*For example:

```
at 17:00 < foo
job 1544 at Wed Dec 15 16:48:00 1993
```

This will cause any commands in the file `foo` to be redirected into the at command and executed at 1700 hours on this day. The output from the commands contained in the file `foo` is sent to you by mail.For more information see "*Using electronic mail" on page 9-1.*

Read the manual page for the at command which is on your system if you plan to use the command on a regular basis. See "*Look-up online manual"* on page 18 of this section.

Look-up online manual

The command **`man`** gives you access to an on-line manual containing a complete description of every command available on the system. Here is a typical example:

```
MAN(1) USER COMMANDS MAN(1)



NAME
man - display reference manual pages; find reference
pages by keyword

SYNOPSIS
man [-] [-t] [-M path] [-T macro-package] [[section]
title
...] ...
man [-M path] -k keyword ...
man [-M path] -f filename ...

DESCRIPTION
man displays information from the reference manuals. It
can display complete manual pages that you select by
title, or one-line summaries selected either by keyword
(-k), or by the name of an associated file (-f).

--More--(8%)
```

`man` can also provide you with one line descriptions of commands specified by name; or for all commands whose description contains any of a set of keywords.

Understanding the organisation of the manual

The manual is organised into several sections each of which covers a different area.

Section ID	Description
1	User Commands
2	System Commands
3	Subroutines
4	Devices
5	File Formats
6	Games
7	Miscellaneous
8	System Administration
l	Local Commands
n	New Commands

The command

```
man section_number intro
```

displays a manual page giving an overview of that section of the manual.

Searching by keyword

The command

```
man -k keyword
```

displays a one line synopsis of each command that has "keyword" in its description.

Enclose a "phrase" in single quotes to search for it.

```
man -k ' phrase_to_search_for'
```

Checking the description of a command

To check what a command does:

```
whatis name_of_command
```

This displays the description for the command that is given in its online manual page.

Viewing manual pages

To view the manual page for a command enter:

```
man name_of_command
```

The content of every man page is organised under a number of headings.

Manual page headings

These are the headings used to organise the content of the manual page for a command. A manual page may not contain information under every heading but the NAME, SYNOPSIS and DESCRIPTION are available for every command.

NAME

This lists the exact name of the command(s) covered by this manual page and gives a short description of what the command does.

SYNOPSIS

This section gives a summary of command usage. It enables you to see how options can be given to the command. For example:

```
cp [-iprR] directory1 directory2
```

The first word or consecutive characters are the name of the command.

The characters enclosed in [square brackets] are optional. A '-' (hyphen) preceding a character indicates that this is an option that can be applied to the command.

Options grouped together after a single hyphen, like this, `[-abcD]` can be used in this format. If they are given separately, like this, `[-n] [-r] [-Z]` then they must be used in this format.

Arguments are represented by a name such as *`file`*, *`filename`* or *`directory`*.

DESCRIPTION

This section gives a detailed description of how the command works. For some commands this section consists of a few lines; for others it can be several pages long.

OPTIONS

This section explains the action that each option has when used with the command.

ENVIRONMENT

This section gives the names of any environment variables used by the command and the value(s) that they can be set to.

EXAMPLES

This section provides you with examples of how the command can be used. It is often omitted.

FILES

This section gives details of the files used by the command.

SEE ALSO

This section refers you to the manual page for commands that contain related information. Follow these references to learn which commands are associated with each other.

WARNINGS

This section warns you against using the command in a way that might cause you problems.

DIAGNOSTICS

This section explains the meaning of the various diagnostic messages that the command may present you with. Messages that are considered self explanatory are not included.

BUGS

This section advises you about known 'bugs' or shortcomings with the command.

Printing manual pages

To print a copy of the manual page to a high quality PostScript printer:

```
man -t name_of_command
```

To print a copy of a manual page on a line printer:

```
man name_of_command | lp -dname_of_printer
```

6

Working with files and directories

6 *Working with files and directories*

Much of the time that you spend using UNIX involves you in working with files (and directories) in one way or another. Before using the information in this section look at the section in this guide on the UNIX file system. For more information see "*UNIX file system*" on page 5-3.

You will also find information about other aspects of working with files in other parts of this document; protecting files, finding files, searching file contents and transferring files.

Copying files

▼
Before you copy a file consider whether making a link to it would provide a better solution. See "*Linking files*" on page 21 of this section.

Copying files in the same directory

To create an exact copy of a file use the cp (copy) command.

```
cp [-option] source destination
```

The *source* is the name of the file to be copied; the *destination* is the name of the file in which the copy is to be placed.

X Caution

Take care what name you give to the file copy. If you choose the name of a file that already exists it will be overwritten by the new copy! To avoid this either:

1. Always use the **cp** command together with its `-i` option. This will prompt you for confirmation whenever the copy would overwrite the existing file.

or:

2. Create a command alias for the **cp** command to make it always use the `-i` option. This will make sure that you are always prompted for confirmation whenever the copy would overwrite an exiting file. For more information see "*Creating a command alias*" on page 5-25.

Examples

1. To copy a single file in the current directory:

```
cp notes sect3.txt
```

This makes a copy of the file `notes` in the file `sect3.txt`.

2. To copy a file using the `-i` option:

```
cp -i notes part2.txt
```

This makes a copy of the file `notes` in the file `part2.txt`. If this file exists the prompt

```
part2.txt: File exists
```

is displayed. The file is not copied.

Copying more than one file

You can use special *wildcard* characters whenever you want to copy several files that have similar filenames into another directory. For more information see "*Defining files with wildcard characters*" on page 5-9.

Instead of entering the **cp** command followed by several filenames you can use a single filename that contains one or more wildcards.

Examples of using cp *with wildcard characters*

1. To copy files that match on several characters:

   cp *.txt chapt1

   This copies all the files in the current working directoy with the extension `.txt` to the sub-directory `chapt1`.

2. Top copy files that match on a single character:

   cp sect?b partA

   This copies any file with the name `sect[1-9]b` to the sub-directory `partA`.

3. To copy all the files within a specified range:

   cp sect[1-4]c partB

   This copies files `sect1c`, `2c`, `3c` and `sect4c` to the sub-directory `partB`.

Copying files to another directory

To copy a file to another directory from your current directory give the name of the source file followed by the pathname to the destination file.

cp *source path_to_destination*

For the destination file to have the same name as the source file use:

cp *source path_to_destination_directory*

Examples of using the cp command

1. To copy a file from your current working directory to a subdirectory:

```
cp fig2 part2/figure2
```

This copies the file `fig2` from your current working directory to the file `figure2` in the subdirectory `part2`.

2. To copy a file to the parent directory:

```
cp mail.txt ..
```

This copies the file `mail.txt` to the directory immediately above the current working directory.

The `..` (dot dot) is shorthand for the parent directory.

3. To copy a file from a subdirectory to a directory elsewhere in the file system:

```
cp docs/vi.Z /tmp
```

This copies the file `vi.Z` in the subdirectory `docs` to the directory `/tmp`.

4. To copy a file to your home directory:

```
cp /usr/local/doc/ue.txt $HOME
```

This copies the file `ue.txt` to your home directory. `$HOME` is the environment variable that contains the value for your home directory. For more information see "*Displaying current variables"* on page 5-1.

Problems copying files

Message	Meaning
cp:file pathname: Permission denied	The access permissions of this file donot premit your to copy it. The access permissiond for the destination directory do not allow you to copy files into it.
cp: file_pathname: Is a directory (not copied).	You have tried to copy a directory
cp: file pathname: No such file or directory	The file (or directory) that you have tried to copy does not exist.
Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2	You have forgotten to specify the destination file (or directory).

Copying files from another directory

To copy a file from another directory to your current directory give the pathname to the source file followed by the name of the destination file.

```
cp path_to_source_file path_to_destination
```

For the destination file to have the same name as the source file use:

```
cp path_to_source_file .
```

The . (dot) is shorthand for the current working directory.

Examples of using the cp *command*

1. To copy a file from a subdirectory to the current working directory:

```
cp notes/note3 sect3.txt
```

This copies the file `note3` from the subdirectory `notes` to the file `sect3` in the current working directory. A relative pathname `notes/note3` is used to define the source

2. To copy a file from another directory to the current working directory, preserving the file name:

```
cp /usr/lib/more.help .
```

This creates a copy of the file `more.help` in the current working directory. A full pathname `/usr/lib/more.help` is used to define the source

3. To copy a file from another user's directory using a relative pathname.

```
cp ~helper/tmp/for_john tmp/comments
```

This copies the file `for_john` from the directory `tmp` which is in the home directory of the user `helper`, to a directory with the same name in the user's home directory. The name of this new file is `comments`.

Notice that a relative pathname `~helper/tmp/for_john` is used to define the source file.

Copying files from another user's account

To copy a file belonging to another user you must know the pathname of the file. For more information see "*Pathnames: full and relative"* on page 5-8.You must also have the correct access permissions to allow you to copy the file. For more information see "*Controlling access to files and directories"* on page 5-1.

▼

Before you copy a file consider whether making a link to it would provide a better solution. See "*Linking files"* on page 21 of this section.

Examples of copying files from another user

1. To copy a file from another user's home directory:

```
cp ~helper/file1 file2
```

This copies file1 from the home directory of user `helper` to `file2` in your current working directory.

2. To copy a file from a subdirectory in the user's home directory:

```
cp ~helper/book/part2/file1 file2
```

This copies `file1` from the subdirectory `book/part2/` in the home directory of user `helper` to `file2` in your current working directory.

Copying files between UNIX hosts.

For more information see "*Copying files between hosts"* on page 5-8.

Creating files

This section describes several methods for creating files. Most of the time you will probably use an editor such as **`vi`** or **`EMACS`** to do this. For more information see "*Text editors"* on page 5-1. You will find creating files with the the **`echo`** and **`cat`** commands useful for making quick notes.

▼
You can create an exact copy of a file by making a copy of it. See "*Copying files"* on page 1 of this section.

Create a file with the cat command

1. Type the command

```
cat > name_of_file
```

2. Type in your text.
3. Press the `<Return>` key to start a new line.

When you have finished typing in your text:

4. Enter `Ctrl-d` (Press and hold down the `Ctrl` key and type a "`d`").

This signals the end of the input to the **cat** command and returns you to the system prompt.

✕ Caution

Do not use the name of a file that already exists. If you do, it will be overwritten by the new copy. To avoid this, set the shell variable `noclobber` so that existing files cannot be overwritten (or clobbered).

Example

1. To create a file with the **cat** command:

```
cat > memo
remember to make appointment at opticians
get cost of scheduled flight to Athens
Ctrl-d
```

This creates a file called `memo` in the current directory containing two lines of text.

Create a file with the echo command

1. Type the command **echo** and then continue to type the text that you want to appear in the file.

echo > *name_of_file*

When you have finished typing in your text:

2. Press the `<Return>` key.

This method is useful for creating short files containing one line of text.

Example

1. To create a file with the **echo** command:

echo use an editor for large files > tips

This creates a file called `tips` in the current directory containing a single line of text.

Creating files with a text editor

For more information see "*Text editors*" on page 5-1.

Defining files with wildcard characters

Wildcard characters can be used to represent many other characters. You can use them on the command line whenever you need to create a filename, for use with a command.

Useful wildcards are:

`*` - matches any characters

`?` - match any single character

`[...]`- matches any character in the enclosed list or range.

Examples of using wildcard characters

1. To match any number of characters in a filename:

```
ls *.txt
```

This lists all the files in the current directory that have the extension `.txt`. So `Help.txt` as well as `nonsense.txt` would be listed.

2. To match any single character in a filename:

```
cat memo? > mymemos
```

This concatenates any or all files with the filenames

```
memo0 -> memo9

memoa -> memoz

memoA -> memoZ
```

and places their contents in the file `mymemos`.

3. To remove a range of files:

```
rm part[1-3]
```

This removes files `part1,part2` and `part3` from the current directory. If no files are found that match these names the message

```
No match
```

is displayed.

Deleting (removing) files

To remove a file use the command:

```
rm filename(s)
```

You cannot remove a file in another user's account unless they have set access permissions for the file which allow you to. For more information see "*Controlling access to files and directories*" on page 5-1.

X Caution

Once a file is removed there is no way to retrieve it. To avoid this, always use the **rm** command together with its `-i` option. For more information see "*Creating a command alias*" on page 5-25.

```
rm -i filename
```

This will prompt you for confirmation that you want to remove the file. You can make this the default action for the rm command by creating a command alias that uses the `-i` option.

Examples of removing files: rm *command*

1. To remove a single file:

   `rm help.txt`

   This removes the file `help.txt` from the current directory.

2. To remove several files:

   `rm artwork/figure[2-7]`

   This removes files `figure2` through to `figure7` from the subdirectory `artwork`.

3. To remove files interactively:

   `rm -i *.fm`

   This will prompt you to confirm that you want to remove a file from the current directory whenever that file has the suffix `.fm`.

Answering **`y`** will delete the file. The file is not deleted if any other response is given.

Problems using the `rm` command

Message	Meaning
`rm` *`filename: No such file or directory`*	`The file or directory you are trying to remve does not exist.`
`rm [-rif] file ...`	`The file or directory you are trying to copy does not exist.`
`rm: override protection 600 for` *`filename`*`?` **`y`** `rm: filename not removed: Permission denied`	`You don't have permission to remove this file or directory. Most likely you're trying to erase a system file or another user's file.`

Determining file type

The **`file`** command examines the content of a file and reports what type of file it is.

To use the command enter:

```
file filename
```

Use this command to check the identity of a file, or to find out if executable files contain shell scripts, or are binaries. Shell scripts are text files and can be displayed and edited.

Examples of using the file *command*

1. To check the identity of a single file:

```
file ReportA
ReportA: Frame Maker document
```

This checks the identity of the file `ReportA` and reports that this file has been created using the FrameMaker application.

2. To check the identity of several files:

```
file *
external.c: c program text
external.h: c program text
mc:         sparc demand paged executable
mcXThelp:   executable shell script
mcdemo:     executable shell script
mchelp:     executable shell script
mchome:     executable shell script
mctools:    executable shell script
```

This checks the identity of every file in the current directory. Notice how it reports whether a file is an executable binary or a shell script.

Displaying files

UNIX provides you with several versatile tools for viewing the content of text files. See section nnn for information about how to use a text editor for this purpose.

Displaying files with the cat command

The **cat** command is useful for displaying short files of a few lines. To display longer files use a pager.

To display the contents of a file use the command:

```
cat filename
```

Examples: the cat *command*

1. To display a file:

```
cat README
```

This displays the contents of the file README. If the file contains more lines than can be displayed on the screen at once it will scroll by: use a pager instead. (See below).

2. To display control characters and other nonprinting characters use the -v option:

```
cat -vt myprog.c
```

The -t option causes each tab to be shown as ^I and each line feed as ^L. This option only works when used with the -t option.

Displaying files using a pager

A pager is a program for displaying the contents of a text file one screenful at a time. The most commonly used pager programs are **more** and **less**.

These pagers have many other capabilities built in, such as searching for a particular piece of text. You can also use a pager for controlling the display of the standard output from a command. For more information see "*Examples of connecting commands with pipelines"* on page 5-14.

Changing the PAGER *environment variable*

Other commands such as **man** and **mail** use a pager to display their output on the screen. The default pager used by these commands is defined by the PAGER environment variable. For more information see "*Displaying current variables"* on page 5-1.

Using the more *pager*

To display a file use the command:

```
more filename
```

This will display the first screenful of the file. The prompt

```
--More--(nn%)
```

is given in the bottom left-hand corner of the screen.

You can now use one of the following commands:

`<space bar>`	Display next screenful of text.
`<Return>`	Display next line of text.
`q`	Exit from more: this can be done at any time.
`d`	Scroll forwards about half a screenful of text.
`b`	Skip backwards one screenful of text.
`h`	Display a list of commands (help).

Using the less pager

`less` has better features than **`more`** and can be customised to suit your particular needs. Many of its commands are the same as those of the text editor **`vi`**.

To display a file use the command:

```
less filename
```

This will display the first screenful of the file. A prompt with the filename is given in the bottom left-hand corner of the screen.

You can now move around the file using the **`less`** commands.

Summary of less commands

Commands marked with * may be preceded by a number. Notes in parentheses indicate the behavior if a number is given.

h H	Display a complete command summary
q :q :Q ZZ	Exit.
e ^E j ^N CR	* Forward one line (or *n* lines).
y ^Y k ^K ^P	* Backward one line (or *n* lines).
f ^F ^V SPACE	* Forward one window (or *n* lines).
b ^B ESC-v	* Backward one window (or *n* lines).
z	* Forward one window (set window to *n*).
w	* Backward one window (set window to *n*).
d ^D	* Forward one half-window (set half-window to *n*).
u ^U	* Backward one half-window (set half-window to *n*).
r ^R ^L	Repaint screen.
R	Repaint screen, discarding buffered input.
/pattern	* Search forward for (*n*-th) matching line.
?pattern	* Search backward for (*n*-th) matching line.

NOTE: search commands may be modified by one or more of:

!	search for NON-matching lines.
*	search multiple files.
@	start search at first line of first file in the comand line list..

Customising the less command

The **`less`** command can be given a number of options that change its behaviour. These options are used by setting the value of the `LESS` environment variable. For more information see "*Displaying current variables"* on page 5-1.

Options given at the command line override those set by the environment variable. An option that is given in the `LESS` variable can be reset by starting the command line option with `-+`.

Examples of customising the less *command*

These examples show how the environment variable `LESS` is set; first by users of the C and TC shells and then by users of the Bourne and BASH shells.

1. To display the size of the file in the prompt:

   `setenv LESS -P`

   or:

   `LESS=-P;export LESS`

   This displays the size (in bytes) of the file being displayed.

2. To change the prompt and the way that text is displayed:

   `setenv LESS '-N -S -P%f'`

   or:

   `LESS='-N -S -P%f';export LESS`

   This puts the filename in the **`less`** prompt(`-P%f`), displays the number of each line in the file(`-N`) and truncates lines that are too long to be displayed on the screen so that they do not wrap onto the next line(`-S`)

   Check the manual page for a full list of all the options that are available.

Displaying the first few lines of a file

To display the first 10 lines of a text file use the command:

```
head filename
```

Examples of using the head *command*

1. To display the first 10 lines of a file:

`head /usr/local/doc/vi.txt`

This displays the first 10 lines of the file vi.txt which is in the directory /usr/local/docs/FR.

2. To display the first 5 lines of several files:

`head -5 *.xdh`

This displays, one after the other the first 5 lines of each file with the extension `.xdh` in the current directory.

The first line of each file is shown as:

```
==> filename <==
```

▼

You may find it useful to pipe the output from this command through a pager such as more or less.. This will make sure that you are able to view the output a screenful at a time.

Displaying the last part of a file

To display the last 10 lines of a text file use the command:

```
tail filename
```

Examples of using the tail *command*

1. To display the last 10 lines of a text file;

`tail fred`

This displays the last 10 lines of the file `fred` which is in the current directory.

2. To display the last 50 characters in a file:

`tail –50c help.txt`

This displays the last 50 characters in the file `help.txt` which is in the current directory.

Editing files

For more information see "*Text editors"* on page 5-1.

Finding a file

To locate a file in the file system, use the **`find`** command.

find *pathname* –name *filename* -print

The *`pathname`* defines the directory to start from. Each subdirectory of this directory will be searched. The `-print` option must be used to display results.

You can define the filename using wildcards. Xre here.If these are used, the filename must be placed in 'quotes'.

Examples of using the find *command*

1.To find a single file below the current directory:

`find . –name mtg_jan92 –print`

This displays the pathname to the file `mtg_jan92` starting from the current directory. If the file is not found nothing is displayed.

▼

The SystemV version of this command takes the ~ (tilde) as meaning the user's home directory: BSD versions require the use of `~/`.

2. To find a file below your home directory:

`find ~/ -name README -print`

This displays the pathname to every file with the name `README` in your home directory or its subdirectories.

3. To find several files below the current directory:

`find . -name '*.fm' -print`

This displays the pathname to any file with the extension `.fm` which exists below the current directory.

4. To find a directory:

`find /usr/local -name gnu -type d -print`

This searches to see if there is a subdirectory `gnu` in the directory `/usr/local`.

Combining other commands with the find *command*

You can also use the **`find`** command to find a file and then carry out a command on that file. To do this use the command:

`find` *`pathname`* `-name` *`filename`* `-print -exec` *`command`* `{} \;`

To be prompted for confirmation before the command is executed use the command:

`find` *`pathname`* `-name` *`filename`* `-print -ok` *`command`* `{} \;`

The command must be followed by a `\` (backslash) and a `;` (semi-colon). The `{}` (pair of braces) substitute the pathname of the file that is found as an argument to the command.

Examples of executing commands with the find *command*

1. To remove several files:

`find . -name 'mtg_*' -print -exec rm {} \;`

This will search for and remove all files starting with the expression `mtg_` from the current directory and its subdirectories.

2. To find and remove specific files:

```
find . -name '*.tmp' -ctime +30 -print -exec rm {} \;
```

This removes all files in the current directory and its subdirectories with the filename extension `.tmp` which have not been changed within the last 30 days.

3. To find and remove files interactively;

```
find ~/Docs -name '*.ps' -print -ok rm {} \;
```

This searches for files with the extension `.ps` starting in the subdirectory `Docs` in the user's home directory. Each time a file is found that matches this expression, the user is prompted to confirm that they want to remove it.

Entering **y** removes the file.

Linking files

Files do not actually reside inside directories. A directory is a file that contains references to other files. The directory holds two pieces of information about each file:

- its filename.

- an inode number which acts as a pointer to where the system can find the information it needs about this file.

Filenames are only used by the system to locate a file and its corresponding inode number. This correspondence is called a *link*.

To the system, the file *is* the inode number. Multiple filenames can be used to refer to the same file by creating a link between an inode and each of the filenames.

To link a file use the command:

```
ln filename1 linkname
```

Making a link to a file does not create another copy of the file; it makes a connection between the file and the linkname.

Examples of using the ln *command*

1. To link files in the same directory:

```
ln notes circular
```

This creates a link `circular` to the file `notes` in the current working directory.

2. To use **`ln`** together with special pattern matching characters:

```
ln project/*.xdh /home/joe/book
```

This links all the files with the extension `.xdh` in the sub-directory `project` to the directory `/home/joe/book`.

Using symbolic links

Your files (and directories) may be located on several different file systems. For more information see "*UNIX file system*" on page 5-3. To link files that are in different file systems you need to make a symbolic link.

To make a symbolic link use the command:

```
ln -s source linkname
```

Examples of using the ln *command*

1. To make a symbolic link from a file in one subdirectory to a file in another subdirectory:

```
ln -s reports/reportA publications/my_report
```

This makes a symbolic link between the file `reportA` in the subdirectory `reports` and the filename `my_report` in the subdirectory `publications`.

2. To make a symbolic link to a file in a subdirectory to your current working directory:

```
ln -s docs/editors/vi.ref .
```

This makes a symbolic link between the file `vi.ref` in the subdirectory `docs/editors/` to the filename `vi.ref` in the current working directory.

▼
You can make a symbolic link that points to a file that does not exist. The link becomes active when the file that it points to is created.

Removing the file does not affect the symbolic link itself and you can substitute another file for it to point to. Look at this example:

```
ln -s $HOME/accounts/pending current
# make a symbolic link to a non-existent file
more current
more: accounts/pending: No such file or directory
# this is what happens if you try to page it
cp ~/accounts/may ~/accounts/pending
# now to create the file that the symbolic link points
# to
more current
XYZ Co. Accounts for May 1993
.....
# and the symbolic link works
rm ~/accounts/pending; more current
more: accounts/pending: No such file or directory
# removing the file makes the symbolic link inactive
cp ~/accounts/june ~/accounts/pending
more current
XYZ Co. Accounts for June 1993...
# creating another file that the link points to
# makes it active once again
```

Removing files that have links

Remove a file that has hard links to it the same way as you would remove any other file:

```
rm filename(s)
```

Any files that were created by making hard links to the original filename will continue to exist in their own right,

Removing a file to which symbolic links have been made has a different effect. Although the linkname remains the actual file that this name points to does not. Trying to use a command on the linkname will give you the message:

```
command: filename: No such file or directory
```

Problems linking files

Message	Meaning
`filename: Cross-device link`	`The file you are trying to make a link to is located on a different physical device. The solution is to create a symbolic link.`
`directory_name is a directory`	`You are trying to make a link to a directory. Check the pathname for the file.`
`ln: filename: File exists`	`A file with the same name as the linkname already exists in the destination directory. Use a different name for the linkname.`

Making links to other user's files

To make a link to a file belonging to another user you need to:

- know the pathname to the file. For more information see "*Pathnames: full and relative*" on page 5-8.

- have appropriate access permissions to the file. For more information see "*Controlling access to files and directories*" on page 5-1.

Only the owner can change a file's access permissions.

Here is an example of a link to a file in another user's home directory:

```
ln ~sandra/prog.c .
```

This makes a link between the file `prog.c` in the home directory of user `sandra` and the filename `prog.c` in the current working directory.

List the files in a directory

You can use the **`ls`** command to list the files in a directory:

```
ls [option] directory_name
```

By combining different command options you can display as little or as much information about each file as you need.

Examples of using the ls *command*

1. To list the contents of the current directory:

```
ls
README       bago.hlp    helpex.hlp  hyperhelp
app-defaults helpex      hoh.hlp
```

This lists the contents of the current directory. File and directory names are listed in columns in alphabetical order.

2. To list each file's type and inode number:

```
ls -Fi
6752 Mail/
102574 SGML/
147215 UNIX_tips/
172802 Xapps/
51200 bin/
44801     dead.letter
44802mbox
44809 test*
102733 tmp/
```

Directories are shown with a trailing `/`; files which have access permissions that allow the file to be executed are shown with a trailing `*`. The inode number of each file is also displayed.

3. To list the contents of directory using wildcard characters:

```
ls *.txt
udi1.txt    wais_x500_www.txt
```

This lists each file in the current directory which has the filename extension `*.txt`.

4. To list files in another directory:

```
ls ../Mail
```

This lists the contents of the directory `Mail` which is at the same level in the directory hierarchy as the current directory.

5. To list each subdirectory and its contents, repeating this recursion:

```
ls -R uunet
```

This will recursively list the contents of each subdirectory there is below the directory uunet.

List of command options

Option	Action
-a	list hidden files
-d	list the name of the current directory
-F	show directories with a trailing '/' executable files with a trailing '*'
-g	show group ownership of file in long listing
-i	print the inode number of each file
-l	long listing giving details about files and directories
-R	list all subdirectories encountered
-t	sort by time modified instead of name

Listing hidden files

The command

```
ls -a
```

lists all the "hidden" files that begin with a `.`(dot). All other files and directories are also listed.

This type of file is often created by you, the user, to hold commands that customise how a paricular program is to work. These files are usually held together in your home directory.

Every directory has two dot files, '`.`' and '`..`' which can be used in a shorthand way to refer to the current directory '`.`'(dot) and the parent directory of the current directory '`..`' (dot dot).

Using a long listing

To get more information about each file and directory, use the command:

```
ls -l
```

This gives you a long listing about each file and directory, giving information about its:

- access permissions
- number of links
- owner
- size
- date and time last modified

▼
You can update the date and time a file was last modified to the current date and time with the **`touch`** command.

```
touch -c filename
```

The `-c` option prevents the *`filename`* from being created if it does not already exist.

Examples of doing a long listing

1. To do a 'long' list of a single file in your current directory:

```
ls -l file1
-rw-r--r-- 2 unixjohn 3287 Apr 8 12:10 file1
```

This displays the following information about the file `file1`.

`-rw-r--r--`	- access permissions
`2`	- number of links to this file
`unixjohn`	- owner
`3287`	- size in bytes
`Apr 8 12:10FR`	- date and time last modified

2. To do a long list of all the files in your current directory:

```
ls -l
-rw------- 1 erpl08 4307   Jun 17 10:00 FAQ.xdh
drwx------ 2 erpl08 1024   Jun 17 10:00 SCCS
-rw------- 1 erpl08 15119  Jun 17 10:00 commands.xdh
-rw------- 1 erpl08 266    Jun 17 10:00 concepts.xdh
```

This displays the following information about every file and directory in your current working directory.

d	- this is a directory
rwx------	- access permissions
2	- number of lins
erpl08	- owner
1024	- size in bytes
Jun 17 10:00	- date and time last modified

Moving files and directories

To move files and directories from one place to another use the **mv** command:

```
mv [option] filename1 filename2
            directory1 directory2
            filename directory
```

You can also change the name of a file or directory by moving it.

✗Caution

Take care what name you give to the file or directory that the file or directory is being moved to.

If you choose the name of a file or directory that already exists it will be replaced by the file or directory that has been moved to its name!

To avoid this use the **mv** command together with its **-i** option. This will prompt you for confirmation whenever the moved file or directory would overwrite an existing file or directory.

You can make this a permanent feature of your UNIX environment by creating a command alias that uses the `-i` option by default. For more information see "*Creating a command alias*" on page 5-25.

Note

The `-i` option is not available with some versions of the command. Check the manual page for the **mv** command on your system.

Examples of using the mv command

1. To rename a file:

```
mv junk precious
```

This renames the file `junk` as the file `precious`.

2. To be prompted when using the mv command:

```
mv -i draft1.txt part1.txt
remove part1.txt?n
```

This command prompts you to confirm that you really want to move the file `draft1.txt` to the file `part1.txt`; a file with that name already exists.

Entering **n** cancels this action: a **y** confirms it.

3. To move a file to another directory:

```
mv precious treasure
```

This moves the file `precious` from the current directory to the directory `treasure`.

4. To move a file to another directory and rename it:

```
mv jewel casket/amethyst
```

This moves the file `jewel` to the filename `amethyst` in the subdirectory `casket`.

5. To move and rename a directory:

`mv vr1.1 ../environments/virtual_rooms`

This moves the directory `vr1.1` from the current directory to the directory `../environments` and renames it as `virtual_rooms`.

Moving linked files

Moving a file that has links to it does not affect the links to the file. Links are only lost when a file is moved from one file system to another. For more information see "*UNIX file system"* on page 5-3.

