Part Number 800-1087-01
Revision: C of 1 November 1983
For: Sun System Release 1.0

Beginner's Guide to the

Sun Workstation

Sun Microsystems, Inc.,
2550 Garcia Avenue
Mountain View
California 94043
(415) 960-1300

Credits

Material in this *Beginner's Guide to the Sun Workstation* comes from a number of sources: *Introducing the UNIX System*, Henry McGilton, Rachel Morgan, McGraw-Hill; *UNIX For Beginners*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *An Introduction to the C Shell*, William Joy, University of California, Berkeley; *An Introduction to the Bourne Shell*, S. R. Bourne, Bell Laboratories, Murray Hill, New Jersey; *Mail Reference Manual*, Kurt Shoens, revised by Craig Leres; *How to Read the Network News*, Mark R. Horton, Bell Telephone Laboratories, Columbus, Ohio; and *A Dial-Up Network of the UNIX Systems*, D. A. Nowitz and M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey. These materials are gratefully acknowledged.

Trademarks

Ethernet is a trademark of Xerox Corporation.

Sun Workstation is a trademark of Sun Microsystems Incorporated.

UNIX is a trademark of Bell Laboratories.

Revision History

Revision	Date	Comments
A	15 May 1983	First release of the Tutorial for Beginners.
B	15 September 1983	Updated and reorganized.
C	1 November 1983	Minor corrections.

Table of Contents

Preface

Welcome to the Sun Workstation. This manual provides a tutorial introduction to help you get used to the main ideas of the system and to begin to make use of them quickly and efficiently. We assume that you are a first-time Sun system user, but that you already know something about an operating system, a terminal keyboard, and a text editor. We provide explanations and exercises for learning to use the Sun system and Sun Workstation in particular, however, most of the information is also applicable to using UNIX on ASCII terminals.

We introduce many of the new ideas and terms to you in this Preface to give you an idea of what the terminology is like. Explanations are provided in the following chapters.

The Sun Workstation is a desktop graphics computer designed to support a wide range of engineering, scientific, and CAD/CAM applications. Sun software is based on the UNIX system developed at the University of California at Berkeley (4.2bsd) and supports some of the most powerful graphics and user-interface software packages available. It is compatible with other versions of UNIX. The Sun system features include efficient networking, support for virtual memory through demand-paging, a high-performance file system, and an advanced graphical user interface.

This powerful graphics computer supports up to four megabytes of main memory, various disks and tapes, and local area networking. It runs the Sun operating system, the basic resident code on which everything else depends. It is an advanced version of the UNIX operating system originally developed at Bell Laboratories. The operating system supports the system calls and maintains the file system.

The Sun system capabilities include re-entrant code for user processes; 'group' access permissions for cooperative projects with overlapping memberships; alarm-clock timeouts; timer-interrupt sampling and interprocess monitoring for debugging and measurement; and multiplexed I/O for machine-to-machine communication. It also provides powerful network protocols for local nets.

The Sun system supports most of the available UNIX programs. The commands are self-contained and do not require extra setup. You can run interactive programs, that is, ones that prompt you for information, from a prepared script or file simply by redirecting input. Most programs intended for interactive use, the text editors for example, provide escapes to the command level interpreter called the Shell. Most file processing commands can also go from standard input to standard output through filters. You can use the piping facility of the Shell to connect such filters directly to the input or output of other programs.

The Sun system provides the following classes of utility programs:

1. File Manipulation
2. The Shell
3. Programming Languages, such as C, Fortran77, and Pascal.
4. Text Editing and Document Processing, such as the *vi* editor and *troff* for phototypesetting documents.
5. Information handling, networking, graphics, and games.
6. System Management

See the *Sun System Summary* for an overview.

If you already know something about UNIX, much of this material will look familiar, but its organization may be somewhat foreign. This format provides exercises for learning to use the system and does not explain the internals of the commands and programs. For more in-depth information on the Sun system and the UNIX operating system in general, there is a plethora of

Sun documentation and literature, which is listed in the appendix. We refer to these manuals from time to time in this tutorial to draw you a roadmap of where to go from here with your education.

Part One of the *Beginner's Guide to the Sun Workstation* describes the basic tools you need to learn to use the Sun system. Part Two provides information on how to use the Shells, the *mail* facility, and the network news. Also included is a glossary and an annotated bibliography.

The chapters in Part One are:

1. Getting Started — Explains how to log in, the Sun system file directory structure, how to use some basic system commands, and how to log out.
2. Working With Files — Describes commands for manipulating your files.
3. Using The Shell — Instructs you how to use the Shell commands to make your work easier.
4. Creating and Editing Text Files — The 'vi' Editor — Provides instructions on the most useful editor commands.
5. Printing and Formatting Documents — Explains how to use the most useful text formatters and macro packages and how to display and print document copies.
6. Communications — Introduces the communication facilities and describes how to use the mail system, how to talk to users on other systems, and how to use your local network.
7. Sun System Summary — Look here for a nutshell version of Sun hardware and software features.

You should have a couple of other documents with you for easy reference as you begin. The most important is the *User's Manual for the Sun Workstation;* it's often easier to tell you to read about something in the user's manual than to repeat its contents here. The user's manual also lists all the manuals supplied with the Sun Workstation. The other useful document is the *Editing and Text Processing on the Sun Workstation,* which explains how to use the text formatters and the *vi* editor.

PART ONE — INTRODUCTION TO THE SUN SYSTEM

1. GETTING STARTED

Sit down at your Sun Workstation. To use this tutorial, do the exercises and then experiment with the suggested options. Look up the commands in the user's manual to get an idea of their additional capabilities. If you don't get a desired response, poke around with the available options and commands until you do. Don't be afraid to sample other commands described in the user's manual; experimenting will quickly teach you how to work your way around the system.

1.1. Logging In

Examine the workstation screen. Since every system has a name, your screen displays something like

```
tutorial login:
```

where 'tutorial' is the system's name or *hostname.* Your installation may have a naming theme, such as naming all systems after planetary bodies, but for our purposes, let's use the hostname 'tutorial.' It's important to remember your system's hostname and the names of your colleagues' so you can communicate with them and share files over the network.

To begin, you also need a login name (also known as an *account),* which your system administrator sets up for you along with your system. To set up your own account or add a user, refer to the *adduser* command in the *System Manager's Manual for the Sun Workstation*.

Be aware that the Sun system has a strong orientation towards *lower-case* characters and makes a definite distinction between upper and lower case. Type your name in lower case if possible — if you type in upper case, the Sun system assumes your terminal can't talk lower case and so talks at you in upper case from here on. Sun system commands are almost always in lower case. If we begin a sentence with a command name in this manual, the first letter is capitalized as a courtesy to English.

After you have typed your login name, and pressed the RETURN key, the system prompts you for a password if your account is set up with one. For example, your screen looks like:

```
tutorial login: evan
Password:
```

when it is waiting for your password. Type your password and press RETURN. The system doesn't display or 'echo' your password, nor does the cursor even move as you type. This keeps your password secret from anyone who is looking over your shoulder. If you don't have a password, typing your login name and pressing RETURN gets you straight onto the system. If you make a mistake typing in your login name, press the DEL key to back up over your mistake, and retype the correct characters. You can also simply press the RETURN key several times until you see the 'tutorial login:' prompt again and start over. Sometimes you may mistype your password, and see the same message as if you had mistyped your login name:

```
tutorial login: evan
Password:
Login incorrect
tutorial login:
```

Simply retype your login name and password to log in.

You know you've logged on successfully when the system displays some messages and then displays a ready prompt. The display is either:

```
tutorial login: evan
Password:
Last login: Mon Jul 18 07:50:22 on ttyp0
Sun UNIX 4.2 UNIX (Berkeley beta release) (GENERIC) #8: Wed Oct 23 13:45:52 PDT 1983
tutorial%
```

or

```
tutorial login: evan
Password:
Last login: Mon Jul 18 07:50:22 on ttyp0
Sun UNIX 4.2 UNIX (Berkeley beta release) (GENERIC) #8: Wed Oct 23 13:45:52 PDT 1983
$
```

The '%' and '$' prompts indicate that the UNIX command interpreter, the *Shell*, is waiting for you to type commands at it. There are two versions of the Shell, the C-Shell and the Bourne Shell. Which character you get as a prompt (% or $) depends on which Shell is listening to you. We use the C-Shell, whose prompt is the '%' sign in all our examples, but refer to it simply as the 'Shell' for simplicity's sake. The Bourne Shell's prompt is the '$' sign. The Shells are described in the chapter *Using the Shell* and in more detail in Part Two of this manual. You can also refer to the *User's Manual for the Sun Workstation* pages on *csh* for the C-Shell and on *sh* for the Bourne Shell.