Printing files

To print a file use the command:

`lpr` *`filename`*

The job is placed in the print queue and you can carry on working.

Some versions of UNIX provide both an **`lp`** and an **`lpr`** command for printing files; other versions support only one or the other. If the **`lpr`** command is not available on your system use the **`lp`** command instead.

Examples of using the lpr *commnand*

1. To print a single file:

`lpr memo`

This prints the file `memo` to your default printer.

2. To print several files:

```
lpr may june july
```

This prints the files `may`, `june` and `july` to your default printer.

Changing your default printer

Most likely you can choose which type of printer to print your file(s) on. Your organisation may provide colour printers, plotters and laser printers as well as simple line printers. Each printer will have its own unique name.

When you print a file it is sent to the default printer that is defined by the value of your `PRINTER` environment variable. This is set up for you when you are given your account.

To change your default printer, change the value of this environment variable to the name of the printer that you want to use. For more information see "*Changing environment variables*" on page 5-7.

Printing PostScript files

You can print PostScript[1] files to a PostScript printer with the standard **lpr** command:

```
lpr filename
```

A PostScript printer automatically recognises that the file consists of PostScript commands and processes it accordingly.

PostScript files can be identified by the filename suffix `.ps`. This type of file can be printed to high quality laser printers that are able to translate the PostScript commands contained in the file into the image to print.

This type of printer is available on most systems.

1. PostScript is the name for a page description language developed by Adobe Systems Ltd.

Previewing files

To preview a text file on your terminal use the command:

```
pr filename
```

The ouput is separated into "pages" of 66 lines, with a 5 line header and footer. The header contains a single line giving the name of the file and other information.

There are a number of options which allow you to change the way in which the file is displayed.

▼

Put the output of the `pr` command through a pager such as `more`. This stops the contents of the file from scrolling out of view on your screen. See "*Displaying files using a pager*" on page 14 of this section.

Examples of using the pr *command*

1. To preview a file:

   `pr chapter3.txt`

   This displays the contents of the file `chapter3.txt` one page after the other.

2. To preview a file using a pager:

   `pr part2.txt | less`

   This displays the contents of the file `part2.txt` with the output piped through the pager program **`less`** to prevent it scrolling out of sight.

3. To change the appearance of the previewed file:

   `pr -l 40 -h Using Hypertext comms_paper | more`

   This prints the file `comms_paper` to the screen, using a page length of 40 lines, with the title "`Using Hypertext`" instead of the filename. The output is piped through the pager program **`more`** to stop it scrolling out of sight.

4. To prepare afile and then print it:

`pr –n2 | lpr`

This prepares the file for printing in two columns, separated into "pages" of 66 lines, with a 5 line header and footer. The header contains a single line giving the name of the file and other information.

The output is then piped into the **`lpr`** command and printed to the default printer.

Printing manual pages

To print a copy of the manual page(s) to a high quality PostScript printer:

`man –t` *`name_of_command`*

For example:

`man –t man`

This prints the manual page to your default printer.

To print a copy of a manual page on a line printer:

`man` *`name_of_command`* `| lpr` *`name_of_printer`*

Using a different printer

To send a file to a different printer use the command:

`lpr –P`*`name_of_printer filename`*

For example:

`lpr –Pps9 paper.ps`

This sends the file `paper.ps` to the printer `ps9`.

Note

If your system only provides the `lp` command, use:

```
lp -d name_of_printer filename
```

Printing multiple copies

1. To print multiple copies of a file use the `#` (hash) option. For example:

   `lpr -#3 memo.txt`

   This prints three copies of the file `memo.txt` on your default printer.

   You can also print multiple copies of several files. For example:

   `lpr -Pps1 -#3 survey.doc letter.txt`

   This prints three copies of the files `survey.doc` and `letter.txt` on the printer `ps1`. On systems that support the `lp` command:

   `lp -d ps1 -n 3 survey.doc letter.txt`

   This has the came action as the previous `lpr` command example.

▼
Do not monopolise the printer! Consider other users and use a Xerox machine if you need to make multiple copies of a file.

Examining the print queue

To display the printer queue use the command:

```
lpq
```

To display the printer que for another printer use the command:

```
lpq -Pprinter
```

Note

If the lpq command is not available on your system use the `lpstat` command instead.

Examples of using the lpq *and* lpstat *commands*

1. To display the print queue for your default printer:

```
lpq
Rank        Owner       Job         Files
active      joe         483         overheads.ps
2nd         sarah       484         /tmp/RFC137
3rd         tony        485         report
```

   This displays the print queue for your default printer: the one defined by your `PRINTER` environment variable.

   Each job in the queue has a job number. Details are also given about the job's position in the queue and the user's name.

2. To display the print queue for a specific printer:

```
lpq -Ppsmath
```

   This displays the print queue for the printer `psmath`.

   The equivalent command on systems that support the `lpstat` command is:

```
lpstat -c psmath
```

Cancelling a print request

To cancel a print request use the command:

```
lprm job_number
```

This removes the print job from the print queue.

You cannot remove another user's print job from the print queue; only the ones that belong to you.

Examples of using the lprm *and* cancel *command s*

1. To remove a print job from the queue for your default printer:

```
lpq
Rank        Owner       Job         Files
active      joe         483         overheads.ps
2nd         sarah       484         /tmp/RFC137
3rd         tony        485         report
```

```
lprm 484
```

The user `sarah` has removed their print job from the the printer queue. On systems that support the use of the lp and lpstat commands, the command

```
cancel printer-jobnumber
```

is used to remove files from the printer queue. Use the `lpstat` command to get the informaiton you need for the argument to the `cancel` command.

2. To remove a print job that has been placed in another printer's queue:

```
lpq -Pps3
Rank        Owner       Job         Files
active      jim         112         doc.ps
2nd         sally       113         presentation.ps
3rd         tom         114         draft
lprm -Pps3 113
```

The user `sally` has removed her print job from the queue for the printer `ps3`.

Working with directories

A directory is a special type of file that holds pointers to other files and other directories. You can create directories in your home directory to hold files and other sub-directories.

Having your own directory structure gives you a definable place to work from and allows you to structure your information in a way that makes best sense to you.

Directories which you create belong to you -you are said to "own" them- and you can set access permissions to control which other users can have access to the information they contain.

Changing to another directory

To change your current working directory use the command:

```
cd pathname
```

where pathname specifies the directory that you want to move to.The pathname can be given as either a full pathname or a relative pathname

Examples of using the cd *command*

1. To move down one level to a subdirectory:

```
cd Firstyear
```

This moves you down one level from your current directory to the subdirectory Firstyear.

2. To move up one level of the directory tree:

```
cd ..
```

Every directory contains a hidden directory .. (dot dot) that is a shorthand name for this directory's parent directory.

Using this shorthand name enables you to move up the directory tree very quickly without having to enter long pathnames.

3. To move to another directory using a relative pathname:

```
cd ../Secondyear
```

This moves you up one level in the directory tree and then moves you into the subdirectory Secondyear.

4. To move to a directory using a full pathname:

```
cd /u/usr/physics/ercy04/ProjectX
```

This moves you to the directory ProjectX which is in the home directory of the user ercy04.

5. To go directly to your home directory:

```
cd
```

Changing to your home directory

Entering the command

```
cd
```

without a pathname returns you to your home directory.

Use ~ character instead of a full pathname

Using the C, Korn, TC or BASH shell enables you to substitute the ~ (tilde) character in place of the full pathname to your own or another user's home directory.

For example, the command

```
cd ~helper
```

changes your working directory to the home directory of the user "helper".

This is very useful if you have access permission to a directory in another users home directory but don’t know the full pathname to it.

Copying directories

To copy a directory use the command:

```
cp -r directory1 directory2
```

This copies `directory1` and everything that it contains to `directory2`. The directory is created if it does not exist.

If `directory2` does exist then `directory1` is created as a subdirectory within it.

▼
Save space by making a link to an existing file instead of creating another copy of it. See "*Linking files*" on page 21 of this section.

Checking directory contents

Before copying or removing a directory list its contents with the command:

```
ls -FR
```

This lists any files and descends any subdirectory trees, listing their contents. For example:

```
ls -FR MetaCard
BLURB.txt BUGS.txt FAQ.txt README.mc metacard/

MetaCard/metacard:
Makefile    XCmdGlue.c  XTtemplate.cXtTemplate.c mc*
README.ext  XCmdGlue.h  XtGlue.c    external.c
XCmdFuncs.c XCmdNames.h XtGlue.h    external.h
```

Examples of using the cp *command*

1. To create a new copy of a directory:

```
cp -r draft presentation
```

This copies all the files and subdirectories in the directory `draft` into a new directory `presentation`.

2. You try to create a copy of a directory using the name of directory that already exists:

```
cp -r planA projects
```

If the directory `projects` already exists, then a copy of the directory `planA` will be created in this directory.

Display the pathname to the current directory

To display the pathname to your current directory use the command:

```
pwd
```

This command has no options.

Example of using the pwd *command*

1. To display the pathname to the current directory:

```
cd
pwd
/usr/graphics/anne
```

The user changes to their home directory and then uses the **pwd** command. This displays the pathname to their home directory `/usr/graphics/anne`.

The path to a user's home directory varies from system to system. Here, the home directory of the user `anne` is located in the directory `graphics`, which is a subdirectory of the directory `usr` which is directly below the root (`/`) directory.

▼
If you use the C,TC or BASH shell, you can use the command

```
dirs -l
```

to display your current directory. This command is built-in to each of these shells. For more information see "*Finding out which shell you are using"* on page 5-3.

Linking directories

To link one or more directories to another directory use the command:

`ln -s` *`directory_name(s) directory_name`*

The use of the `-s` option indicates that this is to be a symbolic link. As a user you are restricted to creating symbolic links between directories using the `-s` option. Only the super-user is allowed make hard links between directories.

Examples of using the ln *command*

1. To link a directory into your current directory:

```
ln -s $HOME/accounts/may .
```

This links the directory `$HOME/accounts/may` into your current working directory. This link will have the same name -`may`- as the directory it points to.

2. To link several directories into a subdirectory:

```
ln -s $HOME/hyperhelp $HOME/metacard experimental
```

This links the directories `$HOME/hyperhelp` and `$HOME/metacard` into the subdirectory `experimental`.

Understanding the location of linked directories

There is a subtle difference between a "normal" directory and a directory created with a symbolic link to another directory.

What happens if you do a **cd** (change directory) to this type of directory? You are placed in the directory that the symbolic link points to!

This can be confusing, especially if you have changed your shell prompt so that it displays the current working directory. For more information see "*Setting the prompt variable: C shell*" on page 5-10.

Look at this example:

```
pwd
# print the user’s working directory
/usr/home/john/tmp
ln -s $HOME/accounts/may fifth
# use a symbolic link named fifth that points to
# the directory $HOME/accounts/may
ls
fifth
cd fifth
# change directory to the new directory
pwd
# print the working directory
/usr/home/john/accounts/may
# the user is actually in the the directory that the
# linked directory points to
```

When the user does a **cd** (change directory) to the symbolically linked directory `may` in their current working directory, they change to the directory that this link points to, the directory `/usr/home/john/accounts/may`.

List the files in a directory

You can use the **ls** command to list the files in a directory:

```
ls [option] directory_name
```

By combining different command options you can display as little or as much information about each file as you need.

For more information see "*Examples of using regular expressions in filenames"* on page 5-5.

Examples of using the ls *command*

1. To list the contents of the current directory:

```
ls
README        bago.hlp    helpex.hlp  hyperhelp
app-defaultshelpex        hoh.hlp
```

This lists the contents of the current directory. File and directory names are listed in columns in alphabetical order.

2. To list each file's type and inode number:

```
ls -Fi
6752 Mail/
102574 SGML/
147215 UNIX_tips/
172802 Xapps/
51200 bin/
44801 dead.letter
44802 mbox
44809 test*
102733 tmp/
```

This lists the contents of the current directory. Directories are shown with a trailing '/'; files which have access permissions that allow the file to be executed are shown with a trailing '*'.

The inode number of each file is also displayed.

3. To list the contents of directory using wildcard characters:

```
ls *.txt
udi1.txt    wais_x500_www.txt
```

This lists each file in the current directory which has the filename extension `*.txt`.

4. To list files in another directory:

```
ls ../Mail
```

This lists the contents of the directory `Mail` which is at the same level in the directory hierarchy as the current directory.

5. To list each subdirectory and its contents, repeating this recursion:

```
ls -R uunet
MetaCard    README      README.ftp
uunet/MetaCard:
BLURB.txt   FAQ.txt     README.mc   metacard
uunet/MetaCard/metacard:
Makefile    XCmdGlue.c  XTtemplate.cXtTemplate.c
README.ext  XCmdGlue.h  XtGlue.c    external.c
XCmdFuncs.c XCmdNames.h XtGlue.h    external.h
```

This will recursively list the contents of each subdirectory there is below the directory uunet.

List of command options

Option	Action
`-a`	list hidden files
`-d`	list the name of the current directory
`-F`	show directories with a trailing '/' executable files with a trailing '*'
`-g`	show group ownership of file in long listing
`-i`	print the inode number of each file
`-l`	long listing giving details about files and directories
`-R`	list all subdirectories encountered
`-t`	sort by time modified instead of name

Listing hidden files

The command

```
ls -a
```

lists all the "hidden" files that begin with a '`.`' (dot). All other files and directories are also listed.

This type of file is often created by you, the user, to hold commands that customise how a paricular program is to work. These files are usually held together in your home directory.

Every directory has two dot files, '`.`' and '`..`' which can be used in a shorthand way to refer to the current directory '`.`'(dot) and the parent directory of the current directory '`..`' (dot dot).

Using a long listing

To get more information about each file and directory, use the command:

```
ls -l
```

This gives you a long listing about each file and directory, giving information about its:

- access permissions
- number of links
- owner
- size
- date and time last modified

Making a directory

To make a directory use the command:

```
mkdir directory_name
```

The access permissions for a directory that you create are set to a predetermined value which ensures that other users cannot get access to your directories and their contents. For more information see "*Controlling access to files and directories"* on page 5-1.

▼

You can also create a new directory by making a link to an existing directory. See "*Linking files"* on page 21 of this section.

Using the /tmp directory

Each UNIX system has a directory `/tmp` which acts as a "scratch area" which you can use to hold files and directories for short periods of time. For example:

```
mkdir /tmp/for_u_tim

cp -r surprise /tmp/for_u_tim
```

This creates the directory for_u_tim in the directory /tmp and then copies the directory `surprise` to it.

User `tim` might have been told that there is "something" for them in `/tmp`.

The access permissions must be set correctly before another user can copy the directory or link to it. For more information see "*Displaying access permissions"* on page 5-1.

Remember to remove files and directories that you copy to or create in the directory `/tmp`; other people need that space too. Do not use the `/tmp` directory to store vital information; it is cleared whenever the system is "booted up" and by the system administrator when the directory gets full.

Examples of using the mkdir *command*

1. To make a directory in the current directory:

```
mkdir specification
```

This creates a new directory `specification` in your current working directory.

2. To make a new directory in the parent directory:

```
mkdir ../presentations
```

This creates the directory `presentations` in the parent directory of the current working directory.

In effect, this creates a new directory at the same level of the file system as the current directory.

3. To make a new directory in a subdirectory:

```
mkdir reports/january/amy
```

This creates the directory `amy` in the subdirectory `repor/january`.

Problems using mkdir

Message	Meaning
mkdir: *directory_name*: File exists	There is already a directory with this name in the location in which you are trying to create the new directory.
usage: mkdir [-p] directory ...	You have entered the mkdir command without giving the name of a directory to be created.
mkdir: *pathname*: Read-only file system	You do not have permission to make files in this directory. You can only list its contents.

Moving files and directories

To move files and directories from one place to another use the mv (move) command:

```
mv [option] filename1 filename2
                              directory1 directory2
                              filename directory
```

You can also change the name of a file or directory by moving it.

✘ Caution

Take care what name you give to the file or directory that the file or directory is being moved to.

If you choose the name of a file or directory that already exists it will be replaced by the file or directory that has been moved to its name!

To avoid this ...

1. Always use the **mv** command together with its `-i` option. This will prompt you for confirmation whenever the moved file or directory would overwrite an existing file or directory.

The `-i` option is not available with some versions of the command. Check the manual page for the `mv` command for your system. For more information see "*Checking the description of a command"* on page 5-19.

Examples of using the mv *command*

1. To rename a file:

```
mv junk precious
```

This renames the file `junk` as the file `precious`.

2. To be prompted when using the **mv** command:

```
mv -i draft1.txt part1.txt
remove part1.txt?n
```

This command prompts you to confirm that you really want to move the file `draft1.txt` to the file `part1.txt`; a file with that name already exists.

Entering **n** cancels this action: a **y** confirms it.

3. To move a file to another directory:

```
mv precious treasure
```

This moves the file `precious` from the current directory to the directory `treasure`.

4. To move a file to another directory and rename it:

```
mv jewel casket/amethyst
```

This moves the file jewel to the filename amethyst in the subdirectory casket.

5. To move and rename a directory:

```
mv vr1.1 ../environments/virtual_rooms
```

This moves the directory `vr1.1` from the current directory to the directory `../environments` and renames it as `virtual_rooms`.

Restrictions on moving directories

Directories cannot be moved from one file system to another. For more information see "*UNIX file system*" on page 5-3.

Removing directories

To remove a directory use the command:

```
rmdir directory_name
```

The directory must be empty before you can delete it. You will need to remove any files and subdirectories that it contains.

To remove a directory that contains files use the command:

```
rm -r directory_name
```

This deletes all the contents of the directory including any subdirectories.

X Caution

If you remove a directory that still holds files there is no way to retrieve it. To avoid this always use the **rm** command together with the `-i` option.

```
rm -ir directory_name
```

Checking directory contents

Before copying or removing a directory list its contents with the command:

```
ls -FR
```

This lists any files and descends any subdirectory trees, listing their contents. For example:

```
ls MetaCard
BLURB.txt BUGS.txt FAQ.txt README.mc metacard/

MetaCard/metacard:
Makefile    XCmdGlue.c  XTtemplate.cXtTemplate.c mc*
README.ext  XCmdGlue.h  XtGlue.c    external.c
XCmdNames.h XtGlue.h    external.h
```

Removing linked directories

To remove a directory that is a symbolic link that points to another directory use the command:

```
rm directory_name
```

Notice that the directory does not need to be empty and you do not have to use the **rmdir** command. You are not really removing a directory, just a named symbolic link that points to the "real" directory.

Removing linked directories that contain links

What happens if you remove a directory that is a symbolic link to another directory and in which you have created links to other files and/or directories?

Remember that the directory is just a linkname pointing to the "real" directory; so it is in this directory that the linknames of files and/or directories exist.

When you remove this type of directory, all that happens is that the linkname of the directory is removed. Any links you have made to the directory remain in the directory that this linkname pointed to.

Look at this example:

```
pwd
#print the user's working directory
/usr/tom/reports
ls ../tmp/drafts
# list the contents of this directory
expenditure targets
ln -s $HOME/tmp/drafts quarterly
# use a symbolic link to make the directory
cd quarterly
# change directory to this linkname
ln $HOME/accounts/expenses
# link a file to this linkname
ls
# list the contents of the linked directory
# the linked file is listed
expenditure expenses targets
cd ..; rm quarterly
# go to the parent directory and remove the linkname
ls ../tmp/drafts
# list the directory the linkname pointed to
expenditure expenses targets
# the linked file is still there
```

Examples of removing directories

1. To remove an empty directory:

```
rmdir docs
```

This removes the empty directory `docs` from the current directory.

2. To remove a directory containing files and subdirectories:

```
rm -r projectX
```

This deletes the directory `projectX` and any files or subdirectories that it holds.

3. To remove a directory interactively:

   `rm -ir plan9`

   You will then be prompted for confirmation before a file or subdirectory is removed.

   To remove an item, simply press the `<Return>` key. You do not have to enter `y` for Yes. To retain an item enter `n` for No.

Problems using the `rm` command

Message	Meaning
`rm` *`filename`*`: No such file or directory`	`The file or directory you are trying to copy does not exist.`
`usage: rm [-rif] file ...`	`You have used a non-existent option or you have failed to give a` *`filename`* `or` *`directory_name`*`.`
`rm: override protection 600 for` *`filename`*`?` **`y`** `rm:` *`filename`* `not removed: Permission denied`	`You don't have permission to remove this file or directory. Most likely you're trying to erase a system file or another user's file.`

Renaming directories

Directories are renamed by moving them using the **`mv`** command. See "*Moving files and directories*" on page 49 of this section.

7

Managing your environment

7 *Managing your working environment*

Your working environment is defined whenever you login or start another shell. This environment is set using the values that the shell finds in initialisation files which it always reads as it starts up.

You can change your working environment by editing these files and setting new values for these *variables*. Each shell handles its initialisation files in a different way; read the relevant section for the shell that you use.

For more information see "*Finding out which shell you are using*" on page 11-3.

Displaying current variables

You can use the `echo` command to display the value of an environment variable. For example:

```
echo $PATH
```

This displays the value of environment variable `PATH`.The name of an environment variable is usually given in `UPPERCASE`. The `$` sign is a shell metacharacter that refers to the value of the variable instead of its name.

The method for displaying the value of every variable varies according to which shell you are using.

For the Bourne shell use the command **`set`** to display the value of environment variables.

For the C and TC shell use the command **`printenv`** or **`env`** to display the value of environment variables and the command **`set`** to display the value of local shell variables.

If the list of variables is so long that it scrolls off the screen and you want to display the information one screenful at a time, pipe the output through a pager. For example:

```
set | more
```

Examples of showing the value of variables

1. In the C and TC shell, to show value of all your environment variables:

```
printenv

TERM=vt100
HOME=/u/user12/eucs/helper
SHELL=/usr/local/bin/Tcsh
USER=helper
TZ=GMT0BST
UCBPATH=:/usr/ucb:/bin:/usr/bin
ATTPATH=:/bin:/usr/bin
UCBSHELL=/usr/local/bin/Tcsh
ATTSHELL=/bin/sh
LOGNAME=helper
MAIL=/u/user12/eucs/helper/.mail
```

This shows that this user's terminal type is set to VT100, that they use the TC shell and that they are on Greenwich Mean Time.

2. In the Bourne, BASH and Korn shell, to show the value of all your variables:

```
set

BASH=/usr/local/utils/bin/bash
BASH_VERSION=1.12.1
DELIVERY=John Smith Room 2004 JCMB
DISPLAY=frolix8.ucs:0
EDITOR=ue
```

```
FOLDERS=/u/user/eucs/erpl08/Mail
HISTFILE=/u/user/eucs/erpl08/.bash_history
HISTFILESIZE=50
HISTSIZE=100
HOME=/u/user/eucs/erpl08
...
```

This shows the value of an assorted range of variables.

Finding out which shell you are using

Information about which shell you are using is held in the `SHELL` environment variable. The command

```
echo $SHELL
```

displays the value of this variable. You can identify which shell you are presently using fromthe last part of the pathname.

Pathname	Shell
/.../sh	Bourne shell
/.../csh	C shell
/.../tcsh	TC shell
/.../ksh	Korn shell
/.../bash	Bourne Again SHell

Changing your environment: Bourne shell

At login the Bourne shell reads the initialisation files`/etc/profile` and `$HOME/.profile`. You cannot change the content of `/etc/profile`, but you have permission to edit the contents of `.profile`, which is in your home directory and is owned by you.

▼

Make a copy of your shell start up file before editing it to change environment or shell variables.

You can always return to using this copy if you find that the modified file is not working and you are unable to fix the problem. For example:

```
cd
cd .profile old_profile
ue .profile
```

The first command changes you to your home directory,where a copy of the file `.profile` is made in `old_profile`. You can then edit the `.profile`.

To revert to using the original .profile you simply copy it back. For example:

```
cp old_profile .profile
```

Making a temporary change

To make a temporary change to the value of a set variable enter the command:

```
$ VARNAME=value_of_variable
```

This value will remain until you logout from the system or exit from the shell you are working in. To make other programs that use the variable aware of the new value enter the command:

```
$ export VARNAME
```

Examples: Bourne shell

1. To change to using another printer for the current login session:

```
PRINTER=ps2
```

Whenever the user prints a file, the output will be sent to the printer `ps2`.

2. To change to using another editor for the current login session and inform (export) this value to other programs which use it:

```
EDITOR=vi
export EDITOR
```

This changes the user's default editor to `vi` and exports this value to other programs -such as electronic mail- which make use of it.

3. To change to using another pager for viewing files add the following lines to your `.profile` file.

`PAGER=less;export PAGER`

To make this change take immediate effect enter the command:

`source .profile`

This causes the `.profile` file to be read, setting the variables to their new values.

Making lasting changes

To make lasting changes to the value of an environment variable:

1. Use your editor to open the `.profile` file and add the line:

`VARNAME=`*`value_of_variable`*`;export VARNAME`

2. Save the file and leave the editor.

3. Enter the command:

`..profile`

This adds the new value to your working environment.

Changing your environment: C and TC shells

At login the C (and TC shell) reads initialisation files in your home directory. First it reads the file `.cshrc` (`.tcshrc` for TC shell) followed by the file `.login`.

In addition to environmental variables, both these shells have a number of local shell variables which are not exported to the environment.

When another shell is created to run a program, it only reads the `.cshrc` (`.tcshrc`) file, but it inherits any environment variables that were set up at login.

This procedure is designed to allow the the login shell to be set up to behave differently to subsequent shells. Since environment variables only need to be initialised once at login, the behaviour of other shells can be changed by setting shell variables in the `.cshrc` which is read each time a new shell is invoked.

Usually, you are are advised to set environment variables in your `.login` file and shell variables in your `.cshrc`. This arrangement prevents your environment being re-initialised every time a shell is started.

If you use a network windowing systems such as X, it is likely that you may start up a shell on a remote host which is not a login shell. If this is the case, any environment variables set in your `.login` on that host will not become part of your environment.

Puzzled? Don't be. The simplest solution is to set *both* environment *and* shell variables in your `.cshrc` (or `.tcshrc`) file.

▼
Make a copy of your shell start up file before editing it to change environment or shell variables.

You can always return to using this copy if you find that the modified file is not working and you are unable to fix the problem. For example:

```
cd
cd .cshrc old_cshrc
ue .cshrc
```

The first command changes you to your home directory,where a copy of the file .profile is made in old_profile\FR.You can then edit the .profile.

To revert to using the original .profile you simply copyit back. For example:

```
cp old_cshrc .cshrc
```

Changing environment variables

To make a temporary change the value of an environment variable enter the command:

```
% setenv VARNAME value_of_variable
```

This value will remain until you logout from the system or exit from the shell you are working in. By convention, the names of environment variables are given in `UPPER CASE`.

To make lasting changes to the value of an environment variable:

1. Use your editor to open the .cshrc (.tcshrc) and add the line:

```
setenv VARNAME value_of_variable
```

2. Save the file and leave the editor.

3. Enter the command:

```
source .cshrc (.tcshrc)
```

This adds the new value for the variable to your environment.

Examples: C and TC shell

1. To change to using another printer for the current login session:

```
setenv PRINTER ps2
```

Whenever the user prints a file, the output will be sent to the printer `ps2`.

2. To change to using another editor for the current login session and inform (export) this value to other programs which use it:

```
setenv EDITOR vi
```

This changes the user's default editor to vi and exports this value to other programs -such as electronic mail-which make use of it.

3 To change to using another pager for viewing files the user adds the following lines to their `.cshrc` (`.tcshrc`) file.

`setenv PAGER less`

To make this change take immediate effect enter the command:

`source $HOME/.cshrc`

This causes the `.cshrc` file to be read, setting the variables to their new values.

Changing shell variables

The C and TC shells have a number of local shell variables which are not exported to the environment. To make sure that these variables are picked up whenever a new shell is invoked, always define shell variables in your `.cshrc` (`.tcshrc`) file.

By convention, names of shell variables are always given in lower case letters. UNIX is case sensitive, so the environment variable `PATH` is *not* the same as the shell variable `path`.

If a shell and an environment variable have the same name, the value of the local shell variable will be used. You can avoid problems by making sure that shell and environment variables with the same name are set to the same value in your `.cshrc` (`.tcshrc`).

For example, suppose you set the following variables in this order in your `.cshrc` (`.tcshrc`) file:

```
setenv TERM vt100
...
...
set term=xterm
```

When a shell is invoked, the value of the environment variable `TERM` is copied into the shell variable `term`. But the value of this shell variable is set later on in the `.cshrc` (`.tcshrc`) file, and it is this value which is then copied back into the environment value `TERM` which is then exported to the general environment.

Setting shell variables

To define a temporary value to a shell variable use the command:

```
set variable_name=value
```

The value of this variable will remain set until you exit from this shell. The value of this variable is not exported to other C (or TC) shells when they are invoked.

To give a lasting value to a shell variable:

1. Use an editor to open your `.cshrc` (`.tcshrc`) file and add the line:

   ```
   set variable_name=value
   ```

2. Save the file and leave the editor.

3. Enter the command:

   ```
   source .cshrc
   ```

   This adds the value of the shell variable to your shell's present environment. When other C shells are invoked they will also have this variable set.

Safeguarding your files

It's easy to overwrite (clobber) existing files when you are redirecting output to a file. To prevent this from happening you need to set the `noclobber` shell variable by adding a line to your *`.shellrc`* file.

Depending on which shell this is:

Shell	Line to add
C shell	set noclobber
TC shell	set noclobber
Korn shell	set -o noclobber
BASH	set noclobber

Setting the prompt variable: C shell

You can customise the prompt so that it displays your current working directory, username on the system and the name of the system; or any combination of these.

How this is done varies according to the shell you are using.

To display your current working directory in the prompt:

1. Use your editor to open your `.cshrc` file and add the line:

```
set prompt = "$cwd:t %"
```

2. Save the file and leave the editor.

3. Enter the command:

```
source .cshrc
```

This adds the new values to your working environment.

To display your username in the prompt:

1. Use your editor to open your `.cshrc` and add the line:

```
set prompt = "$LOGNAME %"
```

2. Save the file and leave the editor.

3. Enter the command:

```
source .cshrc
```

This adds the new values to your working environment.