Now try the following; at the 'tutorial%' prompt, type:

```
tutorial% echo Hello there.
Hello there.
tutorial%
```

Don't forget to press RETURN (also called a carriage-return) after the command, or nothing will happen. If you think you're being ignored, press RETURN, and something should happen. We won't mention this again, but don't forget to press RETURN at the end of each line.

The first word you typed to the 'tutorial%' prompt on the *command line* is the Sun system command *echo*. *Echo* does what it says, it 'echoes' or displays whatever follows it. When you type something to the 'tutorial%' prompt, the first word is always a command, which is also called a *program*. At this point, you are asking the Shell to look for that command and execute it.

Now substitute your name in place of *yourname* in the example below:

```
tutorial% echo Hello yourname
Hello . . .
tutorial%
```

Always separate a command from whatever follows by a space. Now try:

```
tutorial% echo This is fun.
This is fun.
tutorial%
```

These are very simple examples. What follows a command is an *argument*. Often this argument is the name of a file or *filename*, but in the simple example above, it's merely some text.

If you want to compile a program written in the C programming language, for instance, you type:

```
tutorial% cc data.c
tutorial%
```

Here the command is *cc*, the C compiler program. The argument to *cc* is a file called *data.c*, containing the program's source text.

You can name a file almost anything you want, but limit yourself to alphanumeric characters and the period '.'. Other characters such as the asterisk (*), slash (/), and question mark (?) have special meanings when the Shell reads them. These special characters called *metacharacters* are described later.

There is one more thing that you can type on the command line after the 'tutorial%' prompt. This is called an *option* or *flag argument*. Type it after the command, but separate the two by a space. Flag arguments modify the command. For example, if you want to suppress the load phase of the compilation of your program *data.c* and produce an object file, you type the *cc* command with the *-c* option:

```
tutorial% cc -c data.c
tutorial%
```

Sun system commands usually have several options.

Here's a good opportunity to open up the *User's Manual for the Sun Workstation* to learn more about commands and about the user's manual itself. The user's manual contains information on most of the commands used in this tutorial. Turn to the *cc* pages and glance through them. You first see NAME, which gives you the command name as you type it at the Shell prompt 'tutorial%' and a brief phrase describing what the command does. Here, you see that *cc* is the 'C compiler.' Next you see the SYNOPSIS, which shows the command line format and all the options. Yes, there are plenty of options for the C compiler. Most commands do not have this many however.

Following is the DESCRIPTION, which explains in more detail what the command does. OPTIONS lists and describes the flag arguments. The remaining entries, EXAMPLE, FILES, SEE ALSO, DIAGNOSTICS and BUGS provide additional information.

If you're interested in the wealth of information that the user's manual provides, turn to the front of the manual and read on about the Sun system manuals in general to become familiar with the other reference sources.

Note that important terms and command names in the text of this tutorial are printed in italics, *cc* for example, and options in bold, like **-c**. In the exercises, **bold face type like this** indicates what you should type at your workstation.

In general, when you type characters to the Sun system, they are gathered up until you type RETURN or a newline (also called LINEFEED). Up to the point you type RETURN, you have the opportunity to correct typing mistakes — you can back up over characters or words, or you can kill the entire line typed so far and start over. Once you have typed a RETURN, though, the line is passed on to whatever program you asked for (or maybe none at all if you misspelled the program's name).

If you make a mistake typing the command name, and refer to a non-existent command, you will be told. For example, type 'whom':

```
tutorial% whom
whom: not found
tutorial%
```

Clearly, 'whom' is not a command. Of course, if you inadvertently type the name of some other command, it will run with more or less mysterious results.

The Sun system handles the keyboard and screen in *full-duplex* — it has full read-ahead, meaning that you can continue typing even while the system is displaying output on the screen. Of course, this means that what you type in as input is all mixed up with what the system displays — this may or may not bother you. However, what you type is saved up and interpreted in correct sequence.

Try one more command; type the *who am i* command:

```
tutorial% who am i
tutorial!evan  console  Jun 28  14:19
tutorial%
```

The first name, 'tutorial,' is the system hostname; the second is the user's login name, 'evan' in this case.

Briefly, what's happening when you run a program on the Sun system by typing the command name and pressing RETURN is this: the executing program takes input or *data* and produces output or *results*. A program usually expects to read the input from your keyboard, which is called the *standard input*, and the program usually writes its output to your workstation called the *standard output*. You can change these standards by *redirecting* the input and output to come from and go to other places, such as files, line printers, and so. This is explained in *Using the Shell*.

1.2. What to Do If Something Goes Wrong

Sometimes your system may not respond correctly. Here are two brief sections on what to do.

1.2.1. Special Keys and Control Characters

If you make a typing mistake, and see it before you press RETURN, there are several ways to recover using the following keys and control characters. You can also use some of these characters to start and terminate programs. Glance over the list now to become familiar with them.

DEL Called the *erase* character, DEL↓ backs up over and *erases* the previously typed *character*. Successive uses of DEL erase characters back to the beginning of the line, but not beyond.

^U Called the *kill* character, ^U erases the entire *line* you just typed.* If the line is fouled up, type a ^U and start over.

^W This wipes out the previous *word* you typed. (A *word* is a sequence of characters delimited by space(s) and/or tab(s).)

^C ^C aborts or *interrupts* a currently running program. Use ^C to stop a long printout, for example. (Programs, like editors, can either ignore ^C altogether or be notified when it is typed instead of being terminated.)

^S This *stops* output from a running program. It is useful for preventing text being displayed from zipping off your workstation screen.

^Q ^Q resumes output from a program whose display was suspended with ^S.

^O This throws output away without interrupting the program.

↓ On older keyboards, use the BACKTAB key as the DEL key.

^\ The Quit character *quits* a program and saves an image of that program in a file called *core*. This is mostly used by programmers debugging programs.

Tab Tabs are used freely in the Sun system source programs. If your terminal does not have the tab function, use the *stty* command (described in the user's manual) to turn tab characters into spaces during output, and to be echoed as spaces during input. Tabs are set every eight columns.

You can find out what the control characters are anytime with the *stty* command:

```
tutorial% stty all
new tty, speed 9600 baud; tabs
crt
erase  kill  werase  rprnt  flush  lnext   susp  intr  quit  stop  eof
^H     ^U    ^W      ^R     ^O     ^V     ^Z/^Y  ^C   ^\    ^S/^Q  ^D
tutorial%
```

Some of these, like ^Z, are explained later on.

1.2.2. Things That Go Bump in the Night

Sometimes you can get into a state where your workstation or terminal acts strangely. For example, you may not be able to move the cursor, your cursor may disappear, there is no echoing of what you type, or typing RETURN may not cause a linefeed or return the cursor to the left margin. Try the following solutions:

- First, type ^Q to resume possibly suspended output. (You might have typed ^S, freezing the screen.)
- Another possibility is that you accidentally typed a NO SCRL key (also called SET UP/NO SCROLL on some terminals) on your keyboard. This also freezes the keyboard like typing a ^S. Try typing ^Q, which toggles you back to proper operation if you did indeed type the NO SCRL key in the first place.
- Try typing ^C to interrupt the currently running program.
- Next, try pressing the LINEFEED key, followed by typing 'reset', and pressing LINEFEED again.
- If that doesn't help, try logging out and logging back in (see *Logging Out*). If you are using a terminal, try powering it off and on to regain normal operation.

1.3. Changing Your Password — the 'passwd' Command

Passwd is the command that changes your password or installs one for you if you don't already have one. *Passwd* is interactive so that when you type *passwd*, it prompts you for input as follows:

```
tutorial% passwd
Changing password for evan
Old password: xxxxxx
New password: zzzzzz
Retype new password: zzzzzz
tutorial%
```

The system doesn't echo what you type (shown by the x's and z's above), but it does ask you to

•We use the convention "^ *whatever*" to mean control-whatever — that is, hold down the CONTROL (or CTRL) key while typing a *whatever* character. "^D' means hold down the CONTROL key while typing 'd'.

type your new password twice to prevent typographical accidents and to provide better security. You will have to provide a reasonably long password unless you are persistent.

1.4. Logging Out

When you have finished your login session, there are several ways to log out. One way is to use the *logout* command:

```
tutorial% logout
tutorial login:
```

You can type an end-of-file indication, ^D (the EOF character). Your system responds:

```
tutorial% (^D) logout
tutorial login:
```

Finally, you can also change users directly with the *login* command and another login name:

```
tutorial% login jerry
Password:
Last login: Mon Jul 18 07:50:22 on ttyp0
Sun UNIX 4.2 UNIX (Berkeley beta release) (GENERIC) #8: Wed Oct 23 13:45:52 PDT 1983
tutorial%
```

Try these logout methods now.