To display the name of the system in the prompt:

1. Use your editor to open your `.cshrc` file and add the line:

```
set prompt="%m %"
```

2. Save the file and leave the editor.

3. Enter the command:

```
source .cshrc
```

This adds the new values to your working environment.

Setting the prompt variable: TC shell

Display current working directory

To display your current working directory in the prompt:

1. Use your editor to open your `.tcshrc` and add the line:

```
set prompt='%. %'
```

2. Save the file and leave the editor.

3. Enter the command:

```
source .tcshrc
```

This adds the new values to your working environment.

To display your username in the prompt:

1. Use your editor to open your `.tcshrc` and add the line:

```
set prompt = "${user} %"
```

2. Save the file and leave the editor.

3. Enter the command:

```
source .tcshrc
```

This adds the new values to your working environment.

To display the name of the system in the prompt:

1. Use your editor to open your `.tcshrc` file and add the line:

```
set prompt='%m %'
```

2. Save the file and leave the editor.
3. Enter the command:

```
source .tcshrc
```

This adds the new values to your working environment.

8

Text editors

8 Text editors

While using UNIX you will often want to create a text file and then change its content in some way. A text editor is a program that has been designed especially for this purpose.

What is a text editor?

A text editor is a program for inserting or amending text in a file. A text editor is not a word processor although some text editors do include word processing facilities.

What is a file?

A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, or a simple letter. Sometimes you may import a file from elsewhere, for example from another computer. If you want to enter your own text or data, you will start by creating a file.

Whether you copied a file from elsewhere or created your own, you will need to return to it later in order to edit its contents.

Which editor should I use?

This section describes the two most widely used editors in the UNIX environment; `vi` and `Emacs`. The `vi` editor is a standard feature of every UNIX system and `Emacs` (or a version of it) is also available on most. While both editors provide all the features needed for editing work, each is different from the other in the way in which you use it.

Which editor you use is a matter of personal preference: try each of them and then decide. One word of advice; as you begin to make regular use of one editor, you will find it harder to switch to using another.

Using the vi editor

The vi editor (pronounced "vee eye") is available on all UNIX systems: other editors are not. Being able to use vi ensures that you will always have an editor available to you.

Basic concepts

Underlying the use of vi are two very important concepts; *operating modes* and the *text buffer.*

Operating modes

There are two modes in which you use `vi`.

Command mode

This is the mode you are in whenever you begin to use `vi`. In this mode commands are used to move around and edit text objects such as words, sentences and paragraphs.

Pressing the ESC key always returns you to command mode.

Insert mode

This is the mode you use to type (insert) text into a buffer. There are several commands that you can use to enter this mode.

Text buffer

While using `vi` to edit an existing file, you are actually working on a copy of the file that is held in a temporary buffer.

If you invoked `vi` with a new filename, (or no file name) the contents of the file only exist in this buffer.

Saving a file writes the contents of this buffer to the permanent file, replacing its contents. You can write the buffer to a new file or to some other file.

You can also decide not to write the contents of the buffer, and leave your original file unchanged.

About vi commands

While not entering text you use `vi` in command mode and enter editing commands consisting of one or more characters.

The syntax of an editing command is:

`operator object`

The operator is a character -such as **`d`** for delete- that describes the action to be carried out on the object; a word, sentence, paragraph or section.

To return to command mode from insert mode press the <`ESC`> key. If you are already in command mode the terminal will beep at you (or make some similar sound).

Running UNIX shell commands

You can run UNIX commands and see their output without leaving `vi`. You can also insert the output of a UNIX command into the file you that are editing.

To run a single UNIX command use the command:

`:!``unix_command`

You can start a shell from within `vi` and use it as you would your usual UNIX environment, then exit the shell and return to `vi`

To start up a shell enter the command:

`:sh`

The type of shell that is started is determined by the value of the `$SHELL` variable. You can specify that some other shell is to be started by setting the `vi` shell option to relevant sections. See "*Customising vi"* on page 16 of this section.

Return to using `vi` by entering the command **`exit`** or **`Ctrl-D`**

Insert output of a UNIX command

To insert the output from a UNIX command into a file, immediately after the cursor:

```
:r!command
```

For example, this facility would be very useful if you were using `vi` to document a UNIX command and you wanted to include examples of the output from this command.

Starting vi and opening files

To start using `vi` enter the command:

```
vi filename
```

The filename can be the name of an existing file or the name of the file that you want to create.

Examples of startng vi *and opening files*

1. To open an existing file for editing:

   `vi intro`

   This opens the file `intro` for editing. This file already exists in the current directory.

2. To create a file to edit:

   `vi part2`

   This command starts `vi` with a new file named `part2`. This file does not exist in the current directory until it is written to using the write command.

3. To simply start `vi`:

   `vi`

This command places you in `vi`. You can then enter and edit text and save (write) it to a file.

Saving files and exiting vi

There are several ways in which you can end your editing session, leave `vi` and return to the shell.

Saving changes

To quit `vi` and save (write) the contents of the buffer to the file that `vi` is being used to edit:

```
ZZ or :wq
```

Without saving changes

To quit `vi` without saving the contents of the buffer to the file:

```
:q!
```

Saving to another file

To quit `vi` and save the contents of the buffer to a new file:

```
:w filename
:q
```

Quit vi

To quit the `vi` editor:

```
:q
```

If you have edited (changed) the file you will be prompted with the message:

```
No write since last change
```

You can either save the changes or quit without saving any of the changes you have made.

Recovering files after a system crash

Suppose the system fails while you are editing a file! You can recover-the file when you next log on to the system again by using the `-r` option to the `vi` command. For example:

```
vi -r help.xdh
```

This restores the `vi` buffer for the file `help.xdh` to exactly the state it was in immediately before the system failed.

You can continue to edit the file or write the contents of the buffer to the file.

Viewing a file

You can look at a file without the risk of altering its contents by using `vi` in "read only" mode. To do this, use the `view` command:

```
view filename
```

The message `[Read only]` is displayed at the bottom of the screen.

You then use the same commands as you would in `vi` to move around, search for text and quit `view`.

Entering text

To type text into the buffer you must enter insert mode. Choose the command that is best suited to the present position of the cursor and the editing task you want to do.

As the text you are typing in begins to approach the right-hand side of the screen press the `<Return>` key to start a new line.

Why enter carriage returns?

For `vi`, a line is the string of characters between two <RETURN> characters.

Although `vi` will automatically "wrap" the text onto a newline when it reaches the right-hand side of the screen, it does not insert a `<RETURN>` character. Lines of text as they appear on your screen do not necessarily correspond to how `vi` understands their structure. This can give you problems when you use `vi` commands to move around the text as you edit it.

Adding a `<RETURN>` at the end of every line ensures that the appearance of the text on your screen corresponds to `vi`'s understanding of how text is structured.

Setting the wrap margin

Setting the wrapmargin option (`vm`) to a value greater than `0` (zero) automatically inserts a `<RETURN>` character whenever `vi` wraps text. See "*Customising vi*" on page 16 of this section.

▼

As you are entering text, make sure that each sentence ends with a fullstop followed by two or more spaces. You can then be sure of using `vi`'s editing commands to manipulate the sentences in your text.

If necessary, use `vi`'s global replace command to change sentence endings that do not meet this convention.

List of insert commands

To do this ...	Command
Insert text after the cursor	**a**
Insert text before the cursor	**i**
Append text at the end of the current line	**A**
Insert text at the start of the current line	**I**
Open a new line above the current line	**o**
Open a new line below the current line	**O**

Changing text around

There are several ways in which you can change the structure and the content of the file you are editing.

Repeat parts of the text

You can re-use a piece of text in another part of the file by "yanking" it into a temporary buffer and then putting the contents of this buffer back into the file.

The command `y` yanks the text into the temporary buffer. For example:

1. To yank the current sentence into the temporary buffer:

```
y)
```

2. To yank the text from the present cursor position to the beginning of the current paragraph, into the temporary buffer:

```
y{
```

Combined with other commands that are used to move around text objects, the yank command can be used to pull any part of a file into the the buffer.

The command **`p`** pastes the contents of the temporary buffer back into the file immediately after the cursor position.

Insert the contents of another file

To insert the contents of another file use the command:

```
:r filename
```

This inserts the file immediately after the current line.

1. To insert a file immediately after the line containing the cursor:

```
:r examples.txt
```

This inserts the contents of the file examples.txt immediately after the current line.

2. To insert the contents of another file after a specific line in the current file:

```
23:r toc.txt
```

This inserts the contents of the file "toc.txt" after line 23 of the current file. Line 24 is empty; the contents of the inserted file begin at line 25.

Cut and paste text

Moving pieces of text around within a file is referred to as cut and paste. Doing this in `vi` is a three stage process:

1. Cut the text you want to move with one of the commands used to delete text.
2. Move the cursor to where you want to paste the text into the file using one of the commands for moving around the text.
3. Paste the text into the file with the `vi` command:

```
P
```

Moving around the text

There are a range of easy to use commands that provide a great degree of flexibility in how you move around the contents of the file you are editing.

Many of these commands can be combined with a numerical value to extend their functionality.

Moving the cursor along a line

Press the ESC key to enter command mode before using these commands.

To move to	Do this ...
next character	**l**
previous character	**h**
next word	**w**
next n words	**w***n*
previous word	**b**
previous n words	**b***n*
end of current word	**e**
start of current line	**0** (zero)
end of current line	**$**

Moving the cursor between lines

Press the ESC key to enter command mode before using these commands.

To move to	Do this ...
next line down (same column)	j
start of next line down	+
previous line (same column)	k
start of previous line	-

Moving between text blocks

Press the `ES` key to enter command mode before using these commands.

To move to	Do this ...
beginning of next sentence	)
beginning of previous sentence	**(**
beginning of next paragraph	**}**
beginning of previous paragraph	**{**

A full stop followed by two or more spaces is treated as the end of a sentence.

All text to the next empty line is treated as belonging to the current paragraph.

Moving over the file

Press the `ESC` key to enter command mode before using these commands.

To move to	Do this ...
top of the file	**1G**
bottom of the file	**G**

Moving around the screen

There are commands for moving around the screen you are viewing and for moving around the file using the screen view of the file.

Move around the screen

Press the `ESC` key to enter command mode before using these commands.

To move to	Do this ...
top of the screen	**H**
middle of the screen	**M**

last line of the screen	**L**
last line but *n*	**L***n*

Moving between screens

Press the ESC key to enter command mode before using these commands.

To move	Do this ...
forward one screen	**^F**
backward one screen	**^B** ^U
forward half screen	**^D** (down)
backward half screen	**^U** (up)

Deleting text

Text is deleted by combining the delete commands x and d with the type of text object to be deleted.

Numbers can also used with both the x and d commands to delete multiple instances of a text object. For example:

```
5dw
```

This deletes the current word and the following four words.

Deleting characters

Press the ESC key to enter command mode before using these commands.

To delete	Do this
current character	**x**
previous character	**dh**

Deleting words and lines

Press the ESC key to enter command mode before using these commands.

To delete	Do this
current word	**dw**
previous word	**db**
entire line	**dd**
to end of line	**d$**
to start of line	**d0** (zero)
next *n* lines	*n***dd**

Deleting sentences and paragraphs

Press the ESC key to enter command mode before using these commands.

To delete	Do this
to end of sentence	**d)**
to beginning of sentence	**d(**
to end of paragraph	**d}**
to beginning of paragraph	**d{**

Undeleting text

To get back a word or line that you have just deleted enter the command:

```
P
```

Searching for text

Press the ESC key to enter command mode before using these commands.

To search	Do this ...
forward for a pattern	/*pattern*
backward for a pattern	**?***pattern*
repeat previous search	**n**
repeat previous search in reverse direction	**N**

Replacing text

Press the ESC key to enter command mode before using these commands.

To replace	Do this ...
pattern1 with pattern2 on the same line	**:s/***pattern1***/***pattern2*
every occurrence of pattern1 with pattern2	**:g/***pattern1*/s//*pattern2***/g**

✗ Caution

Use global replacement in its interactive mode to avoid making changes that you don't intend. To do this add the `c` option to the end of the global replacement command:

`:g/`*`pattern1`*`/s//`*`pattern2`*`/gc`

`vi` will display the first occurrence of *`pattern1`*.

To replace it with *`pattern2`* you must enter **`y`** (yes) and press `<RETURN>`.

If you don't want to replace this pattern press `<RETURN>` to go to the next occurence.

Examples of replacing text

1. To replace one word in the current line withanother:

```
:s/that/which
```

This replaces the first instance of the word “that” in the current line with the word “which”.

2. To replace every instance of one word in thecurrent line with another:

```
:s/destroy/delete/g
```

This replaces every occurrence of the word “destroy” with the word “delete” in the current line.

3. To replace every occurence of a word throughout the file, with another:

```
:g/insert/s//add/g
```

This replaces every occurence of “insert” with “add”.

4. To interactively replace every occurence of a word throughout the file, with another:

```
:g/in conjunction with/s//with/gc
option used in conjunction with the ls command
            ^^^^^^^^^^^^^^^^^^^
```

This displays the first line containing an occurence of the pattern “`in conjunction with`”, and identifies this occurence by underlining each character with a ^ (caret).

Entering **y** (yes) replaces this instance of “`in conjunction with`” by “`with`” and proceeds on to the next.

Pressing <RETURN> leaves this instance unchanged and proceeds to the next.

Customising vi

You can change the way `vi` operates by changing the value of certain options which control specific parts of the `vi` environment.

Displaying vi option values

To display the current value of all options enter the command:

```
:set all
```

To display the value of those options whose values have been reset from their default enter the command:

```
:set
```

Change an option value temporarily

To change the value of an option temporarily:

```
:set option_name
```

or:

```
:set option_name=value
```

This sets the value of the option until you quit `vi`.

List of useful vi options

Option		Action
shell	noai	automatically indents each line of text
number	nonu	prefix lines with line number
term	$TERM	sets type of terminal to use
scroll	scroll=11	sets number of lines scrolled with ^d and z
shell	sh=/bin/sh	defines the shell used for UNIX commands. Value is taken from SHELL environment variable
shiftwidth	sw=8	sets thenumber of places for use with autoindent
wrapmargin	wm=8	specifies the number of spaces lines are broken before right edge of screen

Examples of setting options temporarily

1. To set the wrap margin:

```
:set wm=5
```

This sets the wrap margin to five characters. In insert mode, text will automatically wrap to a new line when the cursor is five characters fromthe right edge of the screen.

2. To display the line numbers:

```
:set number
```

This causes vi to display a line number (starting at 1) at the start of every line of text.

Change an option value permanently

To make a lasting change, create a file named `.exrc`, containing the `set` commands, in your home directory. The next time you use `vi` these options will take effect and will remain in force until you edit the `.exrc` file to change them.

You can, of course, temporarily change the value of any option. See table m for alist of useful options.

Examples of setting options permanently

1. To set a number of options place the set commands in the file `.exrc`.

```
set ic
set number
set sh=/usr/local/bin/Tcsh
set wm=5
```

This sets `vi` to:

- ignore the case of characters in searches

- display line numbers

- use the TC shell to execute UNIX commands

- wrap text five characters from the right edge of the screen

2. Options can also be set using the environment variable `EXINIT`.

```
setenv EXINIT 'set ic number sh=/usr/local/bin/Tcsh wm=5
```

For the C and TC shell user, this sets the same options as in the example above.

If there is a `.exrc` file owned by you in your home directory or the current directory, vi will ake its options values from this; not from the the `EXINIT` environment variable.

Using the Emacs editor

This section describes how to get started with the `Emacs` editor. The features described here are those provided by GNU Emacs 18.57, from the Free Software Foundation, Inc. For details on how to perform more complicated functions you will need to consult the Emacs Reference Manual, read the manual page or use the `Emacs` online help facility.

Entering commands

To start using the 'toolkit' of editing instructions known collectively as `Emacs` enter the command:

```
emacs
```

You are then placed 'in' `Emacs` and use instructions that enable you to add, delete and manipulate text in the file.

Using Emacs commands

You can type `Emacs` instructions in upper or lower case. There are two types of instruction, CONTROL and META.

CONTROL instructions

These are typed while holding down the `CTRL` key which is signified by the `^` character. For example `^F` is generated by pressing the `F` key while holding down the `CTRL` key.

META instructions

These are typed after pressing the `ESC` key which is signified by `M-`. For example, `M-F` is generated by pressing the `ESC` key and then typing the letter `F`. Throughout this document, the `ESC` key is referred to as the `META` key.

Sometimes a `META` or `CONTROL` instruction will prompt you for more information. This information may be another `META` or `CONTROL` instruction or some text.

Creating a new file

To create a new file you first need to choose a name for it. It is best to give it a name which is meaningful to you, for example, if you are writing a letter call the file `letter`.

You can create a new file in two ways.

Creating a file using a filename

Type

`emacs` ***`filename`***

To create a new file called letter, you would type

```
emacs letter
```

The screen of your terminal will fill with blank lines and the following message will be displayed at the bottom:

```
----Emacs:letter-----------(Fundamental)---All-----
```

This consists of:

- the status of the current buffer
- the name of the temporary work file used to store the file you are editing (the computer term for this is 'buffer')
- the name of the file that text will be written to
- the mode which `Emacs` is currently in
- the position of the viewport into the buffer

The bottom line of your screen is used for messages from `Emacs` and for prompting you for information which you may need to type.

Creating a file without using a filename

Type

```
emacs
```

without a filename. This starts Emacs without a file.

The screen of your terminal will fill with blank lines and the following message will be displayed at the bottom:

```
----Emacs:*scratch*-----------(Fundamental)---All---
```

Remember to save to a new file if you create a file in this way. See "*Saving and closing files*" on page 23 of this section.

Opening an existing file

You may want to open an existing file to change its contents. To do this type:

```
emacs filename
```

To open the file called letter you would type:

```
emacs letter
```

▼

It is good practice to make a copy of your original file; in the event of problems you can always revert to the original.

Entering text

When you start `Emacs`, the cursor will be at the top left hand corner of your terminal screen. Any text you now type at your keyboard will be placed before this cursor. This means the cursor will move to the right each time you type in a character.

When you are typing in text and you reach the end of the line, the cursor will move to the start of the next line of your screen. This is known as 'word wrap'.

Saving and closing files

You can save a file in several ways, some of which may not save your most recent work! Make sure that you understand the action of a particular type of 'save' before using it.You do not want to lose all that hard work you have just done.

Save and continue editing

To save contents of the buffer to a file type the instruction `^x ^s`. You will then see the prompt

```
File to save in: ~/
```

at the bottom of the screen. As a courtesy, `Emacs` supplies the pathname to your current directory. You can either enter the filename or back up with the `<Delete>` key and define an alternative pathname. For example:

```
File to save in: /tmp/party_in.txt
```

Press the <Return> key and `Emacs` will then inform you that it has written the contents of the buffer to a file. For example:

```
Wrote /tmp/party_in.txt
```

When you next save the file, `Emacs` will simply write the present contents of the buffer to the file. You are not prompted to give a filename.

Saving to a new file

To write to a new file type `^X^W` and give the name of a file in answer to the `Write file:` prompt.

Any subsequent editing you do will affect this new file. The old file will be left as it was when you last saved it.

▼

You can type `^G` to cancel a command.

✗ Caution

If the file you specify already exists, its contents will be overwritten.

Leaving a file without saving changes

It is very useful to be able to abandon a file without saving any of the changes you have made since the last time you saved it. To do this type `^X^C`. You are then prompted to confirm that you want to save the current buffer to the file:

```
Save file /u/user20/eucs/erpl08/tmp/more_trash? (y or n)
```

Answer **n** and then confirm with a **yes** to the following prompt:

```
Modified buffers exist; exit anyway? (yes or no)
```

The `Emacs` program then exits and you are returned to the shell prompt.

Moving round the text

Here are the `Emacs` instructions for moving round a file.

Move along a line

To...	Do this...
Move cursor to next character	^F
Move cursor to previous character	^B
Move cursor to next word	M-F
Move cursor to previous word	M-B
Move cursor to start of line	^A
Move cursor to end of line	^E

Moving between lines

To...	Do this...
Move cursor to next line	^N
Move cursor to previous line	^P

Moving between screens

To...	Do this...
Move to next screen	^V
Move to previous screen	M-V

Moving over the file

To...	Do this...
Move cursor to the top of the file	M-<
Move to the bottom of the file	M->

Deleting text

To...	Do this...
Delete the character to the left of the cursor.	Press the DELETE key
Delete the character under the cursor.	^D
Delete from cursor to the end of a word.	M-D
Delete from cursor to the start of a word.	M-<Delete>
Delete from cursor to the end of a line.	^K
Delete an entire line.	^A^K
Undelete a word or line	^Y

Moving, copying and deleting blocks of text

A block of text is referred to in `Emacs` as a 'region'. It is defined by the position of a special 'mark' at one end, and by the cursor (referred to as the 'point') at the other.

To define the 'region'

1. Move the cursor to the start of the text block you want to define.

2. Type **`^<Space>`** (hold down the `<Ctrl>` key and press the `<Spacebar>`).

This sets a 'mark' at the start of the 'region'.

3. Move the cursor to the end of the text block you want to define as the 'region'.

This sets the 'point' at the end of the 'region'. You do not have to type anything to set this 'point', the 'point' is the location of the cursor.

▼
You can check the location by typing **`^X ^X`**. This moves the cursor back and forth between the start and end of the region.

Moving a block of text

Moving text amounts to picking up words and placing them in a new position.

To move a block of text do the following:

1. Define the 'region'.

2. Type **`^W`** to 'wipe' the 'region' and place its contents in the 'kill buffer'.

3. Move the cursor to the place where you want the text to go.

4. Type **`^Y`** to 'yank' the text back.

A copy of the text moves from the 'kill buffer' and appears in its new location. The text remains in this buffer until it is replaced by something else or until you finish using `Emacs`. As long as the text is in this buffer you can continue inserting it into different places.

Copying a block of text

You can leave text in its present location and move a copy of it to a different location. This is useful when you have a special paragraph that you want to repeat in a number of locations.

To copy a block of text:

1. Define the 'region'
2. Type the **`M-W`** instruction to 'copy' the 'region'.
3. Move the cursor to the place where the copied text is to go.
4. Type `^Y` to 'yank' the text.

A copy of the text moves from the 'kill' buffer and appears at its new location.

Deleting blocks of text

To delete a block of text:

1. Define the 'region'
2. Type `^W` to 'wipe' the 'region'.

▼

If you erase a 'region' by mistake, you can get it back if you immediately type the `^Y` instruction.

Recovering from mistakes

If you make a mistake, such as deleting a line or block of text inadvertently, you can recover from this mistake by 'undoing' the previous editing command. To do this enter the command:

^_

Press and hold down the `<Ctrl>` ley and then type an underscore character by holding down the `<Shift>` key and typing a – (hyphen).

In fact, you can recursively undo an almost unlimited number of previous editing actions.

Searching for text

`Emacs` provides searching instructions which enable you to move to specific text.

▼
Type `^G` to cancel a search or replace dialogue.

Search forward

The search forward instruction searches for text from the cursor to the end of the file. If you want to search the whole file move to the start of the file

To search forward:

1. Type **`^s`**

   You will be prompted for the string you want to look for. The prompt will look like:

   ```
   I-search:
   ```

2. Type in the text to search for.

As you enter the text to search for, the cursor moves forward through the file to the first occurrence of the text you have typed so far.

To repeat a search

To continue searching forward through the file for the next occurrence of the text string type the `^S` command again and so on.

Search backwards

The search backward instruction searches for text from the cursor to the start of the file. If you want to search the whole file move to the end of the file.

To search backwards:

1. Type **`^R`**

   You will be prompted for the string you want to look for. The prompt will look like:

   ```
   I-search backward:
   ```

2. Type in the text you want to search for.

As you enter the text to search for, the cursor moves backward through the file to the first occurrence of the text you have typed so far.

To repeat a search

To continue searching backward through the file for the next occurrence of the text string type the `^R` command repeatedly.

Stopping a search

To stop searching through the file press the **`Esc`** key. To return to the point in the file where you started the search from enter the command **`^G`**.

Replacing text

The next step from an instruction which asks 'Find me this text', is one which asks 'Find me this text and replace it with that text'.

Global replace

To replace every occurrence of a piece of text from the cursor to the end of the file:

1. Type **`M-%`**.

This displays the prompt:

```
Query replace:
```

2. Type in the text you want to replace and press the `<Return>` key. You will then be prompted for the new text, with a prompt like this:

```
Query replace string with:
```

3. Type in the new text and press the `<Return>` key.

The cursor then moves to the first occurrence of the text you want to replace.

4. Type an !

All the occurrences of the old text will be replaced by the new and the message

```
Done
```

is displayed at the foot of the screen.

Query replace

This is like a global replace, but instead of replacing every occurrence, it stops at each one and asks whether you want to replace it or move to the next occurrence.

1. Follow steps 1 through 3 above.

The cursor will move to the first occurrence of the text you want to replace.

4.Type in one of these options:

<Space>	yes, do this replace
<Delete>	no, don't do this replace

9

Using electronic mail

9 *Using electronic mail*

Every UNIX system allows you to exchange electronic mail with other users on your local system. Most systems also allow you to exchange mail with users on remote systems.

Electronic mail is very quick. Messages will reach users on the same system almost immediately; and users on systems elsewhere in this country, or other countries, in a few minutes or hours.

About the `mail` program

The `mail` program described here is known as `mail` (or `ucb/mail`) and is available on many systems. Some systems provide a similar mail program called **`mailx`** which is an improved version of `mail`.

If necessary, use the command **`mailx`** instead of **`mail`**.

Other mail programs

There are many other mail programs available for UNIX systems. Check with your system administrator to see if any of them are available on your system. These will be more modern and one of them willprobably be the mailer of choice at your site.

Mail program modes

When you are using the `mail` program you are in one or the other of its modes.

If you are using the `mail` to read your mail, it is in *command mode*. There are a number of commands that you can use in this mode.

If you are using `mail` to write and send a mail message, it is in *`send mode`*. There are a number of commands that you can use in this mode.

Mail program files

The `mail` program uses several files. They are:

`/usr/spool/mail/Mail.rc` The global start-up file from which the `mail` program gets its default values.

`~ username/.mailrc` A personal start-up file in your home directory. You have to create this file and its contents.

`~ username/mbox` The file in your home directory where your mail is stored once it has been read.

`~ username/dead.letter` The file in your home directory where any mail message that you decide not to send is stored.

Addressing mail

You have to provide the mail system with an address for it to deliver the mail to. The syntax of any mail address is either:

`username`

or:

`username@system_name`

Network domains and mail addresses

Large networks may have thousands of systems connected to them. A domain name addressing scheme is used to specify system addresses in a way that humans can understand.

The top level domain for the United Kingdom is `uk`. This domain has several sub-domains; `ac` (for academic) and `co` (for commercial).

The `ac` domain has many sub-domains, based on the name of the organisations within it. The sub-domain for the University of Hull for example is `hull`.

If the organisation has a large number of systems, then the organisational domain might also contain sub-domains, each of which contains a number of systems. This is often done on a departmental or service basis.

The mail address for any system can be given as a series of domains, each of which is separated by a `.` (dot or full stop).

For example, the full mail address to a user on a particular system might look like this:

```
john@ubik.ucs.ed.ac.uk
```

Reading from left to right, this is the mail address for the user known as `john` on the system called `ubik`. This system is in the organisational sub-domain `ucs`, (University Computing Service) which is in the organisational domain `ed` (Edinburgh).

This organisation is located in the `ac` (academic domain) for the national domain `uk` (United Kingdom).

Mail addresses on JANET

Mail addresses on **JANET** use what is known as "big endian" notation, with the top level domain given first. The rest of the world uses "little endian" notation, with the bottom domain given first. For example:

JANET:	`joe@uk.ac.ed.cs.ubik`
Others:	`joe@ubik.cs.ed.ac.uk`

Although many people in the UK academic community still give their addresses in `JANET` notation, it is better to give it in the notation used (and understood) by the rest of the world.

Your mail program might be configured to handle mail addresses given in either notation. Check with your system administrator for further information.

Finding a person's mail address

Electronic mail programs do not provide you with the equivalent to a telephone directory: there is no way that you can look up a person's electronic mail address[1]. However, many people now include their email address on their business stationery. And of course you can always ask them for it.

The "header" of any mail that you receive will contain the email address of the person who sent it, in the `From:` field. For example:

```
Date:       Tue, 18 Aug 92 12:15:17 WET DST
From:       tony@com.cayman
Subject:    Xgator troubles
To:         john@uk.ac.ed.castle
```

Each domain has a "postmaster" who will try to supply you with a mail address if asked. For example:

```
From:       john@uk.ac.ed
Subject:    Address please
To:         postmaster@durham

Please provide the email address for Dr. J.R. Smith
who is a member of staff in the Psychology Dept.
Thanks
```

Using an alias

Using an alias as a mail address allows you to send a message to several people using only a single address.

To create an alias, define it in the `.mailrc` file in your home directory. For example:

```
alias unixpeeps keith@castle rob@castle paul@castle\
tony@castle
```

1. This is not strictly true. The system you are using might have a locally written program that allows you to search for a name in a local mail directory.

This creates a single alias for the addresses of four users. Using the alias `unixpeeps` as the mail address will send the message to each one of them. You can also use an alias within an alias, and so on!

Creating a distribution list

A distribution list is very useful where a group of users need to communicate amongst themselves on a regular basis.

Each list is given a name just like any other user. By addressing a message to the list name, the sender can be sure that everyone will receive a copy.

You cannot set up a distribution list yourself. Ask your system administrator to set one up for you.

Sending mail: the basics

Before sending mail for the first time, make sure that you understand how to give a mail address.

1. Enter the command:

   `mail` *`username@address`*

   You will then be prompted to enter the subject of your mail message.

2. Enter the subject title and press `<Return>`.

   `Subject:` *`subject_title`*

   If you make a mistake use the `<Delete>` key to backup and correct it.

3. Enter the text of your message. Use the `<Delete>` key to backup and correct any typing mistakes. Press the `<Return>`key whenever you want to start a new line.