Note that if you are using a terminal, it is usually not sufficient just to turn off the terminal to log out. Most Sun systems do not use a time-out mechanism, so you'll still be logged in unless you logout explicitly.

1.5. What is the Shell?

As discussed earlier, after you log in and the 'tutorial%' prompt appears, a utility program called the *Shell* listens to what you type. In general, what you type to the Shell is a *command line*. A command always consists of the *command name*, such as the *cc*, which compiles C programs. This is always the first thing on the line. This command name can be followed by optional *arguments*, such as the –c option or a filename. A command's arguments are separated from the command, and from each other, by spaces and/or tabs. The Shell searches for the command in several well-known places, starts up the command, and passes the arguments on to it. When the command has completed its job, the Shell regains control and again listens to what you type.

Most commands are simply executable programs that the Shell looks for, but some commands are 'built-in' to the Shell, and the Shell interprets such commands directly. The Shell has many other capabilities, which are detailed in the user's manual in the *csh* (C-Shell) and *sh* (Shell) entries. These are the two main Shells that you can run.

The Shells are different for all but the most simple terminal usage. For instance, the C-Shell has *history* and *alias* features, which greatly enhance its power when used interactively. The C-Shell also supports a set of job-control facilities. See the reference material on Shells in Part Two for more detailed information.

1.5.1. Login Profile

When you log in to the Sun system, you are logging in to the C-Shell. The Shell looks for files called *.login* and *.cshrc*. If there are such files in your *home directory*, (described later) it runs any commands it finds in them.

You only execute the *.login* file when the Shell is called up as part of the process of logging in. A sample *.login* file is:

```
setenv EXINIT 'set noai wrapmargin=8'
set path=(. / /bin /usr/bin /usr/local /usr/ucb /usr/hosts)
set mail=(/usr/spool/mail/$USER)
```

We won't explain now what all these entries mean; see *Using the C-Shell* in Part Two for details.

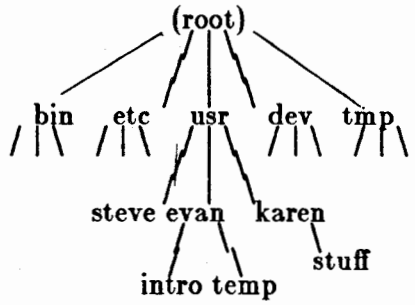
The file *.cshrc* is executed any time the Shell is invoked; for example, when you fork a new Shell. A sample *.cshrc* file is:

```
if (! $?prompt) exit
set history=36
alias lf ls -F
alias cursor echo '^[[s'
alias pq 'lpq -P'
alias tutorial cd supplements/tutorial/tut
```

The *history* and *alias* entries are described later. Again, see *Using the C-Shell* for more details.

1.6. The File System

Sun system files are arranged in a hierarchy of directories. This hierarchy resembles an inverted tree structure: the file system begins at the root directory and branches to the limits of the available mass storage. *Root* is the name of the directory at the 'beginning' of the file system; this is called 'slash.' *root* contains references to files and subdirectories which contain other files and subdirectories and so on. Each directory is thus like a node on the tree — directories contain files (which contain data) and subdirectories (which contain files). Here is a picture which makes this clearer:



You can either move to another directory to work on the files held there, or you can gain access to those files from where you are, but you need to know *where* you are and what the direction or *pathname* is to the directory you want. If you are in */usr* for example, and want to work on a file in *evan*, then you need specify only the *relative pathname* as the directories are on the same branch. Look at this as a sort of 'how to get there from here' situation. If you are in *usr* and want to gain access to something in */bin*, you need to indicate the *absolute* or *full pathname*, which specifies 'how to get there from the system root.'

You can always find out where you are in this structure. Use the *pwd* (print working directory) command and type:

```
tutorial% pwd
/usr/evan/intro
tutorial%
```

to get your *current* or *working* directory. Here we are in */usr/evan/intro*. There is a special notation that indicates, among other things, the current directory. This is the '.' or 'dot'. Two dots '..' means the *parent* directory, that is, the directory that the current directory ('.') is a subdirectory of. This convention is constant wherever you are in the directory hierarchy.

The initial slash '/' in the response to the *pwd* command above names *root*, and successive slashes separate directory names. Here the current working directory is *intro*, which is a subdirectory of the directory *evan*, which is in *usr*, which is in '/'. The *pwd* command displays the full pathname to your directory.

Your system administrator sets up your account by creating a directory, normally using the same name as your login name. This is called your *home* directory. Everytime you log in, your working directory will be set to your home directory. You *own* your home directory. This means you have full permission to read, write, or destroy its contents. You can also create new files and subdirectories within your home directory as needed, and do with them as you please. Access to files that others own is carefully controlled.

There are two kinds of directories, *system* directories and *user* directories. System directories contain files and subdirectories that apply to the whole system. You and your fellow users can make user directories as needed as we show below. We describe how to do this later. For example, earlier you saw a user directory for Evan, */usr/evan*. There can be lots of user directories; it depends on how big the system is and on how many people can use it. You can have */usr/henry* for user Henry, and */usr/gang*, for a particular project's user directory, for example.

There are also system directories, such as */bin*, which holds most of the executable system commands, */etc* which contains system directories, and */usr*, which contains other directories. Your login name, encrypted password, and other information are contained in the *passwd* file in the */etc* system directory for instance.

That's enough detail — let's move on to how you can use files and directories.

1.8.1. Changing to Other Directories with 'cd'

To look at other directories, use the *cd* (change working directory) command. If you're in your home directory now at */usr/evan*, and you want to change to the */etc* directory, type:

```
tutorial% cd /etc
tutorial%
```

You've moved to */etc*. If you then check your working directory with *pwd*, you'll see:

```
tutorial% pwd
/etc
tutorial%
```

A *cd* command without any argument always returns you to your home directory. So if you get lost, type *cd* to get home:

```
tutorial% cd
tutorial% pwd
/usr/evan
tutorial%
```

And to change to the directory directly above the one you are currently in, type:

```
tutorial% cd ..
tutorial%
```

Try changing to some of the other directories noted above, such as */bin*.

1.6.2. What Files Do I Have — the 'ls' Command

You can see what a directory contains using the *ls* or 'list' command. See what files are contained in the root directory '/'. Type the *ls* command:

```
tutorial% ls /
bin             etc             mnt     sys       usr
boot            lib             pub     testfile  vmunix
dev             lost+found      stand   tmp
tutorial%
```

You see several columns of files and directories. Try changing to some of these directories and using *ls* to list the contents.

Now change back to your home directory and use *ls* to list files there.

```
tutorial% cd
tutorial% ls
tutorial%
```

You don't have any files in your home directory so *ls* doesn't list anything. There are, however, some 'hidden' files, which the *ls* command with the **-al** option reveals. First try:

```
tutorial% ls -al
total 14
drwxr-xr-x   2 evan   32    Mar 12 20:31 .
drwxr-xr-x  12 root  240    Jul  7 15:22 ..
tutorial%
```

The **-a** option lists 'all' files. The **-l** asks for a 'long' listing, for without it, only the filenames and subdirectory names are listed. Details on what all the parts of the **-l** listing mean are provided later. You'll also notice here that you can combine flags as a sort of shorthand.

Again, the '.' is your home directory, and the '..' your parent directory or */usr*.

There is another variant *ls -F*, which marks which files contain executable programs, which files are directories, and which files are symbolic links (link a file or directory to another file or directory). Try listing the contents of the *root* directory with the **-F** option:

```
tutorial% ls -F /
bin/            etc/            mnt/    sys/      usr/
boot*           lib/            pub/    testfile  vmunix@
dev/            lost+found      stand/  tmp/
tutorial%
```

An entry without any following mark is a simple file. Those entries marked with an asterisk

sign '*' are executable. Those marked with a slash character '/' are directories and contain more files or subdirectories. Those marked with an 'at' sign '@' are symbolic links.