4. To send the mail message press `<Return>` to start a new line and type a full stop. Press the `<Return>` key again.

Your message will be sent and the shell prompt will be displayed.

Example of sending mail

1. To compose and send a mail message to another user:

```
$ mail admin <Return>
Subject: travel arrangements <Return>
please book me a seat on the 8.00 a.m.  BA shuttle to
Heathrow. <Return>
dave <Return>
.<Return>
$
```

The text that the user types is shown in **bold**. `<Return>`is shown wherever the users presses the `Return` key.

Commands available in send mode

The following commands can be used while you are preparing a mail message in send mode.

To ...	Do this
cancel the mail message	~x
preview the mail message	~p
add users to the Cc: list	~c *users*
add users to the Bcc: list	~b *users*
display this list of commands	~?
use the visual editor on your mail message	~v
insert a file into the mail message	~r *filename*
add your signature	~a or ~A
add users to the To: list	~t *users*
run a shell command	~! *command*
include output from a shell command	~ <! *command*

Cancelling a mail message

You can cancel sending a mail message using the commands ~**x** or **~q**.

The command `~x` cancels the message you are sending and exits `mail`. It does not save the message you were composing.

The command `~q` cancels the message, saves it in the file `dead.letter` in your home directory, and then exits `mail`.

The `dead.letter` file only keeps the most recently cancelled mail message. The next time you cancel a mail message, the contents of the dead.letter file are overwritten.

Previewing a mail message

To preview the mail message up to the point you are currently at, use the command **~p**. For example:

```
To get a demo copy of this do
an ftp to the address above
~p
-------
Message contains:
To: helper
Subject: MetaCard
I've just discovered this application. It's great!.
It's available from MetaCard  Corporation (ftp.metac-
ard.com) To get a demo copy of this do an ftp to the
address above
(continue)
```

Sending a file in the mail

You can send the contents of a file as a mail message. For example:

```
mail tony jane peter < memo
```

This command mails the contents of the file `memo` to three users on the same system as the sender; `tony`, `jane` and `peter`.

Sending binary files through the mail

You can mail binary files between UNIX systems that have the **tarmail** utility installed. Use the **which** command to find out if it is available on your system. For more information see "*Locating a command"* on page 5-15.

The tarmail command converts the binary file into a text file and then compresses it to reduce its size. It then mails the file to the address given. For example:

```
tarmail joe@glasgow.psy "prog i promised u" a.out
```

This sends the file a.out to the person who's mail address is joe@glasgow. The quoted text is used as information for the Subject: field.

tarmail can be used to send several files in the same mail message. For example:

```
tarmail bill@cis "that prog" prog.c a.out
```

This "tars up" the files prog.c and a.out into the single mail message.

Once the mail message containing the file reaches its destination, the person can save the mail message to a file and then decode the file using the **untarmail** command. For example:

```
untarmail fromjohn
```

▼

If the tarmail command is not available on your system, you can send binary files through the mail by converting them into text files using the **uuencode** utility. For more information see "*Encoding files"* on page 12-6.

Preparing your mail using an editor

You can customise the `mail` program so that you can use your usual text editor within the `mail` program to create and edit your mail message. To do this:

1. Enter the `mail` command in the usual way.
2. Enter the subject of the mail message.
3. Call up the editor by entering the command

   ~v

   at the start of line.

   A file is automatically created in the directory `/tmp` to hold the contents of your message until it is sent.
4. Use your editor in the usual way to compose the mail message.
5. Exit the editor, saving the file in the usual way.
6. Send the message by entering a full stop at the start of a new line and pressing the `<Return>` key.

Customising mail *to use a specific editor*

To use an editor to edit your mail messages, set the variable `VISUAL` to the pathname of the editor that you want to use. By default this is set to the `vi` editor. So, for example:

```
set VISUAL=/usr/local/GNU/bin/epoch
```

This sets the `VISUAL` variable so that you can use the **epoch** editor, an X windows based version of the popular GNU emacs editor. See "*Customising the mail program*" on page 23 of this section.

Inserting files in mail

There are two ways to insert a file into a mail message as you are composing it: using a mail command or by using a an editor.

Using a mail *command*

To insert a file into a mail message use the `mail` command:

```
~r filename
```

Example of inserting a file into mail

1. To insert a file into a mail message:

```
here is my first go at that README
file that I agreed
to draft
~r project/README
"project/README" 136/4637
```

This copies the file `README` in the subdirectory `project` into the mail message at this point.

The message

```
filename: No such file or directory
```

is displayed if the file is not found.

Using an editor

1. Customise the `mail` program to use the editor of your choice. See "*Setting the visual editor variable*" on page 23 of this section.

2. Use the editor's commands to insert the file at the point in the mail message where you want to include it.

Both the vi and EMACS editors provide this facility. For more information see "*Insert the contents of another file*" on page 8-8.

Sending Cc: and Bcc: copies

To send a carbon copy of the mail message to other users use the command:

```
~c address1, address2, ...
```

Make sure that each address is followed by a , (comma).

To send a blind carbon copy of the mail message to another user, use the command:

```
~b address
```

Examples of sending Cc: and Bcc: copies

1. To send a carbon copy to another user:

```
the system does its job admirably.
~c neil@@dali <Return>
You might also find that
```

When the mail is sent, a copy will also be sent to the user `neil` on the system `dali`. The recipient of the mail will be able to see that a copy has also been sent to this person.

2. To send a carbon copy to several other users:

```
take a look at MetaCard from the
MetaCard Corporation
~c db@@glas, fra@@durham, lor@@city
```

When the mail is sent, a copy will also be sent to each of the users whose addresses follow the `~c` command. Each address is followed by a , (comma).

3. To send a blind carbon copy to another user:

```
be treated in total confidence
~b peter
Regards
John Smith
```

When the mail is sent, a blind carbon copy will also be sent to the user `peter`. The recipient of the mail will not know that this copy has been sent.

Using UNIX shell commands

To use a UNIX command while you are preparing a mail message use the command:

```
~! command
```

To include the output from a UNIX command into the mail message at this point, use the command:

```
~ <! command
```

Examples of using UNIX shell commands

1. To use a UNIX command while you are preparing a mail message:

```
the application is in
~! which guide
/usr/local/bin/guide
!
the directory /usr/local/bin
```

This displays the output from the command which guide.

2. To include the output from a UNIX command in the mail message:

```
the IP address for my Mac is
~<! grep jazz /etc/hosts
" grep jazz /etc/hosts" 1/30
Alternatively you can call it as jazz.ucs
```

This inserts the output from the command `grep jazz /etc/hosts` into the mail message at this point. Notice that command is echoed to indicate that it has been carried out, but that the command output is not displayed.

Adding your signature

You can add a personalised signature to your mail by creating a signature file and then appending this to your mail message. For more information see "*Creating files with a text editor*" on page 6-9.

To add your signature to a mail message use the command:

```
~a
```

or

```
~A
```

▼

If you want to add a long (or fancy) signature, create a file named `.signature` containing the text for it in your home directory. You can then insert this file into the mail message. Some mail prgrammes and network news readers can be configured pick up this file and sign your messages with it automatically.

Receiving mail

As wellas sending mail, you will need to know how to go about reading the mail messages that you receive.

Notification of mail arriving

When you login, the message

```
You have mail
```

indicates that there is mail for you.

While you are logged on, use the biff utility to inform yourself that you have received new mail.

Using the biff *utility*

Check on the status of biff[1] by entering the command:

```
biff
```

A `y` indicates that biff is already active; an `n` that biff is disabled.

To activate biff put the line

```
biff y
```

in your `.login` file.

Reading mail messages

To read your mail enter the command:

```
mail
```

A summary of your messages, made up of mail headers is displayed. The "mail header" is the information contained in every mail message which is used by mail programs to provide the user with a summary of the origin and contents of each message.

At the bottom of the screen is the `mail` prompt `&` at which you enter the commands that allow you to read your mail.

Example of how mail is displayed

This is an example of how mail is displayed.

```
Mail version SMI 4.0 Thu Oct 11 12:59:09 PDT 1990 Type ? for help.
"/usr/spool/mail/fred": 5 messages 2 new 1 unread
 * 1 F.Bloggs@@ed      Wed Jun 1 14:22 108/4120 using mail
 2 Scott@@ucl.cs       Wed Jun 1 20:17 37/1648 RFI
 U 3 colin             Fri Jun 3 11:37 43/1372 usability testing
 >N 4 karen@@ed.music  Fri Jun 3 11:54 16/365 meeting
 N 5 jcarr@@ed.lfcs    Sun Jun 5 18:29 19/391 Re:Using mail
 &
```

1. "Biff was Heidi Stettner's dog, back when Heidi (and I) and Bill Joy were all grad students at U.C.Berkley and the early versions of BSD were being developed. Biff was popular among the residents of Evans Hall, and was known for barking at the mailman, hence the name of the command."

Eric Cooper

Each message has a number, followed by information about that message. At the beginning of each line is a status indicator:

>	the current message
`U`	message has not been read
`N`	this is a new message
`*`	message has been saved
(*blank*)	message has been read but not saved

Reading commands

Use one of the following commands to read your mail.

To ...	Command
display the current message	`<Return>`
display message number n	`n`
display message for a specific user	`p` *user*
display next (previous) page of headers	`z [-]`
redisplay the mail headers	`h`

Reading the top of each message

To read the first few lines of a message use the command:

```
t message_list
```

This displays the first five (5) lines of each message. You can get the `mail` program to display more or fewer lines by changing the the `topline` variable. See "*Customising the mail program"* on page 23 of this section.

Reading other mail files

The command

```
mail
```

reads the mail that is in your system mailbox. This is a file named after your username that is in the directory `/usr/spool/mail`.

There are other files where your mail can be kept and which you might want to read.

Reading mail in your mbox *file*

When you quit the `mail` program, any messages that you have not explicitly saved or deleted are automatically saved to the file `mbox` in your home directory.

To read the messages stored in this file, start the `mail` program with the command:

```
mail –f
```

Switching to other mailbox files

You can quit reading the current mailbox file and switch to reading another one with the command:

For example

```
& folder +usability
```

reads the mail messages in the file `usability` which is in the folder directory.

Reading mail stored in other files

You can save mail messages in files that are located anywhere in your personal file system. To read files stored in these files, start the `mail` program with the command:

```
mail –f filename
```

Exiting the mail program

You can stop reading your mail and exit the mail program whenever you want. There are two ways that you can exit mail; each has a slightly different effect.

Exit mail *using the* q *command*

Using the **q** command tidies up your mailbox:

- messages you have read are moved to file called mbox in your home directory.
- messages you have deleted are removed from your mailbox.
- any messages that have not been read or deleted are retained in your mailbox and marked as unread.
- messages that have been explicitly saved to a file other than mbox are deleted.

For example, this is the type of message you might see after you quit the `mail` program.

```
& q
Saved 1 message in /user9/eucs/erpl08/mbox
Held 6 messages in /usr/spool/mail/erpl08
```

The user `erpl08` had seven messages in their mailbox.

Before quiting they read one of these messages, which is saved to the file `mbox` in their home directory. The other messages are held in their system mailbox.

▼
To retain all your messages in your system mailbox, set the hold variable to `set hold`. See "*Customising the mail program*" on page 23 of this section.

Exit mail *using the* x *command*

Using the **x** command leaves your mailbox untouched:

- messages that you have read will still be marked as unread.

- messages marked for deletion will not be deleted.

Managing mail

If you intend to use `mail` regularly then you will need to become familiar with some of the facilities which `mail` provides for saving, deleting and replying to mail messages.

Saving mail messages

To save a message (or messages) to a file enter the command:

```
s message_list filename
```

Organise your mail by saving it in files according to its subject or sender.

▼

Mail files are simply text files, with a format that the `mail` program understands. You can process them with any of the standard UNIX utilities like `grep` and read them with editors or pagers.

Examples of saving mail messages

1. To save the current mail message to a file:

```
&s junk
```

This saves the current mail message to the file `junk` in the current directory.

2. To save several mail messages to the same file:

```
&s 5 8 14 project/feedback
```

This saves messages 5, 8 and 14 to the file `feedback`. This file is in the the subdirectory `project` of the directory from which the `mail` program is being run.

3. To save all the messages from a particular user:

```
&s cayman xgator
```

This saves all the messages with the string `cayman` in the `From:` field of their message header to the file `xgator`.

Saving mail in folders

You can customise the `mail` program so that all your mail files are automatically held in one directory. These files are known as folders. See "*Customising the mail program"* on page 23 of this section.

To do this:

1. Create a directory to keep your mail folders in.
2. Set the value of the mail variable `folder` to the pathname for this directory.

To save a file to a folder in this directory, save the mail as usual, and append a + (plus sign) to the name of the file (folder) that the mail is to be stored in.

Examples of saving mail to folders

1. To save the current mail message to a folder:

```
& s +usability
"/usr/eucs/john/mrtidy/usability" [Appended]
```

This saves the current message to the folder `usability`, which is in the directory defined by the `set folder` variable. The message will be appended to the existing file. If the file does not exist, it is created.

2. To save several messages to the same file:

```
& s bob +frombob
 "/usr/eucs/john/mrtidy/frombob" [New file]
```

This saves all the messages from the user `bob` to the the folder `frombob`, which is in the directory defined by the `set folder` variable. In this example the folder does not already exist, so it is created.

Deleting mail messages

To delete a message (or messages) from the mail folder that you are currently reading, enter the command:

```
d message_list
```

Messages are not deleted until you quit using the `mail` program. If you exit the `mail` program they are not deleted.

Examples of deleting mail messages

1. To delete the current mail message:

```
& d
```

This deletes the current mail message.

2. To delete a range of messages:

```
& d 7-12
```

This deletes messages seven to twelve.

3. To delete all the messages that you have read:

```
& d :r
```

This deletes all the messages that you have read since you began this session with the `mail` program.

Undeleting messages

The delete command only marks messages for deletion. Messages are only deleted once you leave the `mail` program, or switch to reading mail from another mail file. Until then you can cancel any delete command with the undelete command.

The format of the command is:

```
u message_list
```

For example, to cancel a delete command:

```
& u 7
```

This cancels a previous command to delete message seven.

Replying to mail messages

There are two ways in which you can reply to a mail message: by sending a reply to the sender of the message or by sending a reply to every person that received it. Use whichever is most appropriate.

Reply to sender only

To reply only to the person who originally sent you the mail message, use the command:

```
R message_list
```

You would use this command where the mail message had been carbon copied to several other people, but you only wanted the original sender to see your reply.

Reply to all recipients

To send a reply to all the people who have been sent a copy of this message, use the command:

```
r message_list
```

You would use this command where the mail message had been carbon copied to several other people, and you wanted everyone to see your reply to it.

Examples of replying to mail messages

1. To reply to the sender of the current message:

```
& R
To: bob
Subject: Re: Travel arrangements
```

This creates a reply to the sender of the current message. The content of the `Subject:` field is taken from the existing message.

The user can now continue to send the mail message containing their reply.

2. To reply to all the original recipients of a particular mail message:

```
& r 4
To: scott tony adam jane
Subject: Re: XGator
```

This creates a reply to all the original recipients of message number 4 in the mail file that is presently being read. The content of the `Subject:` field is taken from the existing message. The user can now continue to send the mail message containing their reply.

Customising the mail program

The `mail` program can be customised by setting variables in the `.mailrc` file in your home directory.

Variables are set using the `set` command. Some variables have values assigned to them with the = (equals-sign).

Displaying preset variables

Your system administrator may have customised the `mail` program for you. The system wide default values are in the file `/usr/lib/Mail.rc`. Check which variables are already set before making changes of your own.

If you are in command mode, enter the command:

```
& set
```

If you are composing a mail message, enter the command:

```
~:set
```

This displays a list of variables that are currently set, and their values.

Setting the visual editor variable

To use an editor to edit your mail message, set the variable `VISUAL` to the pathname of the editor that you want to use. For example:

```
set VISUAL=/usr/ucb/vi
```

This sets the `VISUAL` variable to the `vi` editor.

Creating a signature

To create a signature set the variables `sign` and `Sign` in the `.mailrc` file. For example:

```
set sign='Lost in hyperspace'
```

This creates the signature *`Lost in hyperspace`* that the user can add to the end of their mail messages.

```
set Sign='Fax: +44 31 453 9732'
```

This creates the signature *Fax: +44 31 453 9732* that the user can add to the end of their mail messages.

Useful mail variables

For a complete list of mail variables, see the manual page for the `mail(1)` command. For more information see "*Viewing manual pages"* on page -20.

Variable	Default	Action
askcc	noaskcc	Prompts you for the names of the users you to send carbon copies to.
editheaders		Include the message headers when writing a mail message using an editor.
folder=*directory*		the directory to save mail files in.
hold	nohold	keep messages that have been read inthe system mailbox, not the mbox file.

Variable	Default	Action
`keepsave`	`nokeepsave`	do not delete messages that have been saved. Keep them in the system mailbox.
`PAGER`	`more`	the pager used to paginate output when a message is longer than the screen.
`sign=`*signature*		defines the signature inserted in the mail message by the ~a command.
`Sign=`*signature*		defines the signature inserted in the mail message by the ~A command.
`topline=`*number*	5	the number of lines of the top of each message toprint with the ~t command.
`VISUAL=`*editor*		the name of the editor called by the ~v command. editor is given as a full pathname.

Sample .mailrc file

Lines starting with a # are comments on the line(s) above them.

```
set askcc hold keepsave editheaders
# This line sets four variables. By default, these
variables are
# not set. Variables can also be set one to a line.
set folder=mrtidy
# Sets the directory for files that hold the mail mes-
sages that
# you save. This directory is in the user’s home
directory.
set PAGER=less
# Defines the pager to be used to view mail messages
that are
# longer than a single screenful.
set VISUAL=/usr/local/bin/ue
# Sets the editor that is called when you enter the
command ~v
# while composing a mail message.
sign="The Boy in the Bubble"
# The "signature that is inserted in your mail message
with the
# ~a command.
```

10

Managing jobs and processes

10 Managing jobs and processes

When you enter a command it invokes a program. While this program is running it is called a *process*. Although there is only one copy of a program, any number of processes can be started which run this program.

You can run several processes at the same time, such as an editor and a mail program and change from one to the other. A process, such as printing a file, can be placed in the background and run as a *job*, leaving you to continue working, without having to wait for the process to finish.

There are several commands which allow you to manage the processes that belong to you. You cannot do anything with processes belonging to other users.

Monitoring processes

To monitor the state of your processes use the **`ps`** command.

```
ps [-option]
```

Used without options this produces a list of all the processes owned by you and associated with your terminal.

Options for this command vary with the type of UNIX system that you are using. Consult the man page for the `ps` command to check which command options are available on your UNIX.

It may be that you are running more than one terminal session on the system and/or that you have processes which have no controlling terminal associated with them. This is likely if you are working within a networked windowing environment such as X windows.

To display details about all the processes which belong to you use one of the following commands:

System V	BSD
ps -u *your_user_name*	ps -x

Examples of using the ps *command*

1. To display information about your processes that are currently running:

```
ps
PID TTY TIME COMMAND
13185 ttyAF/ABFd 0:00 rlogind
13187 ttyAF/ABFd 0:01 bash
13218 ttyAF/ABFd 0:00 ps
```

This user currently has three active processes; the process associated with the `ps` command itself (`13218`); their current shell (`13187`) and a process associated with the remote login (`13185`) which indicates that they have logged in to their account on this system from another system.

2. To display information about all your processes (System V):

```
ps –u joanne
PID TTY TIME COMMAND
13251 ttyAF/ABFd 0:00 ps
13187 ttyAF/ABFd 0:01 bash
18765 ? 0:08 xclock
3012 ? 0:00 bash
3015 ? 0:00 zmail.ne
3016 ? 4:03 zmail.bi
18465 ? 0:00 bash
18469 ? 0:34 xman
```

This user has several processes that are not associated with a controlling terminal. They can be identified by the `?` (question mark) under the `TTY` column.

3. To display information about the processes associated with another user (System V):

```
ps -u rob
PID TTY TIME COMMAND
1390 ttyAH/ABHb 0:18 bash
1403 ttyAH/ABHc 0:01 bash
1435 ttyAH/ABHY 0:01 bash
1471 ttyAH/ABHY 5:57 ream
2307 ttyAH/ABHb 0:29 xbiff
2315 ttyAH/ABHb 0:27 xbiff
```

The user rob is running three instances of the bash shell, a mail programme known as ream and two biff utilities.

Understanding process information

The information displayed by the ps command varies acccording to which command option(s) you use and the type of UNIX that you are using.

These are some of column headings displayed by the different versions of this command.

Process Information

Heading	Description
UID\USER	The user ID number (or the username) of the person owning the process.
PID	The process identity (PID) number for this process. As each process is begun it is assigned a process number.Each system has a maximum value for the PID number, when this is reached the PID numbering is started again. This number can be used to kill the process
PPID	The process identity (ID) number of the parent process: the process from within which this process was created.
SZ	The sizes of the process's image in main memory. This is figure is given in K bytes
TTY	The controlling terminal associated with this process. A ? indicates that there is no controlling process.
TIME	The amount of CPU time used by this process.
COMMAND	The name of the command being executed by this process.

Displaying information about other user's processes

To display information about processes associated with another user use one of the following commands:

System V	BSD
ps –u *user_name*	**ps –aux | grep** *user_name*

Running processes in the foreground

When you enter a command at the shell prompt, your shell forks a child process in which to execute the command.

The parent process (your shell) waits for the command to complete and for its child process to die. Until then you are unable to enter another command. Commands entered in this way are known as *foreground processes*.

Once the command completes and the child process dies, the parent process is reactivated, the shell prompt returns, and you can enter another command.

Shell built-in commands

All shells have a number of built-in commands which are executed in *the shell's own process*. When you enter a command, the shell checks to see if it is a built-in command, and if so it executes it. If it is not a buit-in the shell forks a new process in which to execute the command.

Some shells have more built-in commands than others. Check the manual page for your shell for details.

Cancelling a foreground process

To cancel the process that is currently running enter the key combination:

Ctrl–c

Hold down the key marked `Ctrl` and press the `C` key. This cancels the current foreground process completely: it no longer exists.

Example of cancelling a foreground process

1. To cancel the foreground process:

```
find / -name '*.ps' -print
...
...
^C
pwd
/usr/jeremy/book
find . -name '*.ps' -print
```

The user enters a command to find files with the extension `.ps` starting from the root directory!

Realising that this command will take a long time to run and is not appropriate, they cancel it, check their current directory and re-run the command from there.

Suspending a foreground process

To suspend a foreground process enter the key combination:

```
Ctrl-z
```

Hold down the key marked `Ctrl` and press the `Z` key. This stops the current foreground process.

If you are using the Bourne shell you will then have to kill the process. See "*Killing jobs and processes*" on page 10 of this section.

Other shells provide you with a facility to restart a process in the foreground or as a background job.

Example of suspending a foreground process

1. To suspend a foreground process:

```
find / -name *.c -print
find: cannot read /dev/diag
...
^Z
[1]+ Stopped find / -name *.c -print
ps
22926 p9 S 0:03 -Tcsh (Tcsh)
29857 p9 T 0:00 find / -name *.xdh -print
29859 p9 R 0:00 ps

mail tony < memo.txt
```

The user enters a command to find files with the extension `.xdh` starting at the root directory. Realising that this process will take a long time to run they stop it while they do something else.

They check to see that the process still exists: it does. They then send some mail to the user tony. Now, they can restart the process again in the foreground or run it as a background job

Restarting a suspended foreground process

To restart a foreground process that you have suspended use the **`fg`** command. This command is not available if you are using the Bourne shell. For example:

```
lpr -Pps23 bigfile.ps
...
...
^Z
Stopped
date
Thu Feb 18 16:11:05 GMT 1993
fg
lpr -Pps23 bigfile.ps
```

This starts a process in the foreground and suspends it. The user then checks the date and restarts the foreground process.

Move a foreground process into the background

To run a foreground process as a background job:

1. Suspend the foreground process.
2. Enter the `bg` (background) command to move the process into the background. For example:

```
lpr -Pps23 part[1-8].ps
^Z
Stoppped
bg
[1] lpr -Pps23 interface.ps &
```

This starts a foreground process to print several files. The process is then stopped and the job moved into the background.

Running processes as background jobs

To run a process in the background as a job and carry on working add an `&` (ampersand) at the end of the command line. For example:

```
find /usr/man -name *.1 -print > sect1 &
[1] 2548
```

The shell forks a child process to run the command and displays the job number [`1`] and the PID (Process IDentifier) number, `2548`. The shell prompt returns and you can enter further commands.

Redirect the standard output for a command that is being run in the background to a file. This prevents the output from the command appearing on your screen and interrupting your current work. For more information see "*Redirecting standard output*" on page 5-12.

If the command is likely to produce error messages you will need to redirect standard error. Otherwise all error messages are sent to your screen.

Do not run an interactive command that requires you to type something at the terminal as a background job. If you do the job will stop and wait for user input. You will then have to bring it into the foreground and interact with it until it completes!

Managing background jobs

Most shells provide built-in commands to manage background jobs. For more information see "*Description of different types of shell"* on page 11-1 if you are unsure whether your shell provides this facility or not.

See the previous page for details of how to move foreground processes into the background.

Listing active jobs

To display the state of any background jobs that you are currently running use the **job** command. For example:

```
jobs
[1] + Running                   xterm -g 90x55
[2] - Running                   xterm -g 90x55
```

This displays the job number, its state and the command associated with this job.

To display the process number use the `-l` option. For example:

```
jobs -l
[1] + 3584Running               xterm -g 90x55
[2] - 3587 Running              xterm -g 90x55
```

Bring a job into the foreground

To bring a specific background job into the foreground:

1. Enter the **fg** (foreground) command followed by the job number.

fg *job_number*

If there is only one job running in the background just enter the **fg** command. For example:

```
lpr -Pps23 interface.ps &
jobs
[1] + Running                   xterm -g 90x55
[2] - Running                   xterm -g 90x55
[3]                             Running lpr -Ppsa
interface.ps
fg 3
lpr -Pps23 interface.ps
```

This starts a process to print a file as a background job. The user then checks the current active jobs and brings the print job -job number three (3)- into the foreground.

Killing jobs and processes

Each job or process that you start is usually completed within a few seconds. Sometimes a background job or a process without a controlling terminal hangs up and you will need to destroy this process by killing it.

To kill a process use the **kill** command:

kill [-signal] *process_identifier* (PID)

Most shells other than the Bourne shell have a built-in **kill** command. You can also use this built-in shell command to kill background jobs.

kill %*job_number*

See the man page for your shell for more information.

Use the **ps** command if you need to find the PID of a process. Each process is controlled by sending signals to it which tell it what to do. This is how the **kill** command has its effect; by sending a process a signal which tells it to stop, hang up or terminate etc.

You can list the different types of signal that can be sent to a process by using the **kill** command together with its `-l` option. For example:

```
kill -l
 1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL
 5) SIGTRAP 6) SIGIOT 7) SIGEMT 8) SIGFPE
 9) SIGKILL 10) SIGBUS 11) SIGSEGV 12) SIGSYS
13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGUSR1
17) SIGUSR2 18) SIGCHLD 19) SIGPWR 20) SIGWINCH
21) SIGURG 22) SIGIO 23) SIGSTOP 24) SIGTSTP
25) SIGCONT 26) SIGTTIN 27) SIGTTOU 28) SIGVTALRM
29) SIGPROF
```

Signal type 1 (one) will send a process a signal to hang up (SIGHUP). Signal type 9 (nine) is a signal to kill (SIGKILL) the process, this is certain to kill it.

Always try to kill a process with a simple kill command.

`kill` *process_identifier*

This is the cleanest way to kill a process and has the same effect as cancelling a process. If this does not work use the -1 signal:

`kill -1` *process_number*

This tells the process to hangup just as though you were logging out. The system will also attempt to kill any child processes.

As a last resort use the -9 signal:

`kill -9` *process_identifier*

This will stop the process dead in its tracks but it may leave any child processes still running

Examples of using the kill *command*

1. To kill a single process:

```
ps
 PID TT STAT TIME COMMAND
 9408 p9 S 0:00 ue temp2.xdh
 9450 pa S 0:01 -Tcsh (Tcsh)
 9501 pa T 0:00 less csh.1
 9503 pa R 0:00 ps
kill 9501
```

This kills process number 9501. Notice that the ps command which is entered to check on the process ID's has the latest process number.

2. To kill a process that refuses to die:

```
kill -9 9352
```

This makes a certain kill of process number 9352.

3. To kill a background job:

```
jobs
[1] + Running                  xterm -g 70x55
kill %1
[1] Done                       xterm -g 70x55
```

This kills job number `1` (one); the only job that is currently running. The Bourne shell does not support this feature.

4. To kill more than one process:

`kill 8939 9543`

This kills processes `8939` and `9543`.

11

The shell

11 *The shell*

Whenever you login to a UNIX system you are placed in a program called the shell. You can see its prompt at the bottom left of your screen. To get your work done, you enter commands at this prompt.

The shell acts as a command interpreter; it takes each command and passes it to the operating system kernel to be acted upon. It then displays the results of this operation on your screen.

Features provided by the shell

The shell provides you with one or more of the following features. You can:

- create an environment that meets your needs
- write shell scripts
- define command aliases
- manipulate the command history
- automatically complete the command line
- edit the command line

Some shells provide more of these facilites than others.

Description of different types of shell

There are several different shells available for UNIX: the most popular are described here,

You can use any one of these shells if they are available on your system. And you can switch between the different shells once you have found out if they are available.

Bourne shell

This is the original UNIX shell written by Steve Bourne of Bell Labs. It is available on all UNIX systems.

This shell does not have the interactive facilites provided by modern shells such as the C shell and Korn shell. You are advised to to use another shell which has these features.

The Bourne shell does provide an easy to use language with which you can write shell scripts.

C shell

This shell was written at the University of California, Berkley. It provides a C like language with which to write shell scripts; hence its name.

TC shell

This shell is available in the public domain. It provides all the features of the C shell together with EMACS style editing of the command line.