1.6.3. Making Directories with 'mkdir'

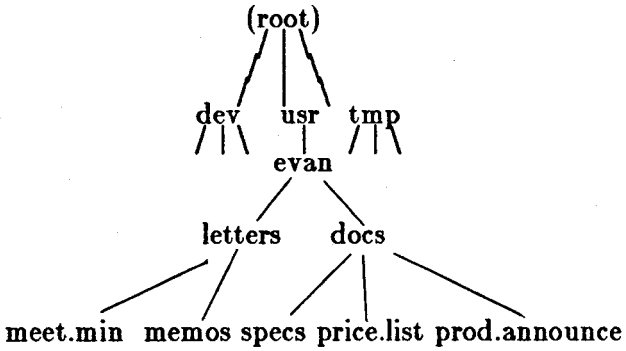
Let's assume you now want to create some subdirectories to hold documentation and related memos for a customer demonstration of your new product. These will be user directories in */usr/evan*. Return to your home directory and make sure you know where you are:

```
tutorial% cd
tutorial% pwd
/usr/evan
tutorial%
```

Use the *mkdir* command to create (or 'make') directories for this project:

```
tutorial% mkdir docs letters
tutorial% cd docs
tutorial% mkdir specs price.list prod.announce
tutorial% ls -l
total 3
drwxr-xr-x  2 evan        24 Aug 19 09:12 price.list
drwxr-xr-x  2 evan        24 Aug 19 09:12 prod.announce
drwxr-xr-x  2 evan        24 Aug 19 09:12 specs
tutorial% cd ../letters
tutorial% mkdir meet.min memos
tutorial% ls -l
total 2
drwxr-xr-x  2 evan        24 Aug 19 09:15 meet.min
drwxr-xr-x  2 evan        24 Aug 19 09:15 memos
tutorial% cd
tutorial%
```

The file system tree structure here looks like:



1.6.4. Removing Directories with 'rmdir'

Use *rmdir* to remove directories. The directory must not contain any files or subdirectories if it is to be removed. For example, try to remove the *letters* directory:

```
tutorial% rmdir letters
rmdir: letters: Directory not empty
tutorial%
```

What happened? The *letters* directory still has subdirectories in it. To remove it, you have to clear it out first with:

```
tutorial% rmdir letters/meet.min letters/memos letters
tutorial%
```

Rmdir also lets you know if you try to remove a non-existent directory:

```
tutorial% rmdir void
rmdir: void: No such file or directory
tutorial%
```

1.8.5. Copying Files with 'cp'

As you know from your programming experience, the easiest way to write a program is to start with a copy of another program and develop it from there. To make a copy of another file, use the *cp* (copy) command on the */etc/hosts.equiv* file:

```
tutorial% cp /etc/hosts.equiv  hosts.machines
tutorial%
```

This copies *hosts.equiv* file in the */etc* directory into a file called *hosts.machines* in your current working directory. Use this simple command to make as many copies of any file you want.

Check your current working directory with the *ls* command to see what it now contains.

```
tutorial% ls -l
total 6
drwxr-xr-x    5 evan      512 Aug 19 09:12 docs
-rw-r--r--    1 evan      480 Aug 19 09:23 hosts.machines
tutorial%
```

Remember the order of the *cp* command: *cp* copies the first file (or 'argument') to the second:

```
tutorial% cp this that
tutorial%
```

To copy the same named file into your own directory, mention the filename twice:

```
tutorial% cp /etc/hosts.equiv  hosts.equiv
tutorial%
```

This puts a copy of the file called *hosts.equiv* into the current directory.

If the second argument is an existing directory, you can use *cp* to copy the file named as the first argument into that directory and retain the same filename. It's a little faster too.

```
tutorial% cp /etc/hosts.equiv .
tutorial%
```

This copies */etc/hosts.equiv* into your current directory; the 'dot' (.) again stands for the current directory.

You can copy a file into another file in your current directory or into one of your subdirectories as follows:

```
tutorial% cp /etc/hosts.equiv docs
tutorial%
```

Or you can copy as many files as you need into a directory:

```
tutorial% cp docs/specs  docs/price.list  .
tutorial% ls –l
total
drwxr-xr-x 5 evan      512   Aug 19 09:28 docs
-rw-r--r--  1 evan      480   Aug 19 09:27 hosts.equiv
-rw-r--r--  1 evan      480   Aug 19 09:23 hosts.machines
-rwxr-xr-x 1 evan       24   Aug 19 09:29 price.list
-rwxr-xr-x 1 evan       24   Aug 19 09:29 specs
tutorial%
```

Several words of warning when using *cp*:

There *isn't* any warning if you try to copy a file to another that already exists. The existing file is written over, and you lose that version.

If a file is protected from being written on for security reasons, or if you try to copy into a non-existing directory, you see the message:

```
tutorial% cp notes intro/lessons
cp: cannot create intro/lessons
tutorial%
```

And if for security reasons, you do not have read permission on the source file, write permission to the directory, or if again, the source file or directory does not exist, you see:

```
tutorial% cp data/today tomorrow
cp: cannot open data/today
tutorial%
```

1.6.6. Moving and Renaming Files with 'mv'

To move files and directories from one place in the file system to another, use the *mv* (move) command. *Mv* renames a file. Moving differs from copying in that the original file disappears.

Try using *mv* as follows:

```
tutorial% ls –l

total
drwxr-xr-x    5 evan    512    Aug 19 09:28 docs
-rw-r--r--    1 evan    480    Aug 19 09:27 hosts.equiv
-rw-r--r--    1 evan    480    Aug 19 09:23 hosts.machines
-rwxr-xr-x    1 evan     24    Aug 19 09:29 price.list
-rwxr-xr-x    1 evan     24    Aug 19 09:29 specs

tutorial% mv hosts.equiv hosts
tutorial% ls –l

total
drwxr-xr-x    5 evan    512    Aug 19 09:28 docs
-rw-r--r--    1 evan    480    Aug 19 09:27 hosts
```

```
-rw-r--r--    1 evan    480    Aug 19 09:23 hosts.machines
-rwxr-xr-x    1 evan     24    Aug 19 09:29 price.list
-rwxr-xr-x    1 evan     24    Aug 19 09:29 specs
tutorial%
```

The file *hosts.equiv* has been renamed *hosts*. Note: If you move a file to a name that already exists, the second file contents are removed *without warning*.

```
tutorial% mv hosts hosts.machines
tutorial% ls -l

total
drwxr-xr-x    5 evan    512    Aug 19 09:28 docs
-rw-r--r--    1 evan    480    Aug 19 09:27 hosts.machines
-rwxr-xr-x    1 evan     24    Aug 19 09:29 specs
tutorial%
```

If the target file is write-protected, *mv* asks you if you really want to write over the file. If you respond with **y** (yes), the file is moved. Otherwise, nothing happens.

To move a file from one directory to another, make the second argument to *mv* the name of the target directory, if the target directory exists:

```
tutorial% ls -l
total
drwxr-xr-x    5 evan    512 Aug 19 09:28 docs
-rw-r--r--    1 evan    480 Aug 19 09:27 hosts.machines
-rwxr-xr-x    1 evan     24 Aug 19 09:29 specs
tutorial% mv hosts.machines docs/specs
tutorial% ls -l
total
drwxr-xr-x    5 evan    512 Aug 19 09:28 docs
-rwxr-xr-x    1 evan     24 Aug 19 09:29 specs
tutorial% ls -l docs/specs
total 1
-rw-r--r--    1 evan    480 Aug 19 09:27 hosts.machines
tutorial%
```

You can move as many files as you like.

You can also move an entire directory to another name, but unlike files, you can only move directories if the second name does not already exist. See the user's manual on *mv* for details on all the facilities.

You can't move or rename a file that doesn't exist. You'll see the message:

```
tutorial% mv illusion delusion
mv: cannot access illusion
tutorial%
```

1.6.7. Removing Files with the 'rm' Command

The *rm* command removes files from a directory. See what is in one of your directories, and remove a file of little importance:

```
tutorial% cd docs
tutorial% ls -l docs
total 3
-rw-r--r--      1 evan      480 Aug 19 09:28 hosts.equiv
drwxr-xr-x      2 evan       24 Aug 19 09:12 price.list
drwxr-xr-x      2 evan      512 Aug 19 09:45 specs
tutorial% rm hosts.equiv
tutorial% ls -l docs
total 2
drwxr-xr-x      2 evan       24 Aug 19 09:12 price.list
drwxr-xr-x      2 evan      512 Aug 19 09:45 specs
tutorial%
```

Again, like the *cp* command, you can work on more than one file at a time. And if any file is write-protected, *rm* asks you whether you really want to remove a file. Responding **y** tells *rm* 'yes,' and *rm* removes the file. If you don't have write permission on a directory, you cannot remove a file from it, even if you own the file and have write permission on it.

If you are worried about removing files that you really don't want to remove, use the **–i** option to *rm* to get an 'interactive' prompt on every file. There is also the **–f** option that 'forces' the *rm* command to remove a file, even if it is write-protected.

With **–r** (recursive) option, the *rm* command searches down the directory tree, removing all files it finds. When a subdirectory is empty, *rm* then removes that subdirectory. This command does this for every file in every subdirectory (and so on) that it finds in the specified directory.

Make copies of some of the files in */usr* or *etc* and try these options. Use the *ls -F* command to copy text files, not executable files or directories.

A word of warning: there are several special characters called *metacharacters* in the Sun system. We'll describe them in more detail later, but for now be careful about using the metacharacter '*' with *rm*, and especially with the **–r** recursive option.

```
tutorial% rm -r *
tutorial%
```

removes everything from the current directory on down as does the *rm -r .* command, so be careful!

Trying to remove a non-existent file results in:

```
tutorial% rm nobody.home
rm: nobody.home non-existent
tutorial%
```

1.7. Security

If you are worried about all the freedom that the Sun system provides to copy files, change directories, and such, rest assured that it also supplies a security system to control access to files and directories. For every file and directory, there are three classes of users who may have access, and for each of these classes, there are three 'permissions,' allowing or prohibiting access to a particular file.

The three classes are:

1. Owner — the person who created the file.
2. Group — the group of users who share ownership of a file. (This is set up in the */etc/passwd* file along with your login name and in the */etc/group* file.)
3. Public — all other system users.

For each of these classes, the three permissions are:

1. Read — allows reading the file.
2. Write — allows changes to the file.
3. Execute — allows listing files in a directory, and execution of programs and Shell scripts.

Consider some of the files and directories from before:

```
drwxr-xr-x   5 evan   512   Aug 19 09:28 docs
-rw-r--r--   1 evan   480   Aug 19 09:27 hosts.machines
-rwxr-xr-x   1 evan    24   Aug 19 09:29 specs
```

Here's how you interpret all that stuff at the beginning of the lines. Consider the *docs* directory:

```
drwxr-xr-x  5 evan      512 Aug 19 09:28 docs
```

The 'drwxr-xr-x' order shows 'user, group, public.' The first 'rwx' belongs to the user 'evan' who has read-write-execute permission. The second 'r-x' belongs to the 'group' which has read-execute permission for the file. The third 'r-x' belongs to the 'public' which also has read-execute permission. Hyphens indicate the absence of a permission. The leading 'd' indicates that it is a directory.

1.7.1. Changing File Permissions with 'chmod'

Suppose you need to change the permissions on a file or directory so others can access them to do work. The *chmod* command changes those permissions or the 'mode' of the file. There are four common modes that set the permissions. The following numbers are based on the octal number format. (Read more about *chmod* in the user's manual if there doesn't seem to be any rhyme or reason.)

- 644 indicates '-rw-r--r--'. The owner can read and write the file, but everyone else can only read it.
- 755 indicates 'rwxr-xr-x'. The owner can read, write, and execute the file, and everyone else can read and execute it.
- 600 indicates 'rw-------' for a file (use 700 for a directory). The owner can read and write the file, and everyone else has no access.
- 444 indicates 'r--r--r--'. Everybody can only read the file.

Suppose you have some directories like 'evan' above, and you decide that only you should be able to read and write the *docs* directory, while everyone else cannot do anything. As it is now, the permissions are:

```
tutorial% ls -l
drwxr-xr-x  5 evan      512 Aug 19 09:28 docs
tutorial%
```

Using the *chmod* command with the '700' permission, type:

```
tutorial% chmod 700 docs
tutorial% ls -l
drwx------  5 evan       512 Aug 19 09:28 docs
tutorial%
```

The '755' permission is the default.

You can't change the mode of a file that doesn't exist, nor can you change the mode of a file that you don't own (remember ownership is indicated by the login name, 'evan' in the example above).

1.8. Finding Out What Is Going On In the System

You now have a basic understanding of your personal directory. Let's take a step outside and see what the system can do for you.

1.8.1. Who Is Logged On — the 'who' Command

See who is currently logged in with the *who* command:

```
tutorial% who
evan    console Jan 16  09:11
smith   ttyp0   Jan 16  09:25
tutorial%
```

The first entry is the user's login name, the second entry is the system's idea of what terminal the user is on, and the date and time is when the user logged in.

1.8.2. Who Is Using the Network — the 'rwho' Command

Rwho produces output similar to the *who* command, but for all systems on your local network. (The 'r' in *rwho* may seem cryptic, but it refers to those 'remote' hostnames that make up the rest of your local network.) *Rwho* is somewhat more particular than *who* however, in that it only reports on machines that have been active during the previous 60 minutes. If a user has not typed to the system for a minute or more, *rwho* reports this idle time in minutes in the rightmost column. A typical example is:

```
tutorial% rwho
bob             krypton:ttyp2    Aug  2 14:19 :03
                < etc. >
joe             venus:console    Aug  2 08:57 :27
marty           paper:ttyp0      Aug  2 14:34
tutorial%
```

Try the *rwho* command now to see who else is logged in on your network.

Use the **-a** option to show everyone logged in, even if they are not doing any active work.

1.8.3. What Is the Network Status — the 'ruptime' Command

To check on the status of each system on the local network, use the *ruptime* command. For example, a typical display from the Sun Microsystems network is:

```
tutorial% ruptime
tutorial    up 4:07       1 user,    load   1.32,  0.92,  1.15
betasun     down 12+ 14:40
datsun      up 4:07,      1 user,    load   1.02,  0.80,  1.19
sundial     down 3+ 19:41
titan       up 4:05,      2 users,   load   1.42,  0.55    1.18
tutorial%
```

Normally the list is sorted alphabetically. Try this now on your system to see what everyone else is up to on your local network. See the user's manual page for details. Note that here as with the *rwho* command, you must have the proper network configuration to get similar results.

1.8.4. What Is the Date and Time — the 'date' Command

Another useful command is the *date* command. Use this one if you have forgotten your watch at home. Try typing:

```
tutorial% date
Fri Aug 19 10:15:08 PDT 1983
tutorial%
```

and you get back not only the day and date, but also the time (that is, the system's idea of what time it is).

1.8.5. What Is the System Doing — the 'ps' Command

Because this is a multi-tasking system, you can run several processes at once. (We'll explain how to do this in *Using the Shell.*) When you do execute several commands at one time, you may need to know how far along they are. Use the *ps* (process status) command for this:

```
tutorial% ps
PID  TTY  STAT  TIME  CMD
2025 02    R     0:01    ps
tutorial%
```

This lists the processes belonging to you. PID indicates the 'process identification' number, TTY the terminal from which the process was started, STAT the state of the process, TIME the amount of computer time used so far, and CMD the command line that was typed to initiate the process.

As you can see, there is one process in operation, the *ps* process.

With the **–x** option, the *ps* command displays all your processes:

```
tutorial% ps –x
PID             TTY     STAT    TIME    CMD
2028            02      S       0:05    ps
2029            08      S       0:05    –csh (csh)
tutorial%
```

Looking at the TTY column, note that in our example, you are logged on twice, once on tty02 and once on tty08. *Ps* displays all processes associated with the user, not those associated with a particular workstation or terminal.

Also try the *ps* command with the **–ax** option, which tells about 'all' processes going on in the system. For example:

```
tutorial% ps -ax
PID             TTY     STAT        TIME    COMMAND
0               ?       D           2:33    swapper
1               ?       I           0:20    /etc/init -
2               ?       D           0:08    pagedaemon
                        < etc. >
5535            co      S           0:22    -csh (csh)
5696            co      R           0:03    ps -ax
tutorial%
```

1.8.6. Who's Doing What — the 'w' Command

If you get really nosey, try the *w* command to check whether there is anyone else logged in to your system and what that user is doing. Try this now and decipher the abbreviations using your knowledge of what the *who* and *ps* commands display.

```
tutorial% w
12:19pm   up 1 day, 18:54,  1 users,    load average: 1.36, 1.22, 1.24
User    tty         login@    idle    JCPU  PCPU  what
evan    console     7:08am    2:31            2:16  w
tutorial%
```

Here, clearly, the user 'evan' is logged in to the 'console' and running the *w* command.

It's also polite to run *w* before using *write* to write a message to someone (we explain this in the *Communications* chapter). Check what your colleague is doing first; it's not nice to interrupt him in the middle of editing a file, for example.

2. WORKING WITH FILES

This chapter explains how to view your files and how to do simple operations on them. First, you need a file to play with. Step through the instructions to prepare the */usr/lib/units* system file for the exercises that follow. The */usr/lib/units* file converts units of measure and is something of a hodge-podge, but it's interesting to glance through. The instructions do not provide explanations of each command here, but you can always refer to the user's manual if you're particularly curious about something at this point. By the end of this tutorial, you will be able to look back at this sequence and understand each step.

Remember: If you make a typing mistake, use the DEL key to back up and correct the error. You can also type RETURN to which the system will respond 'Command not found' or 'Unmatched'. You can then retype the command.