Korn shell

This shell was written by David Korn of Bell Labs. It is now provided as the standard shell on UNIX systems.

It provides all the features of the C and TC shells together with a shell programming language"%scrpt" similar to that of the original Bourne shell.

<u>B</u>ourne <u>A</u>gain <u>SH</u>ell

This is a public domain shell written by the Free Software Foundation under their GNU initiative. Ultimately it is intended to be a full implementation of the IEEE Posix Shell and Tools specification. This shell is widely used within the academic commnity.

bash provides all the interactive features of the C shell (csh) and the

Korn shell (ksh). Its programming language is compatible with the Bourne shell (sh).

If you use the Bourne shell (sh) for shell programming consider using bash as your complete shell environment.

Summary of shells

Feature	Bour ne	C	TC	Korn	BASH
`command history`		✓	✓	✓	✓
`command alias`		✓	✓	✓	✓
`shell scripts`	✓	✓	✓	✓	✓
`filename completion`		✓	✓	✓	✓
`command line editing`			✓	✓	✓
`job control`		✓	✓	✓	✓

Finding out which shell you are using

Information about which shell you are using is held in the SHELL environment variable. The command

```
echo $SHELL
```

displays the value of this variable.

You can identify which shell you are presently using from the last part of the pathname.

Pathname	Shell
/.../sh	Bourne shell
/.../csh	C shell
/.../tcsh	TC shell
/.../ksh	Korn shell
/.../bash	Bourne Again SHell

Changing your shell

First find out which shell you are using from the shells that are available on your system. See "*Finding out which shell you are using*" on page 3 of this section.

Switch to using another shell

You can switch to another shell for the remainder of your login session. To do this enter the shell command name at the system prompt. For example:

```
ksh
$
```

This switches you from your current shell to the Korn shell.

Changing to another login shell

To change the shell that you enter whenever you log in to the system use the **chsh** (change shell) command. Next time you login you will use the new shell.

You must reset any shell and environment variables that you set in your previous shell's startup file. Ifyou do not they will no longer be in effect. For more information see "*Managing your working environment*" on page 7-1.

Examples of changing the login shell

1. To change your login shell to the Korn shell (ksh):

```
chsh
Changing login shell for sarah.
Old shell name: /bin/csh
New shell: ksh
```

This changes the user's login shell from the C shell (csh) to the Korn shell (ksh).

If you mistype the name of the shell or specify a shell that is not available on your system:

- you will be given a list of all the shells that are available on your system, or
- a message similar to this will be displayed:

```
name is unacceptable as a new shell
```

Filename completion

You often enter the name of a file or directory as partof a command. You can complete the file or directoryname automatically by entering the the first few characters of its name and then pressing a particular key

This feature is available with all shells except the Bourne shell.

Filename completion with the C shell

To turn on filename completion set the shell variable `filec` by adding the line

```
set filec
```

to your C shell startup file.

To complete a file or directory name press the `ESC` key.

```
ls -l on<ESC>line_help
```

If the filename is not completed enter more characters until the filename can be uniquely identified.

Directories are shown by the trailing / character.

List possible names

To list all possible names of files which are not uniquely specified, press and hold down the CTRL key and type a D.

```
$ cat co<CTRL-D>
commands.xdh concepts.xdh conventions/ conver_man/
```

Directories are shown by the trailing / character.

Filename completion with the TC shell

To complete a file or directory name press the TAB key.

```
$ ls -l on<TAB>line_help/
```

If the filename is not completed enter more characters until the filename can be uniquely identified. Directories are shown by the trailing / character.

List possible names

To list all possible names of files which are not uniquely specified, type CTRL-D. (Press and hold down the CTRL key and type a D.

```
$ ls co<CTRL-D>
commands.xdh concepts.xdh conventions/ conver_man/
$ ls co
```

Directories are shown by the trailing / character.

Filename completion with the Korn shell

Filename completion is only available if the the environment variables EDITOR and/or VISUAL are set to either emacs, gmacs or vi. For example:

```
EDITOR=emacs
```

The value of the VISUAL variable takes precedence over the value of the EDITOR variable.

Filename completion with emacs *or* gmacs *set*

To complete a file or directory name with emacs or gmacs set press the ESC key twice.

```
$ ls -l on<ESC><ESC>line_help
```

If the filename is not completed enter more characters until the filename can be uniquely identified. Directories are shown by the trailing / character.

Filename completion with vi *set*

To complete a file or directory name with vi set press the \ (backward slash) key.

```
$ ls -l on<\>line_help
```

If the filename is not completed enter more characters until the filename can be uniquely identified. Directories are shown by the trailing / character.

List possible names

To list all the possible names of files whi ch are not uniquely specified, press the ESC key followed by the = key.

```
$ ls -l S<ESC><=>
1) SIRS
2) Solaris
```

Directories are shown by the trailing / character.

Filename completion with the BASH

To complete a file or directory name press the TAB key.

```
$ ls –l on<TAB>line_help
```

If the filename is not completed enter more characters until the filename can be uniquely identified. Directories are shown by the trailing / character.

List possible names

To list all possible names of files which are not uniquely specified, press the <Tab> key twice.

```
$ ls co<Tab><Tab>
commands.xdh concepts.xdh conventions/ conver_man/
$ ls co
```

Directories are shown by the trailing / character.

Editing the command line

You can edit the command line to make changes and correct mistakes before entering a command.

This feature is *not* available with the Bourne and C shells.

Command line editing with the TC shell

To edit the command line you can use the same commands as those provided by the EMACS editor. See the manual pagefor `tcsh(1)` for details of the full range of editing commands.

Using EMACS commands

You can type EMACS instructions in upper or lower case. There are two types of instruction, CONTROL and META.

CONTROL instructions

These are typed while holding down the CTRL key which is signified by the ^ character. For example ^F is generated by pressing the F key while holding down the CTRL key.

META instructions

These are typed after pressing the `ESC` key which is signified by `M-`. For example, `M-F` is generated by pressing the `ESC` key and then typing the letter `F`. Throughout this document, the `ESC` key is referred to as the `META` key.

Sometimes a `META` or `CONTROL` instruction will prompt you for more information. This information may be another `META` or `CONTROL` instruction or some text.

Moving along a line

To...	Do this...
Move cursor to next character	^F
Move cursor to previous character	^B
Move cursor to next word	M-F
Move cursor to previous word	M-B
Move cursor to start of line	^A
Move cursor to end of line	^E

Deleting text

To...	Do this..
Delete the character to the left of the cursor.	Press the DELETE key
Delete the character under the cursor.	^D
Delete from cursor to the end of a word.	M-D
Delete from cursor to the start of a word.	M-^H
Delete from cursor to the end of a line.	^K

Delete an entire line.	^A^K
Undelete (yank back) a word or line.	^Y

Command line editing with the Korn shell

The method for command line editing depends upon the value to which the environment variables `EDITOR` and\ or `VISUAL` are set. This can be `emacs`, `gmacs` or `vi`.

To use the editing commands provided by the `EMACS` editor you would add the following line to your `.profile` file.

```
EDITOR=emacs
```

The value of the `VISUAL` variable takes precedence over the the value of the `EDITOR` variable.

Command line editing can also be turned on by using a shell command to set an option to one of the above values. For example:

```
set -o vi
```

This command can be given on the command line or included in your `.profile` file.

If `emacs` or `gmacs` is set, command line editing is in the style of the `EMACS` editor; if `vi` is set editing is in the style of the `vi` editor.

Editing with EMACS style commands

To edit the command line you can use the same commands as those provided by the EMACS editor. For more information see "*Moving round the text"* on page 8-24.

Editing with vi style commands

To edit the command line you can use the same commands as those provided by the vi editor. For more information see "*Moving around the text"* on page 8-9.

Command line editing with the Bourne Again SHell

If you are using the BASH shell the command line is edited in exactly the same way as it is for the Korn shell. See above.

Using the command history

Each time that you enter a command it is added to a command history list. You can reuse commands from this list.

This command history facility is not available with the Bourne shell (sh).

Using command history with the C shell

To display the command history list enter the command

```
:history
```

This displays a numbered list of commands in the order in which you have used them.

Changing the size of the command history

By default, only the previous command is saved in the command history list. To save a larger number of commands set the history variable in your shell start-up file. For example:

```
set history=50
```

This will keep a history of the fifty (50) most recent commands that you have used.

Saving command history between logins

To save your command history between one login session and another set the variable `savehist` in your `.login` file. For example:

```
set savehist=25
```

This saves the previous twenty-five (25) commands you have used to the file `.history` in your home directory whenever you logout.

When you next login the contents of the file `.history` are placed in the history list. You can then re-use commands from your previous login session.

Too high a value for the `savehist` variable will slow down the login procedure. Do *not set the value* of the `savehist` variable higher than that of the `.history` variable.

Re-running a command from the history list

To run a command from the history list use a command from this table.

To...	Do this...
Run the previous command. 4.	!!
Run command number *n*	!*n*
Run most recent command starting with characters in *string*.	!*string*
Run most recent command containing characters that match *string*.	!?*string*

Examples of running a command from the history list

1. To re-run the previous command:

```
history
1 pwd
2 cp memo.txt /tmp
3 cd
4 history
!!
1 pwd
2 cp memo.txt /tmp
3 cd
4 history
5 history
```

The **`history`** command is used to list the command history. Then the previous command is re-executed, listing the command history again!

2. To re-run a specific command:

```
!3
cd
```

This re-runs command number three (3) in the command history.

3. To re-run the most recent command that starts with a particular character string:

```
!pw
pwd
/home/grace
```

This re-runs the command `pwd`.

4. To re-run the most recent command that contains a particular character string:

```
!?st
history
```

This re-runs the `history` command.

Previewing a command from the history list

You can preview a command from the history list without having to re-run it.

To...	Do this...
Preview the previous command.	!!:p
Preview command number *n*	!*n*:p
Preview most recent command starting with characters in *string*.	!*string*:p

Preview most recent command containing characters that match *string*.	!?*string*:p

Editing a previous command line

You can edit any previous command from the history list to create a new command. Suppose that command number 23 in the history list is:

```
23 lpr part[1-4].ps
```

To change [1-4] to [5-8] use the command:

```
!23:s/[1-4]/[5-8]/
```

This will now run the command:

```
lpr part[5-8].ps
```

Using command history with the TC shell

To display the command history list enter the command:

```
history
```

This displays a numbered list of commands in the order in which you have used them. It also gives the time and date that the command was used.

Changing the size of the command history

By default only the previous command is saved in the command history list.

To save a larger number of commands set the history variable in your shell start-up file. For example:

```
set history=50
```

This will keep a history of the fifty (50) most recent commands that you have used.

Saving command history between logins

To save your command history between one login session and another set the variable `savehist` in your `.login` file. For example:

```
set savehist=25
```

This saves the previous twenty-five (25) commands you have used to the file `.history` in your home directory whenever you logout.

When you next login the contents of the file `.history` are placed in the history list. You can then re-use commands from your previous login session.

Too high a value for the `savehist` variable will slow down the login procedure. Do not set the value of the savehist variable higher than that of the `history` variable.

Re-running a command from the history list

To run a command from the history list use a command from this table.

To...	Do this ...
Move up through the command history list one command at a time.	^P
Move down through the command history list one command at a time.	^N
Run the previous command.	!!
Run command number *n*.	\n
Run most recent command starting with characters in *string*.	!*string*
Run most recent command containing characters that match *string*.	!?*string*

Examples of runnning a command from the history list

1. To move up and down the command list one command at a time.

```
history
1 cd tmp
2 14:00 ls
3 14:01 cd SUMI
4 14:05 less README
5 14:12 cd
6 14:15 history
^P
history^P
cd^P
less^N
cd^N
history
```

Any previous command can be edited once it is displayed. To re-run the command press the<RETURN> key.

2. To re-run a specific command:

```
!3
cd
```

This re-runs command number three (3) in the commandhistory.

3. To re-run the most recent command that starts with a particular character string:

```
!pw
pwd
/home/grace
```

This re-runs the command `pwd`.

4. To re-run the most recent command that contains aparticular character string:

```
?st
history
```

This re-runs the `history` command.

Previewing a command from the history list

You can preview a command from the history list without having to re-run it. You can then edit the command line to modify the command.

To...	Do this...
Preview previous command in the list.	^P
Preview next command in the list.	^N
Preview command number *n*	!*n*:p
Preview most recent command starting with characters in *string*.	!*string*:p

Editing a previous command line

You can edit any previous command from the history list to create a new command. Suppose that command number 23 in the history list is:

```
23 lpr part[1-4].ps
```

Do a preview of the command if you are not sure that it is the right one.

To change `[1-4]` to `[5-8]` use the command:

```
23:s/[1-4]/[5-8]/
```

This will now run the command:

```
lpr part[5-8].ps
```

Another way to edit commands in the history list is to move up and down the list until you find the command you are looking for and then edit its command line. See "*Editing the command line*" on page 8 of this section.

Using command history with the Korn shell

To display the command history list enter the command:

```
history
```

This displays a list of commands numbered in the order in which you have used them.

Changing the size ofthe command history

By default the previous 128 commands are saved in the command history list.

Set the environment variable `HISTSIZE` to save a smaller or larger number of commands. For example:

```
HISTSIZE=50;export HISTSIZE
```

This will keep a history of the fifty (50) most recent commands that you have used.

Saving command history between logins

Whenever you logout or exit from the Korn shell the contents of your command history are saved in the file `.sh_history` in your home directory.

When you next login the contents of the file `.sh_history` are placed in the history list. You can then re-use commands from your previous login session.

Re-running a command from the history list

To run a command from the history list use a command from this table.

To...	Do this...
Run the previous command.	r
Run command number *n*.	r *n*
Run most recent command starting with characters in *string*.	r *string*

Examples of running a command from the history list

1. To run the previous command:

```
pwd
/u/user/eucs/joe
r
pwd
/u/user/eucs/joe
```

2. To re-run a command from the history list:

```
r 15
```

This re-runs command number fifteen from the history list.

3. To re-run the most recent command that starts with a particular string:

```
r grep
grep -n '[dD]on't' tasks | less
```

This re-runs the most recent command that starts with the string `grep`.

Previewing a command from the history list

You can preview a command from the history list without having to re-run it. Before you can do this the `EDITOR` and `VISUAL` variables must be set as they are for command line editing.

You can also list commands from the history list using the `fc` (fix command) command.

To...	Do this	
	Emacs	**vi**
Preview previous command in the list.	^P	M-k
Preview next command in the list	^N	M-j
Preview command starting with *string* (search forward from previous command).	^R^*string*	M-/^*string*
(search backward from command 0).		M-?*string*
Preview command containing string (search forward from previous command).	^R*string*	M-*string*
(search backward from command 0).		M-?*string*

Editing a previous command line

To edit a previous command line use the shell command `fc` (fix command) command.

`fc -e` *editor* `-lnr` *first* `last`

This will open the commands from the history list as specified in the range first to last. When you exit from the editor the command(s) will be run.

Setting the environment variable `FCEDIT` to the editor that you prefer will save you specifying it on the command line. For example:

```
FCEDIT=emacs; export FCEDIT
```

Examples of editing previous command lines

1. To list commands from the history list:

```
fc –l 27 34
```

This lists commands 27 through 34 from the history list. Using the `–n` option instead of `–l` prevents the history numbers from being listed.

2. To edit a command from the history list:

```
fc –e emacs 32
```

This will start the EMACS editor together with the contents of command number 32. You can then edit the command and exit the editor to run the command.

3. To edit several commands from the history list:

```
fc –e vi 27 35
```

This will start the `vi` editor and open a temporary file containing commands 27 through 35. You can then edit the commands and exit the editor to run the commands.

Using command history with the Bourne Again SHell

To display the command history list enter the command:

```
history
```

This displays a list of commands numbered in the order in which you have used them. By default only the previous command is saved in the command history list.

Changing the size of the command history

By default the previous 500 commands are saved in the command history list. Set the environment variable HISTSIZE. For example:

```
HISTSIZE=50;export HISTSIZE
```

This will keep a history of the fifty (50) most recent commands that you have used.

Saving command history between logins

Whenever you logout or exit from the Bourne Again SHell the contents of your command history are saved in the file `.bash_history` in your home directory.

When you next login the contents of this file are placed in the history list and you can re-use commands from your previous login session.

To specify how much of your command history is to be saved set the environment variable `HISTFILESIZE`. For example:

```
HISTFILESIZE=25;export HISTFILESIZE
```

This saves the previous twenty-five (25) commands you have used; not the complete command history list.

Previewing a command from the history list

You can search for and preview a command from the history list without having to re-run it.

You can also list commands from the history list using the `fc` (fix command) command.

To...	Do this (EMACS)
Preview previous command in the list.	^P
Preview next command in the list.	^N
Search backward up the command history.	^R*string*
Search forward down the command history.	^S*string*

Examples of previewing the command history

1. To move back up the list one command at a time:

```
1 ls
2 pwd
3 grep Smith address.txt > smiffs.tmp
4 sort -u post smiff.tmp
5 lp post
6 lpq
^P lpq
^P lp post
^P sort -u post smiff.tmp
```

This takes us back up to command 4. Entering further `CTRL-P` commands takes you further up the command history.

2. To to move down the command list carrying on from the example above:

```
^N lp post
^N lpq
```

This moves us back down to command 6. Entering further `CTRL-N` commands takes you down the command history.

3. To search up the command history:

```
^R
(reverse-i-search)'sort': sort -u post smiff.tmp
```

This searches up the command list for the first command containing the string `sort`. Repeating `CTRL-R` continues the search up the command history.

Once found you can edit the command line.

4. To search down the command list:

```
^S
(i-search)'grep':grep Smith address.txt > smifs.tmp
```

This searches down the command list for the first command containing the string. Repeating `CTRL-S` continues the search down the command history. Once found you can edit the command line.

Editing a previous command line

Preview the command history to find the command that you want to edit.

You can use either of two methods to edit previous command lines. Choose the method that you find best suits your needs.

Using the built-in editing commands

The BASH shell provides you with built-in command line editing that is identical to the EMACS editor. You can also switch to using a vi editing mode by entering the command:

```
M-^m
```

To switch from vi mode to EMACS mode use the command:

```
^e
```

Using the fc command

To edit a previous command line use the shell command `fc` (fix command) command.

`fc -e` *editor* `-lnr` *first last*

This will open the commands from the history list as specified in the range first to last. When you exit from the editor the command(s) will be run.

Setting the environment variable FCEDIT to the editor that you prefer will save you specifying it on the command line. For example:

```
FCEDIT=emacs; export FCEDIT
```

Command name completion

This feature is only available with the Bourne Again Shell (`bash`). It is not a feature of any other shell.

To complete a command name press the `TAB` key.

```
$ tel<TAB>net
telnet>
```

If the shell is unable to identify a unique command from the characters already entered it will ring the terminal bell.

Pressing the `TAB` key again displays all the commands on your path that begin with those letters. For example:

```
$ ma<TAB><TAB>
mach mail makecookie man maze
macptopbm make makedepend mapname
```

This produces a list of commands that start with the letters ma. Command name completion also works in the middle of a command line. For example the command name

```
$ teln<TAB> orion.geo.aber
```

expands to;

```
$ telnet orion.geo.aber
```

Creating a command alias

Remembering the name of a command and how it is used can be difficult. Assigning your own name for a command -an alias- is very easy.

Assigning an alias is done with the command:

```
alias name definition
```

The syntax of the alias command is different for eachshell. Follow the guidelines and examples for the shell that you are using.

▼
The `alias` caommand is not available in the Bourne Shell

Creating an alias with the C shell

To create a command alias use the command:

`alias` *name definition*

Adding a command alias

To make a command alias a permanent part of your working environment:

1. Edit the shell startup file `.tcshrc` and add the line defining the alias.
2. Save the changes and leave the editor.
3. Source the shell startup file to have the changes take immediate effect. For example:

```
source .cshrc
```

Examples of creating command alias in the C shell

1. To create a simple command alias:

```
alias del 'rm -i'
del memo.txt
rm: remove memo.txt? y
```

This creates the alias del for the command `rm -i` -which prompts you for confirmation that you want to remove a file before it does so.

Make this alias a permanent feature of your environment by adding it to your shell start up file `.cshrc`.

2. To create a command alias that consists of several commands:

```
alias what 'ps -aux | grep $USER | less'
```

This creates the alias **what**.

Entering the command **what** will now run the command

```
ps −aux | grep $USER | less
```

which consists of three commands linked together with pipes. For more information see "*Examples of connecting commands with pipelines"* on page 5-14.

Referring to other command aliases

You can use an existing command alias within anothercommand alias. Define a first alias:

```
alias print lpr
```

This creates the alias `print` for the **lpr** command used to print files to your default printer.

This alias can now be used within another alias. For example:

```
alias tree ‘cd; ls −R | print’
```

The alias tree will list the complete directory tree below the users home directory on the default printer. Multiple commands within an alias can be separated by a semicolon; just as they are on the command line.

Passing command arguments

To pass an argument to the alias at a specific point in the command use the one of the following notations:

`!^`-pass the first argument to the alias

`!*`-pass each argument given to the alias

Examples of passing arguments given to a command alias

1. To pass a single argument to a command alias:

```
alias print ‘lpr \!^ −Pps5’
```

```
print memo.txt
```

The notation `!^` causes the first argument to the command alias `print` to be inserted in the command at this point. The command that is carried out is:

```
lpr memo.txt -Pps5
```

Notice that the ! character is preceded by a \ to prevent it being interpreted by the shell as a history command.

2. To pass each argument to a command alias:

```
alias print 'lpr \!* -Pps5'
print part1.ps glossary.ps figure.ps
```

The notation `!*` causes each argument given to the alias `print` to be inserted in the command at this point. The command that is carried out is:

```
lpr part1.ps glossary.ps figure.ps -Pps5
```

Notice that the ! character is preceeded by a \ toprevent it being interpreted by the shell as a history command.

Displaying value of current aliases

To display the value of each alias known to the shell enter the command:

```
alias
```

To display the current value for a particular alias use the command:

`alias` *name_of_alias*

For example to display the current value of the alias named `print`:

```
alias print
lpr !* -Pps5
```

Informs you that `print` is an alias for the command `lpr !* -Pps5`.

Cancelling a command alias

To cancel a command alias during your current login session use the command:

```
unalias alias_name
```

For example:

```
unalias print
```

This cancels the command alias print for you current login session.

You will have to remove the alias definition from your shell startup file if you want to cancel the alias permanently.

Removing a command alias

To remove a command alias:

1. Edit the shell startup file and remove the line containing the alias definition.
2. Save the changes and leave the editor.
3. Source the shell startup file to have the changes takeimmediate effect. For example:

```
source .cshrc
```

Creating an alias with the TC shell

To create a command alias use the command:

```
alias name_of_alias definition_of_alias
```

Adding a command alias

To make a command alias a permanent part of your working environment:

1. Edit the shell startup file `.tcshrc` and add the line defining the alias.

2. Save the changes and leave the editor.

3. Source the shell startup file to have the changes take immediate effect. For example:

```
source .tcshrc
```

Examples of creating command alias in the TC shell

See "*Examples of creating command alias in the C shell*" on page 26 of this section.

Referring to ther command aliases

See "*Referring to other command aliases*" on page 27 of this section.

Passing command arguments

See "*Passing command arguments*" on page 27 of this section.

Displaying the value of aliases

See "*Displaying value of current aliases*" on page 28 of this section.

Cancelling a command alias

See "*Cancelling a command alias*" on page 29 of this section.

Removing a command alias

To remove a command alias:

1. Edit the shell startup file and remove the line containing the alias definition.

2. Save the changes and leave the editor.

3. Source the shell startup file to have the changes take immediate effect. For example:

```
source .tcshrc
```

Examples of creating command alias in the TC shell

See "*Examples of creating command alias in the C shell*" on page 26 of this section.

Creating an alias with the Korn shell

To create a command alias use the command:

alias *name_of_alias=definition_of_alias*

Adding an alias

To make a command alias a permanent part of your working environment:

1. Edit the shell start up file `.kshrc` and add the line which defines and exports the command alias. For example:

```
alias a3print='lp -d any -fa3';export a3print
```

2. Save the changes and leave the editor.

3. Source the shell startup file to have the changes take immediate effect. For example:

```
. .kshrc
```

Examples of creating command alias in the Korn shell

1. To create a simple alias:

```
alias del='rem -i'
del memo.txt
rm: remove memeo.txt? y
```

This creates the alias **del** for the command **rm -i** -which prompts you for confirmation that you want to remove a file before it does so.

Make this alias a permanent feature of your environment by adding it to your shell start up file `.kshrc`.

2. To create a command alias that consists of several commands:

```
alias what='ps -aux | grep $USER | less'
```

This creates the alias **what**.

Entering the command what will now run the command

```
ps -aux | grep $USER | less
```

which consists of three commands linked togetherwith pipes.

Referring to other command aliases

You can use an existing command alias within another command alias. Define a first alias:

```
alias print=lp
```

This creates the alias *print* for the **lp** command used to print files to your default printer.

This alias can now be used within another alias. For example:

```
alias tree='cd; ls -R | print'
```

The alias **tree** will list the complete directory tree below the users home directory on the default printer. Multiple commands within an alias can be separated by a semicolon; just as they are on the command line.

Displaying value of current aliases

See "*Displaying value of current aliases"* on page 28 of this section.

Cancelling a command alias

To cancel a command alias during your current login session use the command:

unalias *alias_name*

For example:

```
unalias print
```

This cancels the command alias print for you current login session.

If the alias name is the same as a command name escape the alias name to temporarily override the alias. For example:

```
alias rm='rm -i'
rm *.ps
toc.ps: ? n
.....
\rm
rm *.ps
```

You will have to remove the alias definition from your shell startup file if you want to cancel the alias permanently.

Removing a command alias

To remove a command alias:

1. Edit the shell startup file and remove the line containing the alias definition.
2. Save the changes and leave the editor.
3. Source the shell startup file to have the changes take immediate effect. For example:

```
..kshrc
```

Creating an alias with the Bourne Again SHell

For instructions on

- adding an alias
- referring to other command aliases
- displaying the value of aliases
- cancelling a command alias

See the appropriate heading in the previous section on the Korn shell.

Removing a command alias

To remove a command alias:

1. Edit the shell startup file and remove the line containing the alias definition.
2. Save the changes and leave the editor.
3. Source the shell startup file to have the changes take immediate effect. For example:

```
source .bashrc
```

12

File transfer utilities

12 File transfer utilities

UNIX provides utilities for transferring files and directories from one host[1] to another. Take the time to look at the next section on preparing files for transfer before going on to use the file transfer utilities.

Preparing files for transfer

Before transferring or copying files between hosts you may need to pre-process them. Utilities are available which convert several files into a single file, reduce the size of a file by compression, and convert binary files into ASCII encoded representation for sending by mail.

Creating a UNIX tape archive

The **`tar`** command can be used to turn several files into a single tarfile. **`tar`** can also be used to bundle up whole directories before transferring them to another system. Use **`tar`** whenever you want to transfer a number of files or directories between different UNIX systems.

Creating an archive file

To create an archive file use the command:

```
tar cvf tarfile_name.tar filenames (directory_name)
```

The command options `cvf` modify the way the **`tar`** command works.

`c` - create a new tarfile and write the named files to it

`v` - be verbose and display the name of each file as it is treated

`f` - use the next argument as the name of the tarfile

1. This section does not describe how to transfer files between a personal computer and a UNIX system. There are several ways in which this can be done. Ask your local system administrator for information.

By convention the name of a tarfile archive ends in `.tar`.

When you create a tarfile of a directory all the subdirectories are added; this can result in very large tarfiles. These should be compressed to reduce their size. See "*Compressing files*" on page 5 of this section.

When you are creating a tarfile, you should never specify a full pathname (i.e. one with first character /) for the input files, as **tar** will attempt to recreate the files in a directory with exactly the same name when you extract them. For more information see "*Pathnames: full and relative*" on page 2-8.

If you are extracting them on another system, this directory might not exist.

Examples of creating an archive file

1. To create a tarfile containing several files:

```
tar cvf data.tar *.dat
```

This creates a tarfile `data.tar` containing all the files in the current directory which have the filename extension `.dat`.

Look at the manual page for details of the command options cvf.

2. To create a tarfile of a directory:

```
tar cvf project.tar Project
Project/README
Project/Makefile
Project/Makefile
Project/main.c
Project/screen.c
Project/menus.c
Project/calculator.c
Project/setup/configuration
```

This creates a tarfile `project.tar` which contains the directory `Project` and its contents.

3. To create a tarfile containing several directories:

```
tar cvf book.tar text illustrations
```

This creates a tarfile `book.tar` containing the directories `text` and `illustrations` and all the files and subdirectories in them.

Making a tarfile smaller

Keep the size of the tarfile to a minimum by excluding files that can be recreated from files contained in the tarfile. Examples are:

- .o (object) files that can be recreated by compiling the program source code.
- .dvi files produced by processing TeX and LaTeX source files.
- .ps (postscript) files generated by drawing and publishing packages.

You can then reduce the size of tarfile by using the **compress** command. For example:

```
compress project.tar
```

This will create a file called `project.tar.Z`. You can unpack a compressed tarfile without uncompressing it (see below).

Examining the contents of a tarfile

To examine the contents of a tarfile use the **tar** command. For example:

```
tar tvf project.tar
```

This will display the contents of the tarfile `project.tar` without unpacking the tarfile. See the manual page for details of the command options `tvf`.

To display the contents of a compressed tarfile use the **zcat** command. For example:

```
zcat project.tar.Z | tar tvf -
```

This will display the contents of the compressed tarfile `project.tar.Z` without uncompressing and unpacking it.

Unpacking a tarfile

To unpack a tarfile use the **tar** command. For example:

```
tar xvf project.tar
```

This will unpack the contents of the tarfile `project.tar` into the current directory.

▼

Always move to an empty directory before unpacking a tarfile.

When you unpack a tarfile its contents are written into the current directory. Any file or directory with the same name as the contents of the tarfile will be overwritten!