1. Be sure you are in your home directory with *cd*:

```
tutorial% cd
tutorial% pwd
/usr/evan
tutorial%
```

2. Use the *head* command with the **–30** line option on the */usr/lib/units* file, and redirect it to a file called *start.here*:

```
tutorial% head –30 /usr/lib/units > start.here
tutorial%
```

3. Use the *tail* command with the **–30** line option on the */usr/lib/units* file and append the results to the *start.here* file:

```
tutorial% tail –30 /usr/lib/units >> start.here
tutorial%
```

4. Use the *tr* (translate) command to change tabs to spaces in the *start.here* file so the file is easier to work with. The sequence of keystrokes here is a little bit tricky; after the first apostrophe, type the TAB key. This jumps the cursor one tab space, and you can continue typing in the rest of the command line, putting spaces between the apostrophes. Here you also create your practice file called *playfile*:

```
tutorial% tr '        ' ' < start.here > playfile
tutorial%
```

5. Check your home directory with the *ls* command to see that the *playfile* is there.

```
tutorial% ls
playfile  start.here
tutorial%
```

   The original *start.here* file is there too.

You are now ready to begin working with a file of data.

2.1. Paging Through a File with 'more'

The *more* command reads one or more text files and displays the contents a screenful at a time. Try it on *playfile*, for instance:

```
tutorial% more playfile
/ dimensions
m   *a*
kg   *b*
sec   *c*
   < etc. >
wey   40 bu
weymass   252 lb
--More--(47%)
```

The '47%' message informs you what percentage of the current file's characters has been displayed so far. A 0% percentage may be displayed if you are looking at a very large file because *more* displays only integer percentages.

To display one more line at the bottom of the screen, press the RETURN key. To see the next screenful, press the space bar. To see the next 11 lines, type 'd' or ^D.

To terminate *more*, simply type 'q' for 'quit' and you return to the 'tutorial%' prompt:

```
strike   2 bu
surveyfoot   british-ft
surveyorschain   66 ft
    < etc. >
Xunit   1.00202-13m
k   1.38047-16 erg/degC
--More--(97%)
tutorial%
```

More has options that help you get to a specific line or text pattern in a file. For instance, to get to line 45 in the file *playfile*, type:

```
tutorial% more +45 playfile
tablespoon   4 fldr
teaspoon   4|3 fldr
tesla   weber/m2
   < etc. >
weymass   252 lb
Xunit   1.00202-13m
k   1.38047-16 erg/degC
tutorial%
```

To start displaying text at the first line which contains a string, 'circle' for example, type:

```
tutorial% more +/'circle'  playfile
...skipping

degree   1|180 pi-radian
circle   2 pi-radian
slug   lb-g-sec2/ft
   < etc. >
weymass   252 lb
Xunit   1.00202-13m
--More--(97%)
```

This is one of the few instances of prefixing an option with a '+'. As you saw earlier, most

options are prefixed with a '–'.

There is another command, the *cat* command (for 'concatenate' or join), which also displays a file on the screen. However, if your file has more than one screenful of data (and yours does, as it has two workstation screens or 60 lines), it zips off the screen before you can see it. Try using *cat* now just to see what it does:

```
tutorial% cat playfile
  < zip!! >
tutorial%
```

We simply mention *cat* here along with the other file viewing commands and describe its more useful capabilities later in *Using the Shell.*

2.2. Browsing Through a File with 'view'

With the *more* command, you can page through a file by typing the space bar or move through a file line by line by typing RETURN. When you want to scroll forward and backward through a file, use the *view* command:

```
tutorial% view playfile
/ dimensions
m   *a*
kg   *b*
  < etc. >
span   9 in
spat   4 pi sr
"playfile" [Read only] 60 lines, 959 characters
```

Use the following characters to move the screen:

```
^D scrolls down one half screen
^U scrolls up one half screen
^F moves forward one screenful
^B moves backward one screenful
```

Remember, the '^' means 'Hold down the CTRL key while typing the letter.'

To exit or 'quit' viewing the file, type **:q**, which is echoed at the bottom of the screen and returns you to the 'tutorial%' prompt.

You'll see in *Creating and Editing Files — The 'vi' Editor* that the *vi* editor uses these same characters.

2.3. Look at the First Few Lines of a File with 'head'

When you need to check the first few lines of a file, use the *head* command. This is one of the commands you typed to make the *playfile. Head* gives you the first ten lines if you don't specify how many you want. Here let's specify the first three lines, for instance:

```
tutorial% head -3 playfile
/ dimensions
m   *a*
kg   *b*
tutorial%
```

Head also accepts a list of filenames and will then display the first few lines from each with a special header to indicate the filename. You have two files, *playfile* and *start.here,* so let's use

those.

```
tutorial% head -4 playfile start.here
==> playfile <==
/ dimensions
m   *a*
kg   *b*
sec   *c*

==> start.here <==
/ dimensions
m          *a*
kg         *b*
sec        *c*
tutorial%
```

Here you have two separate files, showing the first four (if there are four) lines of each. Each file has a separate entry and the filename is enclosed in ==> <== as shown.

2.4. Look at the Last Few Lines of a File with 'tail'

The *tail* command is similar to the *head* command, but displays the *tail*-end of the file. Again, if you do not specify a number, you see the last ten lines. For example, to see the last three lines of the file, type:

```
tutorial% tail -3 playfile
weymass   252 lb
Xunit   1.00202-13m
k   1.38047-16 erg/degC
tutorial%
```

Precede the number given to *tail* by a minus sign to indicate that the last *x* number of lines are to be displayed. If you precede the number with a plus sign, *tail* shows all the lines from that specified to the end of the file.

```
tutorial% tail +55 playfile
tun   8 barrel
water   .22491|2.54 kg/m2-sec2
wey   40 bu
weymass   252 lb
Xunit   1.00202-13m
k   1.38047-16 erg/degC
tutorial%
```

2.5. Counting Characters, Words, and Lines in a File with 'wc'

When you need to count the lines of source code in a program or the number of words in a document, use the *wc* (word count) command. Try it now on your file. For example:

```
tutorial% wc playfile
      60     147     959 playfile
tutorial%
```

Wc provides the number of lines, words, and characters in the file. If you only want one of the three counts, use the **-l** option to count lines, the **-w** option to count words, and the **-c** option

to count characters. Try *wc* with the **–w** option to count the number of words:

```
tutorial% wc –w playfile
    147 playfile
tutorial%
```

Playfile has 147 words.

2.6. Searching for Patterns in a File with 'grep'

Grep searches one or more files for lines which contain strings of a certain pattern. Such lines are said to match the pattern.

Grep looks for a pattern which consists of a fixed character string. It is also possible to describe more complex patterns, called 'regular expressions.' (*Grep* stands for 'global regular expression print,' if that helps).

To search for a character string, give *grep* a fixed character string. To find the string 'inch' in the file, for instance, type:

```
tutorial% grep inch playfile
tutorial%
```

The 'tutorial%' prompt returns for the next command, indicating that there is no match in this case. This can be a common occurrence; it is often a matter of trying several differents aspects of a command to get the desired result.

Now try to find the string 'mercury' in the file. Type:

```
tutorial% grep mercury playfile
mercury   1.33322+ 5 kg/m2-sec2
hg   mercury
tutorial%
```

This shows that there are two such strings in the file.

Or, if you are not sure which file contains the desired string, scan more than one file at the same time:

```
tutorial% grep mercury playfile start.here
playfile:mercury   1.33322+ 5 kg/m2-sec2
playfile:hg   mercury
start.here:mercury          1.33322+ 5 kg/m2-sec2
start.here:hg           mercury
tutorial%
```

Here you see that *grep* labels 'mercury' with the name of the file in which the string is contained.

If the filename is not important, suppress it with the **–h** (omit file header) option. Now consider the example as:

```
tutorial% grep -h fuzz playfile start.here
fuzz  1
c  2.997925+ 8 m/sec fuzz
au  1.49597871+ 11 m fuzz
mole  6.022169+ 23 fuzz
e  1.6021917-19 coul fuzz
fuzz            1
c           2.997925+ 8 m/sec fuzz
au          1.49597871+ 11 m fuzz
mole            6.022169+ 23 fuzz
e           1.6021917-19 coul fuzz
tutorial%
```

This gives you several references to the string 'fuzz'.

When you want to find a specific string and not a lot of extraneous references, type it exactly as you want to find it. For instance, if you are generally interested in finding references to 'survey', type:

```
tutorial% grep survey playfile
surveyfoot  british-ft
surveyorschain  66 ft
surveyorslink  66|100 ft
tutorial%
```

You see three words containing the string 'survey'. However, if you want to find the specific string 'surveyorschain', type that string:

```
tutorial% grep surveyorschain playfile
surveyorschain  66 ft
tutorial%
```

If the pattern you're looking for contains spaces, '2 pi', for example, surround it with quote signs (') so that it forms one argument.

```
tutorial% grep '2 pi' playfile
circle  2 pi-radian
tutorial%
```

A space is used to separate arguments, so a pattern which also contains spaces must be enclosed in quotes.

If you want to find all the lines except those that match the string, use the **-v** (for invert) option. We'll leave this up to you to experiment with.

At times you want to find a string regardless of whether it is in upper or lower case. Use the **-i** *grep* option to 'ignore case'. As the *playfile* is almost solely lower case, here's an example of using *grep* with the -i option to find the string 'bool' in the file *optionsw.c*:

```
tutorial% grep -i bool optionsw.c
#define    OPT_BOOL  1
        case OPT_BOOL:    optb_destroy(ip->oi_data);
struct  optb_data
optsw_bool(struct optionsw *optsw; int line, left; char *label
optsw_bool(struct optionsw *optsw; struct rect r; char *label;
```

This finds all the lines containing 'bool' in either upper or lower case.

Use the **–n** (number) option to show the line numbers of lines that match the string or pattern, for example:

```
tutorial% grep –n fuzz playfile
15:fuzz  1
17:c  2.997925+ 8 m/sec fuzz
19:au  1.49597871+ 11 m fuzz
20:mole  6.022169+ 23 fuzz
21:e  1.6021917-19 coul fuzz
tutorial%
```

This helps when you are using an editor or some other command to process the file using line numbers.

2.6.1. Regular Expressions in Text Patterns

To search for more complex strings than simple fixed character strings, give *grep* a pattern (or template) of the text to search for. For example, 'find all words ending in ing,' or 'all 4-digit numbers appearing at the end of a line.' Such a pattern or template is called a 'regular expression.'

Grep uses certain characters called ***metacharacters*** that represent something other than themselves and have a special meaning. We describe these below with examples, but what you need to know now is that sometimes you may want to use these metacharacters to represent themselves; that is, if you want to find a string with a dollar sign '$', you need to remove its special metacharacter significance. ('$' matches the end of a line.) To remove the special significance of metacharacters, precede them with the backslash ' \ ' *escape* character. In this case then, *grep* considers the '$' sign as a dollar sign, rather than the metacharacter that matches a line end.

Also, enclose the regular expression in quotes. Single quotes (') are safest, but often double quotes (") are sufficient.

2.6.2. Match Beginning and End of Line with ^ and $

To match a string at the beginning of a line, use the '^' character. For example, to find the string 'fuzz' at the beginning of a line, type:

```
tutorial% grep '^fuzz' playfile
fuzz  1
tutorial%
```

The ' ^ ' finds only those lines starting with the word 'fuzz'.

To match a string at the end of a line, use ' $ '. For instance, to find 'fuzz' at the end of a line, type:

```
tutorial% grep 'fuzz$' playfile
c  2.997925+ 8 m/sec fuzz
au  1.49597871+ 11 m fuzz
mole  6.022169+ 23 fuzz
e  1.6021917-19 coul fuzz
tutorial%
```

The ' $ ' selects only those lines ending with the word 'fuzz'.

Preceding an expression by ' ^ ' and following it with ' $ ' as in:

```
tutorial% grep '^fuzz$' playfile
tutorial%
```

selects only those lines consisting of 'fuzz', and nothing else. This is called an 'anchored match' because it is anchored at a specific place on a line. Here, *grep* doesn't find any string to match.

If you put the ' ^ ' and ' $ ' characters in places other than the beginning of the pattern, or the end of the pattern, respectively, they lose their special meanings.

Find blank lines in a file with the expression '^$'. This pattern finds lines which have only a newline, and no other text. It doesn't locate spaces, tabs or other non-printing characters on the line. Use it with the **–n** line number option to specify where those blank lines are:

```
tutorial% grep -n '^$' playfile
12:
14:
26:
28:
tutorial%
```

2.6.3. Matching Any Character with '.'

Use the period (or 'dot') metacharacter to match any character at all. So the string ' w... ' selects all strings starting with the letter 'w', and having three more characters. To find such strings at the beginning of a line, use '^w... '. To find such strings at the end of a line, use the expression 'w...$'. Remember that spaces are counted as part of the string.

As an example, 'w...' finds the patterns:

```
tutorial% grep '^w...' playfile
water   .22491|2.54 kg/m2-sec2
wey   40 bu
weymass   252 lb
tutorial%
```

To find a real period at the end of a sentence, use the ' \ ' escape character to remove any special significance. Thus, use the expression 'w...\.'. The period metacharacter never matches the newline at the end of a line. A consequence of this is that text patterns never match across lines; they only match within a line.

2.6.4. Character Classes with [and] and –

To specify a set of characters, enclose them in brackets, '[ ]'. This matches any one of the characters inside the brackets. The expression '^[abcxyz]' finds all lines beginning with 'a' or 'b' or 'c' or 'x' or 'y' or 'z'.

Use the hyphen character ' – ' to specify a range of characters inside '[ ]', for example '^[a-cx-z]'. Here are the common regular expressions:

[a-z]	all lowercase letters
[A-Z]	all uppercase letters
[0-9]	all digits

If you are looking for four-character strings that begin with 'l' or 'L' and contain only letters, use the pattern '[Ll][a-z][a-z][a-z]', assuming that only the initial letter can be in uppercase.

In the example '^[a-cx-z]' note that the ' ^ ' metacharacter (match the beginning of the line) is outside the brackets. If it's inside '[ ]', it inverts the selection process. So the expression ' [^Ll][a-z][a-z][a-z] ' specifies all four-letter words that begin with something other than 'L' or 'l', and the pattern '^[^a-z]' finds all lines except those that begin with lowercase letters.

It should be emphasized that ranges of characters pertain to the ASCII character set, so that the pattern '[A-z]' gets you all upper and lowercase letters and the characters [ ] ^ _ ' .

This sort of confusion occurs most often when dealing with digits. The pattern '[1-30]' does NOT mean 'numbers in the range one through 30'; it means 'digits in the range 1 through 3, or 0'. It is the same as a pattern that looks like '[1230]' or '[0-3]'.

If you really wish to include ' - ' in the class of characters, it isn't necessary to escape it so long as it is positioned such that it won't be confused with a range specification. For example, a hyphen at the beginning of the pattern stands for itself: [-ab] means the pattern ' - ' or 'a' or 'b'. The same is true for the characters '[' and ']'.

2.6.5. Subsets of Regular Expressions

Here are some possible situations using regular expressions. You can find all but blank lines from a file by:

```
tutorial% grep -v '^$' playfile
/ dimensions
m   *a*
kg   *b*
   < etc. >
k   1.38047-16 erg/degC
tutorial%
```

The regular expression '^$' finds all the blank lines, and the **-v** saves all the other lines and ignores the blank lines. Figure out what this really does; it gets rid of lines that are really blank.

You can delete the apparently blank lines as well, with:

```
tutorial% grep -v '^ *$' playfile
/ dimensions
m   *a*
kg   *b*
   < etc. >
k   1.38047-16 erg/degC
tutorial%
```

The regular expression says, in effect, 'look for a beginning of line, followed by any number of spaces (including no spaces), followed by an end of line.'

If the apparently empty line contains tabs as well as spaces, replace the simple space in the above regular expression with an expression that says 'space or tab', shown here by the '^I' which means type ^I. This is the tab character, or TAB key. You used this tab character with the *tr* command to set up the *playfile.* Your screen shows:

```
'^[      ]*$'
```

Also add the **-n** option to list the line numbers of the blank lines.

```
tutorial% grep -n '^[ ^I]*$' playfile
12:
14:
26:
28:
tutorial%
```

2.7. Sorting Text Files with 'sort'

To order the contents of a file alphabetically or numerically, use *sort*. There are many options which control the sort order. The most useful are described here. For more details, refer to the *sort* utility description in the *User's Manual for the Sun Workstation*.

Sort does not expect fields on a line to appear in a fixed columnar layout. It just works on *fields*, which are normally separated by spaces or tabs (you can specify any field-separator you want). This provides a typewriter oriented approach, rather than the historical punched card orientation.