Unpacking a compressed tarfile

To unpack a compressed tarfile use the **zcat** command. This leaves the compressed `.Z` tarfile intact. For example:

```
zcat project.tar.Z | tar xvf -
```

The **zcat** command recreates the uncompressed tarfile, which is then piped into the **tar** command to extract the files. Because the name of the tarfile is given as a – (hyphen), `tar` reads from standard input - the uncompressed tarfile!

You can also unpack one or more files without uncompressing the tarfile. This is useful if you want to extract one or more files from the tar archive but do not have enough diskspace to uncompress the complete tarfile.

For example:

```
zcat project.tar.Z | tar xvf - project/main.c
```

This extracts the file `main.c` from the tar archive `project.tar.Z`.

Compressing files

To reduce the size of file use the compress command. For example:

```
compress filename(s)
```

Each file is replaced by one with the same name and the extension `.Z`. The amount of compression will vary according to the type of file. Text files can be reduced by as much as 50 to 60%.

Use the `-v` option if you want to see the percentage by which each file is compressed.

Restoring compressed files

To restore files that have been compressed with the **compress** command use:

```
uncompress filename(s)
```

Uncompressing a file removes the compressed version of the file. To preserve the compressed `.Z` file and recreate the original use the **zcat** command shown above.

Examples of compressing files

1. To compress a single file:

```
ls -l sect5.fm
-rw------- 1 erpl08 67584 Sep 30 1992 sect5.fm
compress sect5.fm
ls -l *.Z
-rw------- 1 erpl08 33950 Sep 30 1992 sect5.fm.Z
```

The command `ls -l` shows the size of the file `sect5.fm` to be 67584 bytes before compression and 33950 bytes with compression -reducing the size of the file by 50%.

2. To compress several files:

`compress sect[1-8].fm`

This compresses files `sect1.fm` through `sect8.fm`.

3. To recover compressed files:

`uncompress sect[1-4].fm.Z`

This will decompress the compressed files `sect1.fm.Z` through `sect4.fm.Z`.

4. To recover a file and retain the compressed .Z file:

`zcat presentation.html.Z | less`

This recreates the uncompressed file `presentation.html` which is then piped to the **`less`** command to present the file for viewing. The output from the **`zcat`** command could also be redirected to a file:

`zcat bibliography.txt.Z > bib.txt; vi bib.txt`

This redirects the output from the **`zcat`** command to a file named `bib.txt` which is then displayed using the editor **`vi`**.

Encoding files

A binary file can be converted in to a text file using the **`uuencode`** command. For example:

`uuencode` *`source_file file_label`* `>` *`filename`*

The encoded file is generated in such a way that when decoded, it will have the same ownership and access permissions as the source file. Make sure that you set these to appropriate values before encoding the file and sending it to another user. The name of the file to be encoded is usually used for the *file_label*, which is used for the filename when the encoded file is decoded.

You can use the **tar** command to turn several files into a single file, which is then compressed before being encoded and mailed to another user.

▼

Since an encoded file is an ASCII file, you can edit it to change the access permissions and file label. The first line of an encoded file might be:

```
begin 600 pictures.tar.Z
```

The number 600 defines the access permissions for the original file - which the decoded file will also have. This is followed by the *file_label* to be used as the filename (pictures.tar.Z) for the decoded file.

To change the access permissions edit the number 600 to something more appropriate such as 644 and then save the file. You can change the file label in the same way. For example:

```
begin 644 from_mike.tar.Z
```

The decoded file will now have the name from_mike.tar.Z and everyone will be able to read it.

Examples of encoding files

1. To encode a single binary file:

   uuencode window.xwd window.xwd > window.uu

   This encodes the binary file window.xwd as the file window.uu. The file label is the same as name of the source file. The output from the command has to be explicitly redirected to the file window.uu.

2. To encode several files:

   tar cf - *.xwd | compress | uuencode pics.tar.Z pics.uu

First, this command creates a tarfile of all the files in the current directory with the filename extension `.xwd`. This tarfile is compressed and then encoded as the file `pictures.uu`.

When the user decodes the `.uu` file the compressed tarfile `pictures.tar.Z` will be created. This can then be unpacked to extract the `.xwd` files.

Decoding uuencoded files

An encoded file can be sent by mail to a user on another host. They can then save the file and restore the original binary file using the **`uudecode`** command:

```
uudecode filename
```

The recreated binary file is placed in a file with the name given as the file_label for the encoded file. It will have the same ownership and access permissions as the original file. For example:

```
uudecode window.uu
```

This decodes the file `window.uu` and recreates the file `window.xwd`. The encoded file is not removed.

Copying files between hosts

UNIX hosts are usually connected together by a network. You can copy files between your local host and other remote hosts with the **`rcp`** (remote copy) command.

```
rcp source_file destination_file
```

The *`source_file`* is the name of the file to be copied; the *`destination_file`* is the name of the file in which the copy is to be placed.

Prerequisites

Before using **`rcp`** you must:

- have an account on the UNIX host you want to copy files to and from
- have a special file called `.rhosts` in your login directory on the other UNIX host

The `.rhosts` file is a text file that you can create using any text editor.

Each line of the file has the name of a remote host followed by a space and your username on that host.

X Caution

There may be security measures or other authentication procedures in use at your site that prevent you from using the **`rcp`** command to copy files from one host to another.

The use of user authentication sytem known as Kerberos is now widespread. If it is in use at your site, this removes the need for you to use `.rhost` files. Check with your local system administrator to find out what the situation is on your system.

Example of a .rhosts *file*

These are the contents of a `.rhosts` file for the user "`helper`" on the host known as "`pollux`".

```
quinine helper
castle helper
festival helper
cancer helper
```

This allows the user "`helper`" to use the **`rcp`** (remote copy) command to copy files between their account on `pollux` and their accounts on the hosts `quinine`, `castle`, `festival` and `cancer`.

Copying files from another host

To copy a file from another host use the command:

```
rcp source_file destination_file
```

The source file is given as:

```
system_name:filename
```

When given as a relative pathname, the filename is interpreted relative to your home directory on the remote host. For more information see "*Pathnames: full and relative"* on page 2-8.

Examples of copying files from another host

1. To copy a file from your home directory on another host to your current directory on your local host:

   `rcp goya:notes part1.txt`

   This copies the file `notes` from the the users home directory on the remote host `goya` to the file `part1.txt` in their current directory.

2. To copy a file from a subdirectory in your account on a remote host to a subdirectory on your local host:

   `rcp ubik:gnu/rcs/README tmp/gnu_README`

   This copies the file `README` from the subdirectory `gnu/rcs` of the user's home directory on the remote host `goya` to the file `gnu_README` in the subdirectory `tmp` in their home directory on the local host.

3. To copy a file from a remote host on which you have an account under another username:

   `rcp joe@pollux:News/alt.hypertext tmp/hypertext`

This user has an account on the host `pollux` with the username `joe`. This copies the file `alt.hypertext` from the subdirectory `News` in joe's home directory on `pollux` to the file `hypertext` in the subdirectory `tmp` in their home directory on the local host

Copying files to another host

To copy a file to another host use the command:

```
rcp source_file destination_file
```

The destination file is given as:

```
system_name:filename
```

When given a relative pathname, the filename is interpreted relative to your home directory on the remote host.

Examples of copying files to another host

1. To copy a file from your current directory to your home directory on a remotehost:

   ```
   rcp parta pollux:part1
   ```

   This copies the file `parta` from your current directory to the file `part1` in your home directory on the remote host `pollux`.

2. To copy a file from a subdirectory on the local host to a subdirectory on a remote host:

   ```
   rcp Hyperhelp/helpex.hlp ubik:tmp/helpex/hlp
   ```

This copies the file helpex.hlp from your subdirectory Hyperhelp to the subdirectory FNGtmpFR in your home directory on the remote host "ubik".

Problems using rcp

Message	Meaning
`rcp: filename: No such file or directory`	The file that you are trying to copy does not exist
`Usage: rcp [-p] [-x] [-k realm] f1 f2`	You have failed to give the name of the source_file or the destination file.
`Permission denied`	The local host is not permitted to connect to the remote host. Either your .rhosts file on the remote host is incomplete or there are security/authentication procedures in use at your site that prevent you from using rcp in the normal way.

Copying files between two remote hosts

You can use the `rcp` command to copy files between two remote hosts from your local host. The command is:

```
rcp host_name:filename host_name:filename
```

If you have different usernames on any host you can still copy files between them using the format:

```
username@host_name:filename
```

Examples of copying files between remote hosts

1. To copy a file from one remote host to another:

   `rcp goya:tmp/vi.ref pollux:editors/vi.ref`

   This copies the file `vi.ref` from your home directory on the host `goya` to the subdirectory `editors` in your home directory on the other remote host `pollux`.

2. To copy a file between one remote host on which you have a different username, and another remote host on which you have the same username as on the local host (the one you are giving the command from):

   `rcp festival:README jamie@scorpio:README`

   This copies the file `README` from the home directory of your account on the remote host `festival` to the home directory of your account on the remote host `scorpio`. On this host you have the username `jamie`.

3. To copy a file between remote hosts, on each of which you have a different username to that on the local host (the host you are giving the command from):

   `rcp joe@ubik:News/funnies helper@pollux:new_jokes`

This copies the file `funnies` from the subdirectory `News` in your home directory on the host `ubik`. On this host you have the username of `joe`.

The copy is placed in the file `new_jokes` in your home directory on the host `pollux`. On this system you have the username of `helper`.

Transferring files

You can transfer files between hosts using the **`ftp`** command. This program is the standard means of transferring files between hosts on the Internet network.

Your system might also support the use of the **`hhcp`** (host to host file transfer) command. This is supplied as part of the Colour Book Software (Blue Book) that is used within the UK Academic Community.

Be sure to read section Xef here before transferring files for the first time.

Prerequisites

Before using the `ftp` program or the `hhcp` command to transfer files you will need an account on the remote host.

The *only* exception to this is where the remote host is an information server that allows you to transfer files using `ftp` or `hhcp` without having to supply a username and password as authentication.

▼

You can send a file usingelectronic mail. For more information see "*Sending a file in the mail"* on page 9-7.

Transferring files using the `ftp` program

To begin using the `ftp` program enter the command:

```
ftp
ftp>
```

The program starts and the `ftp>` prompt is displayed. You are now ready to give commands to the `ftp` program

About ftp *Commands*

To list the commands you can use with the `ftp` program:

1. Type a **?** (question mark) and press `<Return>`.

A list of commands is displayed. To get a short description of each `ftp` command:

1. Type **help** followed by the command you want to know more about and press the `<Return>` key. For example:

```
ftp> help open
open connect to remote tftp
ftp>
```

You will find a much fuller for each command in the UNIX programr's Manual pages for `ftp`.

Opening a connection to your remote host

Before you can transfer files you must open a connection to the `ftp` server on the remote host.

1. At the ftp prompt type **open** followed by the name of the host. Press the `<Return>` key.

You are then prompted for access authorisation.

2. Enter your username and password for the remote host.

Understanding host names

Each host has a unique address, which is given as a dot separated list of named domains. For example:

```
ftp.ucs.ed.ac.uk
```

This is the "domain name" of the host which acts as the anonymous ftp server (`ftp`) for the university computing services (`ucs`) at Edinburgh (`ed`). As a university it resides in the academic (`ac`) domain for the United Kingdom (`uk`).

You might not need to give the full address. Someone at another UK academic site only needs to use the `ftp.ucs.ed` part of the name. They can assume that their local host is in the same (academic) domain as this host.

Names can also be given as a series of dot (`.`) separated numbers. For example

```
129.148.31.66
```

can be used instead of

```
wilma.cs.brown.edu
(66) (31)(148) (129)
```

Each number is the identifier for a particular domain. However, unlike domain names, the left-most number is the most significant, and represents the top domain.

Examples of opening a connection to a remote host

1. To open a connection to the `ftp` server on a remote host:

```
ftp> open castor
Connected to castor.ucs.ed.ac.uk
220 castor FTP server (SunOS 4.1) ready.
Name (castor:erpl08): erpl08
331 Password required for erpl08.
Password: not echoed
230 User erpl08 logged in.
ftp>
```

This shows the user opening a connection to their account on the remote host `castor`. Notice that you must supply your username and password for your account on the remote host.

2. To open a connection to an anonymous ftp server on a remote host:

```
ftp> open info.cern.ch
Connected to nxoc01.cern.ch.
220 nxoc01 FTP server (Version 5.20 (NeXT 1.0) .....
Name (info.cern.ch:erpl08): anonymous
331 Guest login ok, send ident as password.
Password: enter your email address here
230 Guest login ok, access restrictions apply.
ftp>
```

This shows a user opening a connection to the anonymous ftp server `info.cern.ch`, which is located in CERN, the European Particle Physics Laboratory in Geneva, Switzerland.

This server is an archive for the software and documentation used by the World Wide Web

Sending files to a remote host

To send one file from the local host to the remote host you can use either the **send** or the **put** command.

To send multiple files to the the remote host use the **mput** command.

Files to be sent are assumed to be in your current directory unless you specify otherwise. Files will be placed in your current directory on the remote host unless you specify otherwise.

Examples of sending files with the ftp program

1. To send a single file from your current directory on the local host to the current directory on the remote host:

```
ftp> send afile
```

This copies the file `afile` from your current directory on the local host to your current directory on the remote host.

If you don't give a name for the remote-file the name of the local-file you are copying is used to name the remote-file.

2. To use file pathnames to define both the local-file and the remote-file:

```
ftp> put REPORTS/april reports/month4
```

This copies the contents of the file `april` in the sub-directory `REPORTS`, which is in your current directory on the local host to the file `month4` in the sub-directory `reports`, which is in your current directory on the remote host.

3. To send multiple files to a remote host:

```
ftp> mput help/*.xdh
mput index.xdh?
```

This prompts you to confirm for each file with the extension `.xdh` in the subdirectory `help` on your local host that it is to be copied to your current directory on the remote host.

Entering a **y** copies the file; an **n** cancels this.

Problems using the ftp program

Message	Meaning
filename: No such file or directory	The file you are tring to get or send does not exist. Most likely that you have the filename wrong.
Password: *your password* 530 Login incorrect. Login failed	Can be caused by any one of: - you have used the wrong username on the remote host - you have used the wrong (or an incorrect) password - you do not have an account on this host

Getting files from a remote host

To get a single file from the remote host to your local host use the **`get`** command. To get multiple files from the remote host to your local host use the **`mget`** command.

Files to be got are assumed to be in the current directory on the remote host unless you specify otherwise. Files will be placed in your current directory on the local host unless you specify otherwise.

▼

To read a text file on the remote host without transferring it, add a - (hyphen) to the end of the get command. For example:

```
get README -
```

This displays the file README without having to get it.

Examples of getting files with the `ftp` program

1. To get a single file from a remote host:

```
ftp> get news
```

This transfers a copy of the file `news` from the current directory on the remote host to your current directory on the local host.

If you don't give a local filename the filename of the remote file you are copyin is used to name the local file.

2. To get a binary file:

```
ftp> binary
200 Type set to 1
ftp> get mosaic_sun.tar.Z
```

This changes the file transer type to binary. The get command is then used to retrive the binary file `mosaic_sun.tar.Z`.

2. To get a file from a subdirectory on the remote host:

```
ftp> get reports/quarterly/march tmp/month3
```

This transfers a copy of the file `march` in the remote subdirectory `reports/quarterly` to the local filename `month3` in the subdirectory `tmp`.

3. To get several files from a remote host:

```
ftp> mget *.ps
```

```
mget PDIAL-By-Section/Intro.ps? n
```

This transfers a copy of each file with the fileneme extension `.ps` in the current directory on the remote host to the current directory on the local host.

You are prompted to confirm that you wish copy each file. Entering a **y** copies the file; pressing `<Return>` confirms the defailt **n** and cancels this copy.

Useful ftp *commands*

When you come to use `ftp` on a regular basis you will find the following commands useful. They allow you more control over the file transfer process.

Setting transfer type to binary

Many of the files that you transfer will be in a binary format, acompressed tarfile, for example. Before you can transfer this type of file you must change the file transfer type to binary:

```
ftp> binary
200 Type set to 1
ftp>
```

To reset the file transfer type for transfering text (ASCII) files:

```
ftp> ascii
200 Type set to A
ftp>
```

Making a directory on the remote host

Use the **mkdir** command to make a directory on the remote host. For example

```
ftp> mkdir reports
257 MKD command successful.
ftp>
```

creates the directory reports in your login directory on the remote host.

Changing your local directory

You can check for and then change your current directory on the local host using the **lcd** command. For example:

```
ftp> lcd
Local directory now /u/user/eucs/eucs99
ftp> lcd REPORTS
Local directory now /u/user/eucs/eucs99/REPORTS
```

Here, you have changed to your current directory and then moved to the subdirectory `REPORTS`.

Using UNIX commands on your local host

Entering an **!** (exclamation mark) at the `ftp` prompt starts an interactive shell and displays the shell prompt. You can then use any UNIX command before typing **exit** at the shell prompt and returning to the `ftp` prompt. For example:

```
ftp> !
% cal 7 1990
July 1990
S M Tu W Th F S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

```
29 30 31
% exit
exit
ftp>
```

Listing directories on the remote host

You can list the contents of a directory on the remote host. The command **ls** prints a short version; the command **dir** a longer version. You will need to give the path to the directory if it is not in your home directory on the remote host. For example:

```
ftp> ls
200 PORT command successful.
150 ASCII data connection for /bin/ls
(129.215.128.23,12707)
(0 bytes).
public
reports
26 ASCII Transfer complete.
17 bytes received in 0.26 seconds (0.064 Kbytes/s)
ftp>

ftp> dir reports/quarterly
200 PORT command successful.
50 ASCII data connection for /bin/ls
(129.215.128.23,12708)
(0 bytes).
total 10
rw-rw-rw- 1 eucs99 ug         9383 Jul 18 15:59 march
226 ASCII Transfer complete.
71 bytes received in 0.26 seconds (0.27 Kbytes/s)
ftp>
```

Changing directories on the remote host

You use the **cd** command to change from one directory to another on the remote host. A path to the directory can be given as an argument to the command. Entering the command **pwd** will display the path to your current directory. For example:

```
ftp> cd reports/quarterly
250 CWD command successful.

ftp> pwd
257 "/home/pisces/erpl08/reports/quarterly" is cur-
rent directory.
```

Using anonymous ftp

Some remote hosts are set up to allow you to open an `ftp` connection to them even though you don't have an account on them. That's right, anyone can login and see what's available!

The convention is to use the username `anonymous` and then give your email address in place of a password.

Once you are connected to this type of host you can use the usual `ftp` commands and get any file(s) from amongst those that you are allowed access to. See "*Useful ftp commands*" on page 20 of this section.This type of host is used to hold public archives of documents, data and software.

Example of using anonymous ftp

1. To connect to the anonymous ftp server at the Univerity of Edinburgh and list the files in the login directory:

```
ftp> open ftp.ed.ac.uk
Connected to pcserver2.ed.ac.uk.
======================================================
======
Welcome to the University of Edinburgh Anonymous FTP
server
======================================================
======
....
220-The current time is Thu Sep 30 10:02:46 1993.
220-
220 pcserver2 FTP server (Version 2.0WU(20) Fri Jun 25
13:34:24 BST 1993) ready.
```

```
Name (ftp.ed.ac.uk:erpl08): anonymous
331 Guest login ok, send your complete e-mail address
as password.
Password:enter full mail address
230-
....
230 Guest login ok, access restrictions apply.
ftp> dir
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 130
drwxr-xr-x 2 13 13 512 Sep 29 1992 .cap
-rw-r--r-- 1 root daemon 530 Mar 17 1993
.login_message
drwxr-xr-x 2 root daemon 512 Jun 25 12:53 bin
drwxr-xr-x 2 root daemon 512 Oct 6 1992 dev
drwxr-xr-x 3 root wheel 512 Aug 12 18:19 emwac
drwxr-xr-x 3 root daemon 512 Mar 17 1993 etc
drwxr-xr-x 2 1400 daemon 512 Jun 30 08:27 george
drwxr-xr-x 2 root daemon 512 Nov 12 1992 gnu
drwx-wx-wx 2 root daemon 512 Sep 29 15:06 incoming
-rw-r--r-- 2 root daemon 50874 Sep 29 23:00 index
-rw-r--r-- 2 root daemon 50874 Sep 29 23:00 ls-lR
-rw-r--r-- 1 root daemon 15086 Sep 29 23:00 ls-lR.Z
drwxr-xr-x 3 1407 daemon 1024 Nov 12 1992 maps
drwxr-xr-x 25 root daemon 2048 Aug 11 11:07 pub
drwxr-xr-x 4 1005 daemon 512 Jul 8 1992 uniras
drwxr-xr-x 2 root daemon 512 Aug 26 1992 unixhelp
drwxr-xr-x 4 root daemon 512 Jun 25 12:52 usr
226 Transfer complete.
1053 bytes received in 1.1 seconds (0.92 Kbytes/s)
ftp>
```

Transferring files with the hhcp command

The **hhcp** (host to host copy) command is widely used within the UK Academic Community. It is also used by some commercial organisations within Europe.

Although it is possible to use it over Local Area Networks, it is mostly used to transfer files from one site to another over a Wide Area Network such as JANET.

Getting information about hosts

If you already know the name of a host, you can use the command **hosts** to check if it appears in the NRS (Name Registration Scheme) and,if so, what services it supports.

Names for hosts on the JANET network have the form:

uk.ac.*name_of_organisation*.*name_of_host*

For example

hosts festival

gives the following information:

```
Standard Name : uk.ac.edinburgh.festival
Abbreviated Name             : uk.ac.ed.festival
Description                  : EUCS Central Unix
Service
Services:                    : X29/JANET FTP/JANET
MAIL/JANET
```

The information on `Services` shows that this host provides `FTP/JANET`, allowing you to transfer files between your account on your local host and your account on this remote host.

Host names for commercial hosts on the UK PSS (Packet Switch Stream) have the form:

uk.co.*name_of_organisation name_of_host*

For example:

```
hosts ixi
```

gives the following information:

```
Standard Name                    : uk.co.ixi-limited
Abbreviated Name                 : uk.co.x
Description                      : IXI limited
Services                         : MAIL/JANET MAIL/PSS
JANET MAIL Relay                 : uk.ac.uknet
PSS MAIL Relay                   : uk.ac.uknet
```

You can send electronic mail to IXI Ltd.; but their host(s) do not support file transfer.

Using the hhcp *command*

To transfer files use the command:

```
hhcp source_file destination_file
```

The *source_file* is the ifle that you want to transfer. It can be a file on your local host or on the remote host. The transfer can be in either direction.

The *destination_file* is the copy of the *source_file*. It too can be on your local host or a remote host, depending on where the source file is.

Getting access authorisation

For security reasons, the `hhcp` command will prompt you for access authorisation that is acceptable to the operating system on the remote host.

On a UNIX host this is usually in the form of a username and password. Other types of host may use a special file transfer password.

This is an example of how you are prompted for access authorisation by the `hhcp` command.

```
hhcp afile castle:bfile
[castle: no stored attributes]
transfer authorization:username
transfer password: password
afile: transfer id is nnnn
```

Storing transfer authorisation for a host

You can use the **hhstore** command to store access authorisation for your account on a remote host.

Whenever you transfer a file to or from this host the access authorisation that you have stored is automatically used by `hhcp` to authorise the file transfer. For example:

```
hhstore -a festival
transfer authorization: helper
transfer password: password
```

This stores authorisation for transfers to and from the user's account on the host `festival`, on which their registered username is `helper`. Note that the password is not displayed.

The `-a` option ensures that you are only prompted for your username and password.

Changing transfer authorisation for a host

You can also use the `hhstore` command to remove the stored authorisation for a host whenever you change your password. For example:

```
hhstore -r festival
```

This removes any access authorisation that you might have stored for the host `festival`. Use of the `-r` option indicates that you want to remove the current authorisation for a host.

Sending files to a remote host

To send a file to a remote host use the command:

hhcp *filename hostname*:*filename*

The destination file is defined by using the name of the remote host followed by the filename. A colon (`:`) is used between the host name and the filename. This filename can be either a full pathname, or a relative pathname from your home directory on the remote host.

To send binary files use the `-b` option.

▼

If the pathname part of the destination-file contains any characters which have special significance for UNIX they must be quoted

For example, the character > is a special character for UNIX, so the definition for the destination-file on a host whose operating system delimits file paths using this character might look like this:

```
hhcp myfile remote-host:"n>docs>mine>memo3"
```

Examples of sending files with the hhcp *command*

1. To send a single file to your home directory on the remote host:

   `hhcp myfile festival.ed:newfile`

   This copies the file `myfile` from your current directory on the local host to the file `newfile` in your home directory on the host with the name `festival.ed`.

2. To copy a file from one subdirectory to another:

   `hhcp tmp/guide.txt castle.ed:book/chapt1/part2`

   This copies the file `guide.txt` from the subdirectory `tmp` in your current directory, to the file `part2` in the subdirectory `book/`

`chapt1` which is in your home directory on the remote host with the name `castle.ed`.

3. To copy a binary file:

`hhcp -b afile walden.co:incoming/afile`

This copies the binary file `afile` from your current directory to a file with the same name in the subdirectory `incoming`, in your home directory on the host with the name `walden.co`.

Problems using the hhcp command

Message	Meanng
`No remote filename`	You have forgotten to give the filename for the file thatis to be sent to or copied from the remote host.
`New mail(nnn), From: root. On: X25 Xfer status of .`	You have received an email message to tell you that the transfer has failed. Read the mail message to find out the reason for this.

13

Text file manipulation utilities

13 Text file manipulation utilities

UNIX provides you with several utilities for working with text files. These utilities can be combined with each other to provide more complex general purpose tools.

Comparing files

You can display the line by line difference between two files with the **diff** command.

```
diff file1 file2
```

The information given by the command tells you what changes need to be made for *file1* and *file2* to match. If there is no difference between the files then no output is produced.

diff indicates which lines need be added (a), deleted (d) or changed (c).

Lines in *file1* are identified with a (<) symbol: lines in file2 with a (>) symbol

Examples of using the diff *command*

1. To compare the contents of two files:

```
diff email addresses
2a3,4
>Jean JRS@pollux.ucs.co
>Jim jim@frolix8
```

This displays a line by line difference between the file email and addresses.

To make these files match you need to add (`a`) lines 3 and 4 (`3,4`) of the file `addresses` (`>`) after line `2` in the file `email`.

Here are the contents of files `email` and `addresses` used in this example. Line numbers are shown at the beginning of each line to help you follow this example.

email	addresses
1 John erpl08@ed	1 John erpl08@ed
2 Joe CZT@cern.ch	2 Joe CZT@cern.ch
3 Kim ks@x.co	3 Jean JRS@pollux.ucs.co
4 Keith keith@festival	4 Jim jim@frolix8
	5 Kim ks@x.co
	6 Keith keith@festival

2. To compare two files ignoring differences in the case of the letters and blank spaces:

```
diff -iw part1 part_one
```

Lines in the two files that differ only in case and by the number of spaces they contain are considered to be identical.

Comparing directories

You can use the `diff` command to compare the contents of two directories:

```
diff directory1 directory2
```

The information given by the command tells you what to do to make the directories match. If there is no difference between the directories then no output is produced.

`diff` reports if a file or directory is only to be found in *directory1* or *directory2*. It also reports any line by line differences for files that are common to each directory.

Example of comparing directories with the diff *command*

1. To compare two directories in the current directory:

```
diff usability Usability
Only in Usability: schedule
diff usability/email Usability/email
2a3,4
>Jean JRS@pollux.ucs.co
>Jim jim@frolix8
```

`diff` reports that the file (or directory) `schedule` is unique to the directory `Usability`.

`diff` then reports that there are differences between the file `email` in the directory `usability` and the file of the same name in the directory `Usability`.

To make these files match you need to add (`a`) lines 3 and 4 (`3,4`) of the file `Usability/email` after line `2` in the file `usability/email`.

Contrasting two text files

To display a clear visual contrast between two text files use the command:

```
sdiff file1 file2
```

`sdiff` displays each line of the two files side by side, making it easier for you to see the difference between them

Lines that are different are shown with a | symbol. Lines unique to *`file1`* are identified by a < symbol: lines uniq to *`file2`* with a >. A blank line indicates that the lines are identical.

Example of contrasting files with the sdiff *command*

1. To contrast two files:

```
sdiff -w 80 email addresses
Mike erpl08@ed                          | John erpl08@ed
Joe CZT@cern.ch                           Joe CZT@cern.ch
                                          >Jean JRS@pollux.ucs.co
                                          >Jim jim@frolix8
Kim ks@x.co                               Kim ks@x.co
Sam s.wally@aston                         <
Keith keith@festival                      Keith keith@f
```

This contrasts the two files `email` on the left and `addresses` on the right.

The option `-w 80` is used to set the width of the output from the command to 80 characters -the most that will fit across most computer terminals. The default is `130` characters - a left-over from the days when output was to a line printer not a terminal.

The first line in each file is different and is marked with a `|`. Lines 3 and 4 of the file `addresses` are unique as is line 6 of the file `email`. All the other lines are identical.

Removing duplicate lines with uniq

To remove duplicate lines from a file use the command:

```
uniq in_file out_file
```

`uniq` searches the *`in_file`* for consecutive duplicate lines -a line that is immediately followed by one or more lines that are identical to it- and removes them. The result is an *`out_file`* that contains no duplicate lines.

Use the `sort` command if you want to remove duplicate lines that occur in different parts of a file.

Examples of using the uniq *command*

1. To remove consecutive duplicate lines from a file:

`uniq acc_nos accounts`

This removes all consecutive duplicate lines from the file `acc_nos` and places the output in the file `accounts`.

2. To list the consecutive duplicate lines and the number of times that each occurs:

`uniq -c tel_nos errors`

This lists each of the consecutive duplicate lines in the file `tel_nos` and the number of times that each is repeated. The results are placed in the file `errors`.

Searching the contents of a file

To search a text file for a string of characters or a regular expression use the command:

`grep` *`pattern filename(s)`*

Using this command you can check to see if a text file holds specific information. **`grep`** is often used to search the output from a command. Any regular expression containing one or more special characters must be quoted to remove their meaning.

Examples of using the grep *command*

1. To search a file for a simple text string:

`grep copying help`

This searches the file `help` for the string `copying` and displays each line on your terminal.

2. To search a file using regular expression:

```
grep -n '[dD]on\'t' tasks
```

This uses a regular expression to find and display each line in the file tasks that contains the pattern `don't` or `Don't`. The line number for each line is also displayed.

The expression is quoted to prevent the shell expanding the metacharacters `[`, `]` and `'`. Double quotes are used to quote the single quote in `dDon't`.

3. To use the output of another command as input to the `grep` command:

```
ls -l | grep '^d........x'
```

This lists all the directories in the current directory for which other users have execute permission. The expression is quoted to prevent the shell interpreting the `^` metacharacter.

4. To redirect the results of a search to a file:

```
grep Smith /etc/passwd > smurffs
```

This searches the system password file for each occurrence of the name `Smith` and places the results of this search in the file `smurffs`. There being a lot of Smiths everywhere, this is quite a large file.

Using regular expressions with grep

The following characters can be used to create regular expressions for searching on patterns with `grep`. Always quote the regular expression. This prevents the shell from interpreting the special characters before it is passed to the `grep` command.

`c`	any non-special character represents itself
`\c`	turns off the meaning of any special character
`^`	beginning of a line
`$`	end of a line
`.`	matches any single character except a newline
`[...]`	matches any of the enclosed characters
`[^...]`	matches any character that is not enclosed
`[n-n]`	matches any character in this range
`*`	matches any number of the preceding character

Examples of using regular expressions

1. To search using enclosed characters:

```
grep -n '[dD]o' tasks
```

This uses a regular expression to find and display each line in the file `tasks` that contains the pattern `do` or `Do`. The line number for each line is also displayed.

2. To turn off the meaning of a special character:

```
grep '^\$' money
```

This lists all the lines in the file `money` that begin with a `$` (dollar) sign. This character is preceded by a `\` (backslash) to remove its meaning as a special character.

3. To search from the beginning of a line:

```
ls -l | grep '^d........x'
```

This lists all the directories in the current directory for which other users have execute permission.

4. To search for lines ending in a particular pattern:

```
ls -l | grep -v '\.xdh $'
```

This lists all the files and directories in the current directory which do not end in `.xdh`.

Searches with the `fgrep` utility

`fgrep` is a fast searching utility that searches for multiple fixed strings simultaneously. Its main use is for directory or bibliographic type searches. For example:

```
fgrep -f surnames addresses
```

This searches the file `addresses` for each name held in the file `surnames`; a text file with one name per line.

▼

Text editors provide commands which you can use for searching the contents of a file. For more information see "*Searching for text*" on page 8-28.

Similar search facilities are also available with pager programs such as `less` and `more`. For more information see "*Displaying files using a pager*" on page 6-14.

Sorting the content of a file

The **`sort`** command is a simple database tool. It allows you to specify the field to be sorted on and the type of sort to be carried out on it.

```
sort [options] filename
```

Lines are sorted into the following order: lines starting with numbers come first, followed by lines starting with upper-case letters, which are followed by lines starting with lower-case letters and finally symbols such as `%` and `!`.

Examples of using the `sort` *command*

These examples include the use of options and features explained in later sections. Read these sections if you cannot understand an example.

1. To sort a file on field2 (the third word):

```
sort +2 -3 names
```

This sorts the file `names` on the third word of each line.

2. To sort several files using a different sort order and send the results to another file:

```
sort -n -o report +1 -2 month[1-3]
```

This sorts the files `month1` through `month3` on field1 (the second word) according to their numeric value and places the result in the file `report`.

3. To sort a file on a specific part of one field, using a different field delimiter, removing duplicate lines and sending the output to another file:

```
sort -t: -o birthdays +3.2 -4.5 friends
```

This sorts the file `friends` on the third and fourth characters of field3 (fourth word). Each field in this file is delimited by a `:` (colon). The results are placed in the file `birthdays`.

Sorting on a specific field

Lines are sorted character by character, starting at the first character in the line. You can also sort the contents of a file on a specific part of each line.

Each line of text is a series of *fields* -words and other characters- serparated from each other by a delimiter character - by default, the spaces between the words.

Changing the field delimiter

Unless you specify otherwise `sort` treats the one or more space characters between words as field delimiters. You may want to sort files where words are separated by another character such as a colon (`:`).

Use the `-t` option to specify another character to act as the field delimiter.

For example

```
-t:
```

specifies the colon (`:`) character as the delimiter.

One specific delimiter followed by another indicates an empty field. For example with a colon (`:`) as a delimiter

`field0`:*`field1`*::*`field3`*:*`field4`*

`field2` is taken to be empty.

Defining the sort field

The first field of each line starts at 0 (zero); the second is 1 (one) and so on.

To define which field to sort on you give the position of the field at which to start the sort followed by the position at which to end the sort.

The position at which to start the sort is given as the number of fields to skip to get to this position. For example

```
+2
```

tells `sort` to skip the first two fields.

The position at which to stop the sort is given as the number of the field at the end of which the sort stops. For example

```
-3
```

tells `sort` to stop the sort at the end of field three.

To sort on the third field of a line the definition:

```
+2 -3
```

To sort on the fields 5 and 6:

```
+4 -6
```

Sorting on several fields

You can sort a file on several fields with the same `sort` command. The file is sorted on the first field; if some lines have the same value for this field they are sorted on the next field you specify and so on

This is the file `scores` each line of which gives the initial, surname, gender, age and test score of a person. Each field is separated by a space.

```
J. Arnott F 43 78
T. Rice M 42 67
M. Macdonald F 39 85
A. Board M 36 43
M. Mouzin F 27 57
R. Bramley M 42 67
```

To sort the file on two fields, first by age (field4) and then by score (field5), placing the output in the file results.

```
sort -o results +3 -4 +5 scores
```

This gives the following result in the file results.

```
M. Mouzin F 27 57
A. Board M 36 43
M. Macdonald F 39 85
T. Rice M 42 67
R. Bramley M 42 75
J. Arnott F 43 78
```

From the first sort on field4 (age) sort finds that `Rice` and `Bramley` are the same age so it now sorts these two lines on field5 (score), placing `Rice` (67) before `Bramley` (75).

Sorting on a specific part of a field

You can also sort on a specific part of a field if necessary.

To define which part of a field to sort on you give the position in the field at which to start the sort followed by the position in the field at which to end the sort.

The position at which to start the sort is given as the number of fields to skip followed by the number of characters to skip. For example

```
+2.3
```

tells sort to skip the first two fields and the first three characters of field3.

The position at which to stop the sort is given as the number of the field followed by number of the character in that field at which the sort is to stop. For example:

```
-3.6
```

tells `sort` to stop the sort at the 6th character in field three.

To sort on the fourth and fifth characters of field three:

```
+2.3 -3.6
```

Saving the results of a sort to a file

`sort` sends its results directly to your screen. To save them to a file use the `-o` option. For example:

```
sort -o mail addresses
```

This saves the results of sorting the file `addresses` in the file `mail`.

Checking if a file has been sorted

You can check if a file has already been sorted with the `-c` option. For example:

```
sort -c +2 -3 accounts
```

This checks to see if the file `accounts` has already been sorted on the third field of each line. You will only get a message if the file is not sorted according to this sequence.

Merging sorted files

You can merge two or more files at the same time as they are sorted. For example:

```
sort -m -o accounts +1 -2 month[1-3]
```

This sorts the files `month1` through `month3` on the information in the second field of each line of each file. The results are merged and sent to the file `accounts`.

Files that have already been sorted can be merged using only the `-m` option. For example:

```
sort -m first_qtr second_qtr  halfyear
```

This merges the contents of files `first_qtr` and `second_qtr` and places the result in the file `halfyear`.

The > character is used to direct the output of the sort command into the file halfyear.

Changing the sort order

`sort` has a number of options which allow you to change the order in which files are sorted. This enables you to match the action of the sort command to the type of data you have.

Options that can be used to change the sort order

Always use these options immediately after the sort command. For example

```
sort -n +1 -2 costs
```

sorts the data in field 2 of the file costs by its arithmetic value.

`-d`	(Dictionary order) sort only uses numbers, letters and blank spaces when sorting the file.
`-f`	(Fold in lower case) sort treats all lower-case letters as if they were upper-case letters.

`-i`	(Ignore non-printing characters) sort ignores any characters that do not print.
`-n`	(Numeric order) sort recognises the value of numbers and sorts them in terms of their value.
`-r`	(Reverse order) sort reverses its default sort order.
-M	sort fir st three non-blamk characters of filed into month order. JAN, FEB etc.

Examples of changing the sort order

1. To sort a file in dictionary order:

   `sort -d +1 -2names  orders`

   Lines in the file `names` are sorted on field 2 into dictionary order and the result placed in the file `orders`.

2. To sort a file in month order:

   `sort -M +2 -3 -o breakdown orders`

   Lines in the file `orders` are sorted on field3 into month order and the result placed in the file `breakdown`.

Removing multiple occurrences of a line

Used with its `-u` option the `sort` command removes duplicate lines from a file. For example:

`sort -u -o mail_labels addresses`

This removes all duplicate lines from the file `addresses` and places the output in the file `mail_labels`.

Duplicate lines do not have to be consecutive for the `sort` command to identify and remove them.

14

Programming utilities

14 Programming utilities

UNIX provides you with a comprehensive programming environment. There are tools for:

- automating program builds
- checking program syntax
- de-bugging source code
- maintaining different versions of source files

Using the Source Code Control System

The Source Code Control System is an essential tool for any project with multiple source files or which has several people working with multiple source files. It provides you with a way to keep track of a file's development and to prevent it from being altered by more than one person at a time.

Putting files under SCCS control

Files are placed under the control of SCCS by making a SCCS directory and creating the SCCS file. You also have the option of placing ID keywords in each file.

Making an SCCS *directory*

To create the SCCS directory use the command:

```
mkdir SCCS
```

The directory name *must* be in upper case.

Inserting SCCS *ID keywords in a file*

You can put one or more ID keywords in each file. These keywords are used to display information about the file whenever a read only version is checked out.

SCCS ID keywords include:

`%G%`	- the date of the delta
`%I%`	- SID[1] of the retrieved text
`%M%`	- the current module name
%R%	- the current release number
`%W%`	- a shorter form for `%Z%%M%<tab>%I%`
`%Z%`	- string used by the **what** command

Creating a SCCS file

To place files under the control of SCCS use the command:

```
sccs create filename(s)
```

For each filename given as an argument `sccs create` does the following:

- *filename* is moved to `,`*filename*
- a history file `s.`*filename* is created in the `SCCS` subdirectory
- a "read only" copy of the initial version of the file is made by **sccs get**.

You can use the `ls` command to check that the history files have been created.

```
ls SCCS
```

Now you can remove the `,`*filename* files:

```
rm ,*
```

1. SCCS delta ID. See "*Checking in files*" on page 4 of this section.

Checking out files

You can check out a file to read it or edit it and then check it back in again. You can also check out earlier versions of a file.

Check out a file for reading

To get a copy of one of your files for reading use the command:

```
sccs get filename
```

To get a copy of each of your files for reading use the command:

```
sccs get SCCS
```

X Caution

When you have finished using a file that was fetched from `SCCS` by `sccs get` do not attempt to replace it there: *remove it*.

Check out a file for editing

To check out a copy of one of your files for editing use the command:

```
sccs get -e filename
```

SCCS will record your login name. No other users can check out this file but they can see who it is checked out to. If a read-only copy of this file already exists then it will be overwritten.

To check out a copy of each of your files use the command:

```
sccs get -e SCCS
```

Check out an early version of a file

You can check out an earlier version of a file use the command:

```
sccs get -rversion_number
```

Cancelling an editing session

You check out a file, edit it, and then decide that you don't want this new version.

```
sccs unedit filename
```

This removes the file and reverts to the previous version number.

Checking in files

After editing a file you can check it in with the command:

```
sccs delta filename
```

SCCS will then:

- make sure you are the user who checked it out.
- ask you for a descriptive comment on the changes you have made.
- record what changes made.
- removes the writeable copy of the file from the current directory.

Getting information about files

SCCS provides facilities for getting the following types of information about files under its control.

Which files are being edited?

You can get information about which files are being edited, and who is editing each one, with the command:

```
sccs info
```

To list the files being edited by a particular person use the command:

```
sccs info -uusername
```

What is the latest versions of each file?

To see which versions of each file are current use the command:

```
sccs what filename(s)
```

What is the history of this file?

To see the recorded history of any file under the control of SCCS use the command:

```
sccs prt filename
```

This displays:

- the version number
- the date and time of its creation
- who made this version
- line-by-line change statistics
- the comments written by the user

Creating new releases

To begin a new release for a file you "get" it with the next release number:

```
sccs get -e -r release_number filename
```

For example for a file with a version of 2.4 you would want the new release to be 3.

```
sccs get -e -r3 sect1.txt

s.sect1.txt:
2.4
new delta 3.1
20 lines
```

Comparing files under SCCS control

You can compare any two versions of a file that is under the control of SCCS with the command:

```
/usr/sccs/sccsdiff -rversion -rversion s.filename
```

Using this command produces a line by line difference between the two versions of the file. Results are shown in the same format as that produced by the `diff` command. For more information see "*Comparing files*" on page 13-1.

Examples of comparing files with the sccsdiff *command*

1. To compare two versions of the same SCCS file:

```
/usr/sccs/sccsdiff -r1.3 -r1.5 s.foo.c
SCCS/s.foo.c: 1.3 vs. 1.5
21c21
< static char exname[] = "Template Xternals";
----
> static char exname[] = "Template XTernals";
```

This displays a line by line difference between versions 1.3 and 1.5 of the SCCS file `s.foo.c` which is in the current directory.

The two versions of this file differ at line 21, where the variable name has been changed from `Xternals` to `XTernals`.

2. To compare two versions of two or more SCCS files:

```
/usr/sccs/sccsdiff -r1.2 -r1.3 s.part[1-3].c
```

This displays a line by line difference between versions 1.2 and 1.3 of the SCCS files `s.part1.c` through `s.part3.c` which are in the current directory.

Program development using shell scripts

If the source code of your program is in one file, you will have to compile all of it after every change you make to it. If the command line for this compilation includes options and specifications of libraries you should consider using a shell script.

For example, you could save these lines in a file called `mycomp` by using your favourite editor:

```
#!/bin/sh
cc -o myprog myprog.c -lnag
```

The first line `#!/bin/sh` is there to ensure that the Bourne shell is used to execute this script. This shell is available on all UNIX systems. The last line compiles the C source in the file myprog.c linking in the necessary libraries of object programs.

Give yourself execute access permission on the file `mycomp` by issuing the command:

```
chmod +x mycomp
```

You can now compile your code by typing:

```
mycomp
```

To run your program you would enter:

```
myprog
```

You could improve this shell script by using positional parameters and changing it to:

```
#!/bin/sh cc -o $1 $1.c -lnag
```

Simply use the name of the file you wish to compile as the argument of the shell script `mycomp`:

```
mycomp myprog
```

When writing programs with more than one source file, there is the possibility of inconsistencies between source and object files and between the options and libraries used at compiling and loading. The **make** program can help you avoid such inconsistencies and save unnecessary recompilations. See "*Building programs with the make utility*" on page 9 of this section.

You can call make from within a shell script, to increase the amount of work achieved by your one script command. Suppose you want to work within a particular directory and to tidy away all unnecessary files. Then the following script, using one of the makefile files described in the next section, might suit your needs:

```
#!/bin/sh
cd Sect1043
make
if test $? -ne 0
then echo
echo An error has occurred: fix it and rerun script
cd ..
exit
else
echo
echo Success, so now tidy up.
rm *.o
cd ..
fi
```

The $? in the third line of this script is replaced by the exit status of make i.e. it tests to see if make has been successful.

Building programs with the make utility

The **make** command is useful for updating software which is held as a number of source files, where the main program is in one and the subprograms called by it are in the others. You define each object file and list its dependants in a file - usually called `makefile` or `Makefile`.

`make` ensures that all the object files upon which the “target” object file depends exist and are up to date. It then creates each object file which has not been modified since either its source or its dependants were last altered. Doing this, `make` takes the fewest actions necessary to achieve the updating.

`make` is frequently used to compile UNIX software which is available in the public domain or as shareware. The software distribution will contain a `makefile` (or `Makefile`) together with the files containing the source code for the program and any specific libraries that this uses. Before rebuilding the software it is often necessary to edit the `makefile` to take account of conditions on the local host. For example, the path to object libraries referenced in the `makefile` may be different on the local host where `make` is run.

The format of the `make` command is:

```
make [flags] [macro definitions] [objects]
```

When the command is issued without an argument, the first object file mentioned in the file `makefile` or `Makefile` in the current directory is created.

If your specifications are in a file with a name other than `makefile`, then introduce it with the flag `-f`. For example:

```
make -f mycomp
```

More than one file may be introduced in this way. For example:

```
make -f myload -f mycomps
```

You can tell `make` that the standard input (usually your terminal) is to be used instead of `makefile` with the command:

```
make -f -
```

You then type the specifications according to the rules which follow, and indicate the end of these specifications by typing `^D`.

Format of makefile

Suppose that the source of your Fortran program is held in four files, `myprog.f`, `mysubs.f`, `partic.f` and `commons`, and that the first two contain an `include` referring to the last.

You want to compile each `.f` file and load the three object files and relevant elements of a library into a file called `myprog`. You could create a `makefile` such as this:

```
myprog: myprog.o mysubs.o partic.o
                              epcf77 -ats -fpa
myprog.o mysubs.o partic.o -lnag\
                              -o myprog
# epcf77 is the Edinburgh Portable Compilers
# Fortran 77 compiler
myprog.o: myprog.f commons
                              epcf77 -c -ats -fpa
myprog.f
mysubs.o: mysubs.f commons
                              epcf77 -c -ats -fpa
mysubs.f
partic.o: partic.f
                              epcf77 -c -ats -fpa
partic.f
```

The structure of all four paragraphs is the same:

- a line starting with the name of the object file, followed by a list of the names of the files upon which it depends: a colon separates these two components.

- a line giving the shell commands to build the file named in the previous line. *This line must begin with a tab.*

If either line is too long to be literally one line, you can break it at a convenient point by typing a `\` and continuing on the next line after a tab.

Comments in makefile

You can write comments in your makefile by typing a line consisting of the character `#` followed by the text of your comment. You can also type spaces (but *not* tab characters) before the `#` to format the makefile.

Arguments with make

You can pass the names of "target" object files defined in `makefile` as arguments to `make`. For example:

```
make mysubs.o
```

The entry defining `mysubs.o` will then be executed if any of its dependencies has been altered since `mysubs.o` was last created.

Macros with make

The first paragraph of the example `makefile` contains a list of object files twice, once in the list of dependencies and again in the `epcf77` statement. Any change in one list must be made in the other list. The possibility of errors due to inconsistencies in these lists may be reduced by using a macro.

Add the following line at the beginning of your `makefile` to define a macro:

```
OBJ = myprog.o mysubs.o partic.o
```

The first paragraph can now be rewritten to use this macro:

```
myprog: $(OBJ)
                                    epcf77 -ats -fpa
```

```
$(OBJ) -lnag -o myprog
```

A macro is defined by a statement of the form:

```
name = value
```

The convention is that name be in upper case letters (see below). A macro is used by typing a dollar sign followed by the macro name in brackets: the brackets are optional if the name is a single letter. For example, you could define and use a macro for Fortran as:

```
F = epcf77 -ats -fpa
myprog: $(OBJ)
                              $F $(OBJ) -lnag -o
myprog
```

Overriding macro values

The values defined in macros can be altered by command line options to `make`. For example, if you define a macro for using with Fortran as:

```
FFLAGS = -ats -fpa
```

The line to create `myprog.o` could become:

```
epcf77 $(FFLAGS) -c myprog.f
```

And the final load becomes:

```
epcf77 $(FFLAGS) $(OBJ) -lnag -o myprog
```

Now issue the command:

```
make "FFLAGS=-ats -fpa -x"
```

The `-x` flag as well as the `-ats` and `-fpa` flags is passed to `epcf77`. Note that these names `must` be in upper case if the second rule in the next subsection is to apply.

Suffix rules in make

`make` has two rules which allow you to abbreviate the information which you give in your `makefile` about any object file:

1. If the object file is, say, `myprog.o`, and there is a file `myprog` with suffix `.c`, `.f` or `.p` in either the `makefile` or your current directory, then `make` assumes that this is the required source file.
2. If the command to make the relocatable file is omitted, then the relevant rule in the following table is applied:

Suffix rules in make

source code suffix	command assumed	flags from menu
.f	fortran -c	FFLAGS
.c	cc -c	CCFLAGS
.p	pascal -c	PPFLAGS

Hence our example `makefile` could become:

```
OBJ = myprog.o mysubs.o partic.o
FFLAGS= -ats -fpa
myprog: $(OBJ)
epcf77 $(OBJ) -lnag -o myprog
myprog.o: commons
mysubs.o: commons
partic.o:
```

Strict syntax is that the first line of each paragraph is a space-separated list of object files followed by a colon and then a space-separated list of dependants. This and the rules just mentioned allow us to abbreviate the specifications to:

```
OBJ = myprog.o mysubs.o partic.o
FFLAGS = -ats -fpa
myprog: $(OBJ)
epcf77 $(OBJ) -lnag -o myprog
myprog.o mysubs.o: commons
partic.o:
```

Options with make

You can use options to control the behaviour of make.

`-f`	Allows you to specify where the input to make comes from (see earlier examples).
`-n`	Prints but does not execute those commands which would have to be executed to update the object files defined in makefile or its alternative.
`-s`	Suppresses the printing of each command line as it is executed.

Each command line in `makefile` is executed by its own shell, one at a time, and a line is printed to let you know how the updating is progressing.

You can suppress this printing either by using the flag `-s` (see above) or by defining the special item `.SILENT`. For example:

```
OBJ = myprog.o mysubs.o partic.o
FFLAGS = -ats -fpa
.SILENT:
myprog: $(OBJ)
epcf77 $(OBJ) -lnag -o myprog
myprog.o: myprog.f commons
epcf77 -c $(FFLAGS) myprog.f
mysubs.o: mysubs.f commons
epcf77 -c $(FFLAGS) mysubs.f
partic.o: partic.f
epcf77 -c $(FFLAGS) partic.f
```

Alternatively, to suppress only some of the printing, type an '`@`' as the first character of each command which you do not wish to see printed out as it is executed. For example:

```
OBJ = myprog.o mysubs.o partic.o
FLAGS = -ats -fpa
myprog: $(OBJ)
epcf77 $(OBJ) -lnag -o myprog
myprog.o: myprog.f commons
@epcf77 -c $(FFLAGS) myprog.f
mysubs.o: mysubs.f commons
@epcf77 -c $(FFLAGS) mysubs.f
partic.o: partic.f
@epcf77 -c $(FFLAGS) partic.f
```

Each occurrence of `.SILENT` and `@` does no more than suppress the printing of the command line; neither suppresses printing reports from the compiler.

Debugging programs with dbx

`dbx` is an interactive, line-orientated debugger which operates at source-level. Source code may be in C, Fortran, or Pascal. Use it to:

- determine where a program crashes
- view the values of variables and expressions
- set breakpoints in code
- single step a program
- run and trace a program

This section explains how `dbx` can be invoked to debug single process applications. The version described here is that supplied with Sun OS, other versions of this tool may differ in the facilities they provide. Check the local manual page for this command.

The information in this section should also enable you to begin using debugging tools running in GUI environments such as `dbxtool` for Sun View and `xdbx` for X windows. For the most part, these clients are wrappers around the `dbx` tool.

As `dbx` examines special symbol tables which are not available by default it is necessary to compile with the `-g` option present. For example:

```
cc -g -o myprog myprog.c
```

▼

On some systems, optimization of the source code is *not* carried out when the `-g` option is used; on others it is allowed. Be aware that if you do optimize, that the program image might not be an accurate interpretation of the source code.

Starting dbx

To start debugging a program use the command:

```
dbx program_name
```

Enter the command without a program name and you are prompted for the object file name:

```
enter object file name (default is 'a.out'):
```

You do not have to give the name of the program source file; `dbx` will determine this from the executable's symbol table.

If there is a `core` file in the current directory or the `-corefile` argument has been supplied, you can use `dbx` to examine the state of the program when the `core` file was produced. For example:

```
dbx myprog core
```

This tells `dbx` to read the executable file `myprog` and its default `core` image file.

Configuring dbx

Any of the requests for configuring the action of `dbx`, which are described in this document and in the manual page can be placed in the file `.dbxinit`. These requests are executed when `dbx` is invoked and before the symbol table is read in. `dbx` looks for this file first in your current directory and then in your home directory. For example, suppose that your `.dbxinit` file contains the following lines:

```
setenv EDITOR emacs
use $HOME/project/src
alias x quit
```

Now, when invoked, `dbx` will use the `emacs` editor by default, look for source files in the directory `$HOME/project/src` and accept `x` as an alias to the `quit` command.

Getting help

`dbx` has its own help facility, called by typing `help`. For example:

```
(dbx) help help
help - Print a summary of all commands
help <cmd> - Print help about <cmd>
(dbx)
```

If you have experience of programming in a high level language, you might find that this help information is sufficient to enable you to use `dbx`.

Running a program

To run the current program from within `dbx` use the command:

```
(dbx)run
```

The program will then run and crash, giving a message about the bug which is the cause of this. This message often contains enough information to enable you to fix the bug. To find the exact location of the bug use the `where` command. For example:

```
(dbx) where
abort() at 0xfed3
_Fabort(0x21) at 0x656f
_Fatal(0x21, 0x629f, 0x0, 0x0) at 0x5e61
MATHERR.MATHERR(0x3fffebfc, 0xf7b1) at 0xd2a7
sqrt.sqrt(0x13f24) at 0xf7ad
MAIN(), line 6 in "dtest.f"
main.main(0x1, 0x3fffedf8, 0x3fffee00) at 0xa3c9
(dbx)
```

This suggests that the error arose in line 6 of the main program when a routine to calculate the square root of a number is called.

Listing source files

To examine the source for the program at the point at which `dbx` indicates that it fails use the **`list`** command:

```
(dbx) list first_line,last_line
      procedure
```

Editing source files

To edit the source file for the current program use the **`edit`** command:

```
(dbx) edit
```

To edit a specific program source file:

```
(dbx) edit filename
```

The default editor called by `dbx` is `vi`. For more information see "*Using the vi editor*" on page 8-2. To change to another editor use the command:

```
(dbx) setenv EDITOR name_of _editor
```

Set this variable in the file `.dbxinit` to configure `dbx` so that this is the default editor. See "*Configuring dbx*" on page 17 of this section.

Using breakpoints

To assist debugging, `dbx` allows you to set *breakpoints*, before each line of a program or sub-program. The **`stop`** request is used to set a breakpoint. For example:

```
(dbx) stop at 5
```

You can give this request a conditional expression to avoid needless single- stepping. For example:

```
(dbx) stop at 5 if (i==1)
```

The **`status`** request lists active breakpoints: the **`delete`** request removes them. For example:

```
(dbx) status
(23) stop if 'prog'main'i=-1
(dbx) delete 23
```

Notice that you use the identification number that `dbx` uses to refer to the breakpoint `(23)` and not the line number or condition. You can delete several breakpoints with a single command. For example:

```
(dbx) delete 23, 29, 53
(dbx)
```

This deletes the breakpoints with the identification numbers 23, 29, and 53.

Once you have set a breakpoint you can use the `run` request to execute the program.

```
(dbx) run
```

The program will stop at the first breakpoint. You can now proceed in one of several ways:

```
(dbx) cont
```

This request continues execution after a stopping point until the program finishes or encounters another breakpoint.

```
(dbx) step
```

This request executes one source statement at a time, in functions as in main programs.

```
(dbx) next
```

This request executes one source statement, but does not take statements separately inside any function calls.

Printing variable values

The request `print` can be used at any time during statement stepping to give the value of a variable. For example:

```
(dbx) print x
`prog'main'x = 0
```

Note that the Sun version of `dbx`, which is described here uses single left quotes to delimit fields in variable names. Other versions use a full-stop . (or period).

Other useful requests are `whatis`, which identifies the type of the array or variable and `whereis` which shows where an array, variable or function is defined

Tracing execution

The `trace` request allows you to watch the execution of your program. It prints out each line of the source code that it executes. A full, unlimited trace will produce too much information for your needs and will slow down the speed at which the program is executed. To use trace in a narrower scope, two of the most useful options are:

```
(dbx) trace variable
```

and

```
(dbx) trace function
```

Tracing can be further restricted by making it conditional. For example:

```
(dbx) trace log(x) if x < 1
(28) trace log if `prog`main`x < 1
(dbx) run
```

Quick reference: dbx requests

Executing and tracing

`run`	- begin execution of the program
`cont`	- continue execution
`trace <line#>`	- trace execution of the identified line
`trace <proc>`	- trace calls to the procedure
`trace <var>`	- trace changes to the variable
`trace <exp> at <line#>`	
	- print `<exp>` when `<line#>` is reached
`stop at <line>`	- suspend execution at the line
`stop in <proc>`	- suspend execution after `<proc>` is called
`status`	- print trace/stops in effect
`delete <number>`	- remove trace or stop of given
`catch`	- start trapping specified signals
`ignore`	- stop trapping specified signals
`step`	- single step one line, stepping into functions
`next`	- step to next line, even if it is a function
`ps`	- print status of processes/core dumps

Displaying data

`list <line>,<line>`	- list source lines
`print <exp>`	- print the value of the `<expression>`
`whatis <name>`	- print the declaration of the `<name>`
`which`	- print outer block associated with the identifier
`whereis <name>`	- print all symbols matching identifier
`assign`	- set the value of a variable

Function and procedure handling

`where`	- print currently active procedures
`down`	- move down the stack towards the stopping point
`up`	- move up the stack towards main
`call <proc>`	- call a `<procedure>` in program
`dump`	- display names and values of all local variables
`%<number>`	- switch to context of process `<number>`

Accessing source files and directories

`edit`	- invoke the editor on current source file
`file`	- change current source file
`func`	- change the current function or procedure
`list`	- display lines of source code
`use`	- set directory list to search for source files
`/.../`	- search down in file to match regular expression
`?...?`	- search up in file to match regular expression

Miscellaneous commands

`sh`	- pass command line to the shell
`alias`	- change dbx command name
`help`	- explain command
`source`	- read commands from external file
`quit`	- exit dbx

15

Shell programming

15 Shell programming

As well as using the shell to run commands you can use its built-in programming language to write your own commands or programs. You can put commands into a file -known as a *shell script*- and then execute that file as you would a command or program.

The use of the Bourne shell (**sh**) is illustrated as this is available on all UNIX systems. Bourne shell scripts will also work under the Bourne Again SHell. The C and TC shells use a programming language which is similar to the C programming language.

Creating and executing simple shell scripts

To create a simple shell script, you can put command lines into a file, set the proper access permissions and then execute the file.

Example of a simple shell script

1. These are the contents of a shell script called display:

```
$ cat display
# This script displays the date, time, username and
# current directory.
echo "Date and time is:"
date
echo
echo "Your username is: `whoami` \n"
echo "Your current directory is: \c"
pwd $
```

The first two lines beginning with a # hash are comments and are not interpreted by the shell. Use comments to document your shell script; you will be surprised how easy it is to forget what your own programs do!

The backquotes (‘) around the command **whoami** illustrate the use of command substitution. For more information see "*Command substitution*" on page 5-11.

The `\n` is an option of the **echo** command that tells the shell to add an extra carriage return at the end of the line. The `\c` tells the shell to stay on the same line. See the man page for `echo` for details of other options.

The argument to the `echo` command is quoted to prevent the shell interpreting these characters as though they had been escaped with the `\` (backslash) character.

Creating the shell script

To create a script use a text editor to create a new file containing the commands that you want the script to execute.

Executing a shell script

Before using a file as a shell script you must change its access permissions so that you have execute permission on the file, otherwise the error message

```
Permission denied
```

is displayed. For more information see "*Changing access permissions with the chmod command*" on page 4-7.

To run the shell script, simply type its name at the prompt. The commands in the script will then execute one at a time as though you were typing them in at the terminal.

To give yourself execute permission for the file containing the script use the command:

```
$chmod u=rwx filename
```

The `=rwx` after the u allows you to read, write to and execute the script: no one else has permission to read, write or execute.

To give other users permission to read and execute but not alter the shell script use:

```
$ chmod go=rx filename
```

Example of using a shell script

1. To make a file containing a shell script executable and then run the script:

```
$ display
display: execute permission denied
$ chmod u+x display
$ display
Date and time is:
Mon Mar 8 10:51:17 GMT 1993
Your username is: erpl08
Your current directory is: /home/erpl08/scripts
$
```

The command **display** does not run the script as the user does not have execute permission for the file. They give themselves execute permission and then re-enter the command to run the script.

You can also use command line options to debug your script. See "*Debugging shell scripts*" on page 27 of this section.

Using the Bourne shell to interpret a shell script

To make sure that your shell script is always run from a standard Bourne shell have the line

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

as the first line of the script.

Problems running shell scripts

Message	Meaning
`command_name: command not found`	Your shell's search path does not include the directory that the shell script is stored in. Move the file containing the shell script to the directory $HOME/bin. This directory is usually in your default search path.
`various messages`	Meaning There are errors in the construction and syntax of the shell script. Read through your script checking its logical operations and the commands that it uses.

Bourne shell programming

The shell also provides you with a programming environment with features similar to those of a high level programming languages

Examples of Bourne shell scripts

1. To read input to a command and process it in some way:

```
# usage: fsplit file1 file2
total=0; lost=0
while read next
do
   total=`expr $total + 1`
   case "$next" in
      *[A-Za-z]*) echo "$next"  $1 ;;
```

```
        *[0-9]*) echo "$next"  $2 ;;
        *) lost=`expr $lost + 1`
esac
done
echo "$total lines read, $lost thrown away"
```

The user types the command:

fsplit *file1 file2*

They then enter lines of text and issue an EOF instruction. The script then processes the lines as follows

A line with at least one letter is appended to *file1*; any line with at least one digit and no letters is appended to *file2*. All other lines are thrown away.

2. To read commands from the terminal and process them:

```
# usage: process sub-directory
dir=`pwd`
for i in *
do
    if test -d $dir/$i
    then
        cd $dir/$i
        while echo ''$i:''
            read x
        do
            eval $x
        done
    fi
done
```

The user types the command:

process *sub-directory*

This script will read and process commands in the named sub-directory. The user is prompted to supply the name of the command to be read in. This command is executed using the built-in `eval` function.

3. To create a command:

```
flag=
for i
do
   case $i in
      -c) flag=N ;;
      *) if test -f $i
      then
         ln $i junk$$; rm junk$$
      elif test $flag # true if not null
      then
         echo '$i' does not exist
      else
         >$i
      fi ;;
   esac
done
```

This command takes filenames as its parameters. If a file exists it changes the modification date. If no file exists it creates a new one. This script is similar in action to the **`touch`** command.

The `-c` argument lets you specify that you only want to update a file that already exists and not to create one if it doesn't.

Passing arguments to the shell

Shell scripts can act like standard UNIX commands and take arguments from the command line.

Arguments are passed from the command line into a shell program using the positional parameters `$1` through to `$9`. Each parameter corresponds to the position of the argument on the command line.

The positional parameter `$0` refers to the command name or name of the executable file containing the shell script.

Only nine command line arguments can be accessed, but you can access more than nine using the `shift` command This makes what was `$2` now `$1`, and what was `$3` now `$2` etc. See "*Using the shift command*" on page 8 of this section.

All the positional parameters can be referred to using the special parameter `$*`. This is useful when passing filenames as arguments.

Examples of passing arguments to the shell

1. To pass several arguments from the command line to the shell:

```
$ cat first_5args
# This script echos the first five arguments
# supplied to the script
echo The first five command line
echo arguments are $1 $2 $3 $4 $5


$ first_5args mines a pint john o.k.
The first five command line
arguments are mines a pint john o.k.
$
```

This passes the arguments represented by parameters `$1` through `$5` to the `echo` command.

2. To pass the value of each positional parameter to the shell script:

```
$ cat printps
# This script converts ASCII files to PostScript
# and sends them to the PostScript printer ps1
# It uses a local utility "a2ps"
a2ps $* | lpr -Pps1
$ printps elm.txt vi.ref msg
```

This processes the three files given as arguments to the command `printps`.

Using the shift command

Usually only nine command line arguments can be accessed using positional parameters. The `shift` command gives access to command line arguments greater than nine by shifting each of the arguments.

The second argument (`$2`) becomes the first (`$1`), the third (`$3`) becomes the second (`$2`) and so on. This gives you access to the tenth command line argument by making it the ninth. The first argument is no longer available.

Successive shift commands make additional arguments available. Note that there is no "unshift" command to bring back arguments that are no longer available!

Example of using the shift *command*

1. To successively shift the argument that is represented by each positional parameter:

```
$ cat shift_demo
echo "arg1=$1 arg2=$2 arg3=$3"
shift
echo "arg1=$1 arg2=$2 arg3=$3"
shift
echo "arg1=$1 arg2=$2 arg3=$3"
shift
echo "arg1=$1 arg2=$2 arg3=$3"

$ shift_demo one two three four five six seven
arg1=one arg2=two arg3=three
arg1=two arg2=three arg3=four
arg1=three arg2=four arg3=five
arg1=four arg2=five arg3=six
arg1=five arg2=six arg3=seven
```

Handling shell variables

The shell has several variables which are automatically set whenever you login.

The values of some of these variables are stored in names which collectively are called your user environment. For more information see "*Managing your working environment*" on page 7-1.

Any name defined in the user environment, can be accessed from within a shell script. To include the value of a shell variable into the environment you must have exported it. For more information see "*Setting shell variables*" on page 7-9.

Special shell variables

There are some variables which are set internally by the shell and which are available to the user:

Name	Description
$1 - $9	these variables are the positional parameters.
$0	the name of the command currently being executed.
$#	the number of positional arguments given to this invocation of the shell.
$?	the exit status of the last command executed is given as a decimal string. When a command completes successfully, it returns the exit status of 0 (zero), otherwise it returns a non-zero exit status.
$$	the process number of this shell - useful for including in filenames, to make them unique.

$!	the process id of the last command run in the background.
$-	the current options supplied to this invocation of the shell.
$*	a string containing all the arguments to the shell, starting at $1.
$@	same as above, except when quoted.

Note

$* and $@ when unquoted, are identical and expand into the arguments.

"$*" is a single word, comprising all the arguments to the shell, joined together with spaces. For example '1 2' 3 becomes one argument "1 2 3".

"$@" is identical to the arguments received by the shell, the resulting list of words completely match what was given to the shell. For example '1 2' 3 becomes two arguments "1 2" and "3"

Evaluating shell variables

The following set of rules govern the evaluation of all shell variables.

Definition	Description
$var	signifies the value of var or nothing, if var is undefined.
${var}	same as above except the braces enclose the name of the variable to be substituted.
${var-thing}	value of var if var is defined; otherwise thing. $var is not set to thing.
${var=thing}	value of var if var is defined; otherwise thing. If undefined

	$var is set to thing.
${var?message}	If defined, $var; otherwise print message and exit the shell. If the message is empty, print a standard message.
${var+thing}	thing if $var is defined, otherwise nothing.

Reading user input

To read standard input into a shell script use the `read` command. For example:

```
echo "Please enter your name:"
read name
echo "Welcome to Edinburgh $name"
```

This prompts the user for input, assigns this to the variable `name` and then displays the value of this variable to standard output.

If there is more than one word in the input, each word can be assigned to a different variable. Any words left over are assigned to the last named variable. For example:

```
echo "Please enter your surname"
echo "followed by your first name: c"
read name1 name2
echo "Welcome to Glasgow $name2 $name1"
```

Conditional statements

Every UNIX command returns a value on exit which the shell can interrogate. This value is held in the read-only shell variable $?.

A value of `0` (zero)signifies success; anything other than `0` (zero) signifies failure.

The `if` statement

The if statement uses the exit status of the given command and conditionally executes the statements following. The general syntax is:

```
if test
then commands (if condition is true)
else commands (if condition is false)
fi
```

`if`, `then`, `else` and `fi` are shell reserved words and as such are only recognised after a newline or `;` (semicolon). Make sure that you end each `if` construct with a `fi` statement.

`if` statements may be nested:

```
if ...
then ...
else   if ...
       ...
       fi
fi
```

The `elif` statement can be used as shorthand for an `else if` statement. For example:

```
if ...
then ...
elif ...
...
fi
```

Example of using an if *construct*

1. To carry out a conditional action:

```
if who | grep -s keith > /dev/null
then
   echo keith is logged in
else
   echo keith not available
fi
```

This lists who is currently logged on to the system and pipes the output through **grep** to search for the username `keith`.

The `-s` option causes grep to work silently and any error messages are directed to the file `/dev/null` instead of the standard output.

If the command is successful i.e. the username `keith` is found in the list of users currently logged in, then the message

```
keith is logged on
```

is displayed; otherwise the second message `keith is not available` is displayed.

The && operator

You can use the `&&` operator to execute a command and, if it is successful, execute the next command in the list. For example:

```
cmd1 && cmd2
```

cmd1 is executed and its exit status examined. Only if *cmd1* succeeds is *cmd2* executed. This is a terse notation for:

```
if cmd1
then
    cmd2
fi
```

Example of using the && operator

1. To notify the user about the outcome of a previous command:

```
$ cat deliver
#!/bin/sh
# usage: deliver username filename
{ cat $2 | write $1 ; } && echo done
```

The user types a command such as:

```
deliver keith greeting
```

The first command `{ cat $2 | write $1 ; }` concatenates and displays the message held in the file `greeting` and pipes the output through the **`write`** command whose argument is the name of the user to whom the message is to be sent.

Note the use of the positional parameters `$1` and `$2`. The `;` (semicolon) is needed to sequentially execute the preceeding pipeline.

If this command is successful, the message

```
done
```

is displayed on standard output.

The || operator

You can use the || operator to execute a command and, if it fails, execute the next command in the command list. For example:

cmd1* || *cmd2

`cmd1` is executed and its exit status examined. If `cmd1` fails then `cmd2` is executed. This is a terse notation for:

```
cmd1
if test $? -ne 0
then
cmd2
fi
```

Example of using the || operator

1. To send a message to a user using the appropriate utility:

```
$ cat writemail
#!/bin/sh
# usage: writemail user message
echo "$2" |{ write "$1" || mail "$1" ;}
```

The user types a command such as:

```
$ writemail sarah 'call me'
```

The message entered by the user is piped through the command `{write "$1" || mail "$1" ; }`.

If the message cannot be sent to the user's terminal (they are not logged on) with the command `write "$1"` then the message is sent to the user by mail.

Testing for files and variables: the test command

The shell uses a command called `test` to evaluate conditional expressions. Full details of this command can be found in the test(1) manual page. For example:

```
if test ! -f $FILE
then
       if test "$WARN" = "yes"
       then
          echo "$FILE does not exist"
       fi
fi
```

First, we test to see if the filename specified by the variable `$FILE` exists and is a regular file. If it does not then we test to see if the variable `$WARN` is assigned the value `yes`, and if it is a message that the filename does not exist is displayed.

Flow of control statements

The Bourne shell provides several flow of control statements.

The case statement

`case` is a flow control construct that provides for multi-way branching based on patterns.

Program flow is controlled on the basis of the *word* given. This *word* is compared with each *pattern* in order until a match is found, at which point the associated *command(s)* are executed.

```
case   word   in
       pattern1) command(s)
       ;;
       pattern2) command(s)
       ;;
       patternN) command(s)
       ;;
       esac
```

When all the commands are executed control is passed to the first statement after the `esac`. Each list of commands must end with a double semi-colon *;;*.

A command can be associated with more than one pattern. Patterns can be separated from each other by a | symbol. For example:

```
case   word   in
       pattern1|pattern2) command
...    ;;
```

Patterns are checked for a match in the order in which they appear. A command is always carried out after the first instance of a pattern.

The `*` character can be used to specify a default pattern as the `*` character is the shell wildcard character. For more information see "*Defining files with wildcard characters"* on page 6-9.

Example of using the case *statement*

1. To specify an action when a word matches the pattern:

```
$ cat diary
today=`date +%m/%d`(gives date in the format 01/31)
case   $today in
       07/18)echo"Aonoch Mhor"
       ;;
       07/21)echo"Ben Wyvis"
```

```
        ;;
        08/02)echo"Buicheille Etive Mhor"
        ;;
        08/03)echo"Slioch"
        ;;
        *) echo"Wet..low level today"
esac
$ date +%m/%d
07/18
$ diary
Aonoch Mhor
$
```

The value for the word `$today` is generated by the **`date`** command. This is then compared with various patterns so that the appropriate commands are executed.

Note the use of the pattern `*`, this can be used to specify default patterns as the `*` character is the shell wildcard character. For more information see "*Defining files with wildcard characters"* on page 6-9.

The for statement

The for loop notation has the general form:

```
for var in list-of-words
do
   commands
done
```

`commands` is a sequence of one or more commands separated by a newline or `;` (semicolon).

The reserved words `do` and `done` must be preceded by a newline or `;` (semicolon). Small loops can be written on a single line. For example:

```
for var in list; do commands; done
```

Examples of using the for *statement*

1. To take each argument in turn and see if that person is logged onto the system.

```
$ cat snooper
# see if a number of people are logged in
for i in $*
do
    if who | grep -s $i  > /dev/null
    then
        echo $i is logged in
    else
        echo $i not available
    fi
done
```

For each username given as an argument an `if` statement is used to test if that person is logged on and an appropriate message is then displayed.

2. To go through each file in the current directory and compare it with the same filename in another directory:

```
# compare files to same file in directory "old"
for i in *
do
    echo $i:
    cmp $i old/$i
    echo
done
```

3. If the list-of-words is omitted, then the loop is executed once for each positional argument (i.e. assumes `$*` in the for statement). In this case the loop will create the empty files whose names are given as arguments.

```
# create all named files
for i
do
```

```
   >$i
done
```

4. Some examples of command substitution in for loops:

```
# do some thing for all files in current
# directory according to time modified
for i in `ls -t`
do
...
done
# do something for all non-fred files.
for i in `cat filelist | grep -v fred`
do
...
done
# do something to each sub-directory found
for i in `for i in *
do
   if test -d $i
   then
      echo $i
   fi
      done`
   do
...
done
```

The while and until statements

The while statement has the general form:

```
while command-list1
do
   command-list2
done
```

The commands in command-list1 are executed; and if the exit status of the last command in that list is 0 (zero), the commands in command-list2 are executed.

The sequence is repeated as long as the exit status of command-list1 is 0 (zero).

The until statement has the general form:

```
until command-list1
do
   command-list2
done
```

This is identical in function to the **`while`** command except that the loop is executed as long as the exit status of *`command-list1`* is non-zero.

The exit status of a while/until command is the exit status of the last command executed in *`command-list2`*. If no such command list is executed, a `while/until` has an exit status of `0` (zero).

Examples of using the while *and* until *statements*

1. To wait for someone to logout:

```
while who |grep -s $1 > /dev/null
do
   sleep 60
done
echo "$1 has logged out"
```

This script checks to see if the username given as an argument to the script is logged on. While they are, the script waits for 60 seconds before checking again. When it is found that the user is no longer logged on a message that they have logged out is displayed.

2. To declare when a file has been created:

```
until test -f $FILE
do
sleep 60
done
echo "$FILE now exists"
```

This tests every 60 seconds until the filename represented by the variable `$FILE` exists. A message is then displayed.

3. To watch for someone to logging in:

```
# make sure we pick up the correct commands
PATH=/bin:/usr/bin
# remember $# is number of positional arguments
case $# in
   1) ;;
   *) echo 'usage: watchfor username' ; exit 1
esac
until who | grep -s "$1" /dev/null
do
   sleep 60
done
echo "$1 has logged in"
```

If more than one username is given to the command **watchfor** the message

```
usage: watchfor username
```

is displayed and the command fails.

The break and continue statements

It is often necessary to handle exception conditions within loops. The statements `break` and `continue` are used for this.

The `break` command terminates the execution of the innermost enclosing loop, causing execution to resume after the nearest done statement.

To exit from *n* levels, use the command:

```
break n
```

This will cause execution to resume after the done *n* levels up.

The `continue` command causes execution to resume at the `while`, `until` or `for` statement which begins the loop containing the `continue` command.

You can also specify an argument *n* to `continue` which will cause execution to continue at the *n*th enclosing loop up.

Example of using the break *and* continue *statements*

1. To prompt for and run commands:

```
while echo "Please enter command"
    read response
do
    case "$response" in
        'done') break # no more commands
        ;;
        "") continue # null command
        ;;
        *) eval $response # do the command
        ;;
    esac
done
```

This prompts the user to enter a command. While they enter a command or null string the script continues to run. To stop the command the user enters **done** at the prompt.

More shell functions

This section provides you with information about several functions which can be used within a Bourne shell script.

Doing arithmetic

The shell does not have any arithmetic features built in and so you have to use the **expr** command. Details are in the `expr(1)` manual page.

Including text in a shell script

Text can be included in the shell script by using a here document, a special form of input redirection.

The << symbol is used to indicate that text should be read up to a given mark. For example:

```
# this script outputs the given text before it runs
cat << EOF
This shell script is currently under development,
please report any problems to Danny (danny@ed)
EOF
exec /usr/local/test/bin/test_version
```

The text is read from the script until a pattern is found which matches that after the << symbol; execution then proceeds as normal.

Forcing evaluation of commands

Another built-in function is `eval` which takes the arguments on the command line and executes then as a command. For example:

```
echo "enter a command:"
read command
eval $command
```

Execute a command without creating a new process

The `exec` statement causes the command specified as its argument to be executed in place of the current shell, without creating a new process. For example:

```
$ exec zmail -visual
```

This runs just the `zmail` program without a shell. When you quit the application the current shell also exits.

Controlling when to exit a shell script

The `exit` statement will exit the current shell script. It can be given a numeric argument which is the script's exit status. If omitted the exit status of the last run command is used. `0` (zero) signifies success, non-zero signifies failure. For example:

```
if [ $# -ne 2 ]
# "$#" is number of parameters- here we test
# whether it is not equal to two
then
echo "Usage $0 \<file1\> \<file2\>"
#not two parameters so print message
# and fail ($0 is name of command).
exit 2
fi
<rest of script>
```

This script is supposed to take two positional arguments. It will exit with status `2` (error) rather than `0` (success) if it is not called with two parameters.

Trapping operating system signals

Shell procedures may use the **`trap`** command to catch or ignore UNIX operating system software signals. The form of the trap command is:

```
trap 'command-list' signal-list
```

Signals to be caught

The following are the signals that are usually caught with the trap command.

0	shell exit (for any reason, including EOF)
1	hang-up
2	interrupt (^C)
3	quit (^\) causes program to produce a core dump
9	kill (cannot be caught or ignored
15	terminate; default signal generated by kill

Several traps may be in effect at the same time. If multiple signals are received simultaneously, they are serviced in ascending order.

To check what traps are currently set use the `trap` command. For example:

```
trap
```

The command list is placed between single quotes, as the command line is scanned twice, once when the shell first encounters the trap command and again when it is being executed.

The single quotes inhibit immediate command and variable substitution but are stripped off after the first scan, so that the commands are processed when the command is actually executed.

If *command-list* is not specified, then the action taken on receipt of any signal in the signal-list is reset to the default system action.

If *command-list* is an explicitly quoted null command (`' '` or `" "`), then the signals in signal-list are ignored by the shell.

The *command-list* is treated like a subroutine call. The commands in the list are executed when the signal is trapped and control is then returned to the place at which it was interrupted.

Examples of interrupt handling

1. To use single quotes to inhibit command substitution:

```
trap 'echo `pwd` >>$HOME/errdir' 2 3 15
for i in /bin /usr/bin /usr/any/bin
do
  cd $i
  some series of commands in the directory $i
done
```

The file `errdir` will contain the name of the directory being worked on when the procedure is interrupted. What happens if the same procedure has double quotes around it?

```
trap "echo `pwd` >errdir" 2 3 15
```

The file `errdir` will just contain the name of the directory from which the procedure was invoked because the **pwd** command would be substituted on the first scan by the shell and not when it is invoked in the script.

2. To remove temporary files when a procedure is interrupted.

```
temp=/tmp/file.$$
trap 'rm $temp; exit' 0 1 2 3 15
ls > $temp
 .....
```

If any of the named signals are encountered, the command

rm $temp; exit

will be executed. The **exit** command is needed to terminate the execution of the whole procedure.

3. To continue processing commands after a **trap** command;

```
# read and process commands
 dir=`pwd`
 for i in *
 do
   if test -d $dir/$i
   then
       cd $dir/$i
       while echo ''$i:''
          trap exit 2 # trap ^C
          read x
       do
          trap ' ' 2 #ignore interrupts
          eval $x
       done
   fi
 done
```

The shell continues to process commands after a `trap` command. The entire procedure is terminated if interrupted when waiting for input, but the interrupt is ignored while executing a command. The `command list` is an explicitly quoted null command and so the signal is ignored by the shell.

Debugging shell scripts

To see where a script produces an error use the command:

```
$ sh -x script argument
```

The `-x` option to the `sh` command tells it to print commands and their arguments as they are executed.

You can then see what stage of the script has been reached when an error occurs.

Debugging options

You can use the following options either on the line or with the built-in `set` command within a script to help you when debugging.

-e	in non-interactive mode, exit immediately if a command fails.
-v	print shell input lines as they are read.
-n	read commands but do not execute them.

Example of debugging shell script

1. To print commands and their arguments as they are executed:

```
$ cat example
TEST1=result1
TEST2=result2
if [ $TEST1 = "result2" ]
then
echo $TEST1
fi
   if [ $TEST1 = "result1" ]
then
echo $TEST1
fi
```

```
if [ $test3 = "whosit" ]
then
echo fail here\'cos it\'s wrong
fi
```

This is a script called `example` which has an error in it; the variable `$test` is not set so the 3rd and last `test` command will fail.

Running the script produces:

```
$ example
 result1
 [: argument expected
```

The script fails and to see where the error occurred you would use the `-x` option like this:

```
$ sh -x example
 TEST1=result1
 TEST2=result2
 + [ result1 = result2 ]
 + [ result1 = result1 ]
 + echo result1
 result1
 + [ = whosit ]
 example: [: argument expected
```

The error occurs in the command `[ = whosit ]` which is wrong as the variable `$test3` has not been set. You can now see where to fix it.

Getting further information

You can get more information from the manual page for the Bourne shell (sh).

To get information about shell programming with another shell read the manual page for that shell. For example:

```
man ksh
```

This will display the manual page for the Korn shell.

There are a large number of books on the different shells and how to program them. Many will be stocked in the computer section of most major booksellers, and they may be available through your library.

G

Glossary

GLOSSARY

access permissions

a set of permissions associated with every file and directory that determine who can read it, write to it, or execute it. Only the owner of the file (or the super-user) can change these permissions.

argument

a mathematical term that refers to a number, character or word which changes the action of the command it is applied to.

ASCII (American Standard Code for Information Interchange)

a standard code used for representing information on computer systems and networks. It uses the printing and non-printing characters that can be generated by the keyboard. Since 7 bits are available to represent each character it is possible to represent a total of 128 different characters.

A table of the ASCII character set can be found in the file `/usr/pub/ascii` on every UNIX system.

binary file

a file that contains codes which are not part of the ASCII character set. A binary file can contain any type of information that can be represented by an 8 bit byte - a possible 256 values.

bit

a binary digit of information that can have either of two values, 1 or 0 (zero).

blind carbon copy

a copy of a message which all persons on the `Bcc:` list receive. Recipients cannot see who else has received the message: hence the term "blind".

boot

this term derives from 'bootstrap loader', a short program that was read in from cards or paper tape, or toggled in from the front panel switches. This program was always very short (great efforts were expended on making it short in order to minimise the labour and chance of error involved in toggling it in), but was just smart enough to read in a slightly more complex program (usually from a card or paper tape reader), to which it handed control; this program in turn was smart enough to read the application or operating system from a magnetic tape drive or disk drive. Thus, in successive steps, the computer 'pulled itself up by its bootstraps' to a useful operating state.

buffer

a temporary area of memory which data can be read from and written to.

byte

a unit of memory or data consisting of 8 bits which is used to represent one character.

carbon copy

a copy of a message which all persons on the Cc: list receive. Recipients of the message can see who else it has been circulated to.

CPU (Central Processing Unit)

a logic chip that executes instructions passed to it.

current directory

the directory you are currently "in". Its value is displayed with the pwd command. It is synonymous with the term *current working directory.*

daemon
a background process (pronounced "demon") that carries out tasks on behalf of every user. Daemons spend most of their time sleeping until something comes along which requires their help. A UNIX system has a lot of daemons.

database concepts
a *database* is the name given to the combination of a program that can manipulate data, together with the data that it acts upon.

a *record* is a set of data about a person or item. Name, telephone number and postcode is an example of such a set.

a *field* is one part of that record; the postcode for example.

a *delimiter* character separates each field from the field preceding it and the field that follows it. A colon (:) is often used as a field delimiter.

a *key* is the field or part of a field that the program uses when it acts on the data.

default
a value to which some variable is automatically set, unless an alternative value is provided by the user.

domain
a part of a hierarchical naming scheme.

environment variable
a variable that defines an aspect of your working environment such as your home directory, editor or printer. These variables are set during the login procedure.

file-creation mode mask
a number consisting of three octal digits which is subtracted from a similar number used by the system to create files and directories.

file system
the physical or logical device that holds a collection of files and directories. This might be a hard disk drive or a partition on a disk drive.

full pathname
the name of a directory or file in relation to the root(/) directory.

group
a collection of users who have protected access to their resources.

home directory
the directory you are placed in immediately after you login to the system.

host
a computer system that provides a working environment to many users. So a “UNIX host” is such a system that runs the UNIX operating system.

inode number
a unique number associated with each filename. This number is used to look up an entry in the inode table which gives information on the type, size, and location of the file and the userid of the owner of the file.

link
a connection between an inode number and the filename(s) associated with it.

local host
the host on the network which you are currently using.

mail header
information contained in every mail message which is used by mail programs to provide the user with a summary of the origin and contents of each message.

metacharacter
a single character that is used in place of another character or set of characters.

multitasking
the execution of two or more processes at the same time.

parent directory
the directory that is one level above the current directory.

password
a security device consisting of a string of characters known only to the user and the system. The user must supply this string when prompted to gain access to the system.

pathname
a file name given as the sequence of directories that lead to the file. A pathname can be either a full pathname or a relative pathname.

prompt
(See "*shell prompt*" on page 6 of this section.)

regular expression
a text pattern consisting of a combination of alphanumeric characters and special characters known as metacharacters.

relative pathname
the name of a directory or file given in relation to your current working directory.

remote host
any host on the network which you are not presently using.

root directory
the directory located at the top of the UNIX file system. It is represented by the "/" (forward slash) character.

shell

a utility program that enables the user to interact with the UNIX operating system. Commands entered by the user are passed by the shell to the operating system which carries them out. The results are then passed back by the shell and displayed on the user's display.

There are several shells available. The user may select which one they wish to use.

shell prompt

a character at the start of the command line which indicates that the shell is ready to receive your commands. The character is usually a '%' (percent sign) or a $ (dollar sign). It may be different on your system.

standard input

the source of information for a command. This is assumed to be the keyboard unless input is redirected from a file or another command.

standard output

the destination for information from a command. This is assumed to be the terminal display unless output is redirected to a file or another command.

syntax

rules governing the way in which characters and words must be put together to form a command that can be recognised and acted upon by the UNIX operating system.

system administrator

a person or persons who has responsibility for managing the system(s) you are using.

text file

a file that contains only ASCII characters, allowing it to be edited or used with programs that do not handle binary files, such as electronic mail.

username

a unique “name” by which each user is known to the system. This name is given to the user whenever they register to use the system.

Index

Symbols

A

B

C

E

G

H

J

N

O

P

T

U