Consider a file with a random list of classical music composers and their birthdates. Sort it as follows:

```
tutorial% sort composers
Franz Haydn       1732
Franz Schubert    1797
Gustav Mahler     1860
Johannes Brahms   1833
Wolfgang Mozart   1756
tutorial%
```

Sort orders and displays the list alphabetically by first names. You can tell *sort* to sort on the last name too. Each line in the file is considered to consist of fields, the fields being separated by spaces. To get your file in order of last name, tell the program to skip one field, the first names, and then start sorting:

```
tutorial% sort +1 composers
Johannes Brahms   1833
Franz Haydn       1732
Gustav Mahler     1860
Wolfgang Mozart   1756
Franz Schubert    1797
tutorial%
```

This works as long as there are only two fields, and no middle initials, for example. Plan your sort accordingly to account for the number of fields. See the user's manual page on *sort* for more details.

For numerically ordered sorting in this example, skip two fields before starting to sort. Type:

```
tutorial% sort -b +2 composers
Franz Haydn       1732
Wolfgang Mozart   1756
Franz Schubert    1797
Johannes Brahms   1833
Gustav Mahler     1860
tutorial%
```

The **-b** option tells *sort* to ignore blanks in all fields on the line. Without this option, each field is considered to start immediately after the end of the previous field, so the spaces between the last name and the birthdate count as part of the birthdate. Also, *sort* considers ASCII characters, and the character for space has a lower value than any of the characters for the digits 0 through 9.

To restrict the effect of ignoring blanks to only the numeric field, use the **b** option in this way:

```
tutorial% sort +2b composers
Franz Haydn       1732
Wolfgang Mozart   1756
Franz Schubert    1797
Johannes Brahms   1833
Gustav Mahler     1860
tutorial%
```

Here the **+2b** is called a 'flag,' rather than an option, and gives the same results as before. However, in other situations, you will use one or the other depending on the desired results.

If you need to sort a file by numbers that do not have the same number of digits, tell *sort* that the characters in the field are to be treated as numbers, and to sort according to the arithmetic values of those numbers. Add the letter **n** (for numeric) after the number of the fields skipped before the field that is to be treated in this way:

```
tutorial% sort +2n +1 composers
Johannes Brahms 4
Gustav Mahler   9
Franz Schubert  9
Wolfgang Mozart 41
Franz Haydn     104
tutorial%
```

Using the numeric option implies that spaces are ignored, so you don't have to use the **b** option. Mahler and Schubert composed the same number of symphonies, and they are listed in alphabetical order by ordering that field (by **+1**).

To tell *sort* to reverse the order of sorting of a field, add the letter **r** (for reverse) after that field:

```
tutorial% sort +2nr +1 composers
Franz Haydn       104
Wolfgang Mozart   41
Gustav Mahler     9
Franz Schubert    9
Johannes Brahms   4
tutorial%
```

Reversing the sorting order with **r** is not restricted to numeric sorting, although that's where

you'll usually use it.

To save the output in a file, use the **–o** (for output) option followed by the name of the file that is to contain the output. You can give all the filenames to join sorted files. And if your input files are already sorted, you can simply merge them by using the **–m** (for merge) option. Refer to the user's manual for more details on these options, and then experiment with them. The *–m* option saves *sort* some work by indicating that the files are already sorted, and need only be merged.

You can also get rid of possible duplications of names in files with the **–u** (for unique) option.

Normally *sort* assumes fields are separated by spaces or tabs, but they can be separated by something else. Use the **–t** option to tell *sort* this. For example:

```
tutorial% sort –t: +5 /etc/passwd
tutorial%
```

sorts the */etc/passwd* file, which uses the colon character ':' as its field separator.

2.8. Finding Differences Between Files with 'diff'

When you need to tell what differences there are between one version of a file and another, use the *diff* utility. For example, if you want to find the new ski team members between last year's and this year's lists, type:

```
tutorial% diff skiteams.82 skiteams.83
2c2
< Shelley Curtis
---
> Norman Travers
7d6
< Harry Cuthbertson
10a10
> Karen Stevens
tutorial%
```

This indicates that there are three changes to the file. The second line has changed ('2c2'), but it is still the same line number. The first file (or old version of the line in this case) is preceded by a '<', and the second file (the new version in this case) is preceded by a '>'. The '7d6' shows that line 7 of the original file has been deleted. And '10a10' indicates the third change, the addition of a new line after line 10 in the original file.

Try the *diff* command on your two files, *playfile* and *start.here*:

```
tutorial% diff playfile start.here
```

The results zip off the screen pretty quickly (we'll explain how to manipulate that display for examination later), but you see that there are lots of differences. Try the *diff* command on some other files and interpret the results.

There are several options to the *diff* command that you should read about in the user's manual.

3. USING THE SHELL

You have seen how the Shell is the interface between you and the various Sun system commands. Here we describe some more Shell facilities that you should experiment with to do your tasks.

3.1. Redirecting Standard Input and Standard Output

When you run a program on the Sun system, it usually expects some input (data) and produces some output (results). This input or ***standard input*** is the place from which a program expects to read its input, usually your keyboard. The output or ***standard output*** is the 'file' to which the program writes its results, usually your workstation screen.

You can change these defaults by telling the Shell that the standard input for a command is to be taken from a file, or that the standard output of a command written to a file. This process is called 'redirection' and uses those '>' and '<' characters you saw in some of the examples in the chapter on *Working With Files*.

You can also take the standard input of a command directly from the standard output of another command, and similarly to direct the standard output of one command straight to the standard input of the next command. This feature is called *piping*, and a string of commands hooked together in this way is called a *pipeline*.

If a filename argument to a command is prefixed by the '>' character, the standard output of that command is *redirected* to that file instead of going to your workstation screen. For example, the *ls* command displays a list of the directory contents on your workstation screen, but if you use '> *files*' as a redirection indicator, the command line puts the contents listing of the current working directory into a file called *files*, which is placed into the current directory. Try this now from your home directory as indicated in the following example:.

```
tutorial% cd
tutorial% ls -l > files
tutorial% ls -l
total 202
-rw-r--r--  1 evan      110 Aug  6 14:03 files
-rw-r--r--  1 evan      959 Aug  6 11:20 playfile
-rw-r--r--  1 evan      959 Aug  6 11:18 start.here

tutorial%
```

You can also see that your directory now contains three files, *playfile*, *start.here*, and *files*, which contains the long *ls* listings of all the files. If the named file to which output is redirected doesn't already exist, it is created.

You can now view the file at your leisure, or change it around for any useful purposes of your own. Look in your new *files* file with the *more* command:

```
tutorial% more files
-rw-r--r--  1 evan        0 Aug  6 11:40 files
-rw-r--r--  1 evan      959 Aug  6 11:20 playfile
-rw-r--r--  1 evan      959 Aug  6 11:18 start.here
tutorial%
```

You have your current directory listing, including *files*, which is created before the command runs.

Be careful, though, because if the file to which the Standard Output is redirected already exists, the previous contents of the file are lost! The C-Shell has a ***noclobber*** variable that you can set to prevent this from happening. Read about this in the ***csh*** description in the user's manual.

If you don't want to lose the contents of an existing file, but want the output of a command appended to the end of it, prefix the filename with two right chevron signs '>>'. You used this notation before to prepare *playfile*. As another example, type:

```
tutorial% pwd  >> files
tutorial% more files
-rw-r--r-- 1 evan       0 Aug  6 11:40 files
-rw-r--r-- 1 evan     959 Aug  6 11:20 playfile
-rw-r--r-- 1 evan     959 Aug  6 11:18 start.here
/usr/evan
tutorial%
```

which prints the name of the current working directory at the end of the contents listing of the directory.

In many ways '>>' is safer to use than '>' because it doesn't destroy previous information.

Just as the '>' character or the '>>' sign redirects the Standard Output of a command to a file, you can redirect the Standard Input for a command to come from a file, instead of your workstation keyboard. Prefix the file name with a left chevron sign '<'. For example:

```
tutorial% mail < gossip
tutorial%
```

takes mail messages from the file *gossip*, created previously by an editor.

You'll use redirection of Output more often than redirection of Input because a lot of commands are designed to take their input from files anyway.

If you do not specify any files, commands use the Standard Input. The *cat* command, which we mentioned briefly in *Working with Files* as a file viewing command, is one:

```
tutorial% cat
Type what you want here.
Cat then displays it on the Standard Output,
your workstation screen.
^D
Type what you want here.
Cat then displays it on the Standard Output,
your workstation screen.
tutorial%
```

Typing a ^D tells *cat* that you have finished.

There are many commands which only act on the Standard Input, that is, what you type at your keyboard. If you want these to work on a file instead, you have to redirect the input to be from that file; for instance:

```
tutorial% cat < files
-rw-r--r--  1 evan        0 Aug  6 11:40 files
-rw-r--r--  1 evan      959 Aug  6 11:20 playfile
-rw-r--r--  1 evan      959 Aug  6 11:18 start.here
/usr/evan
tutorial%
```

concatenates the contents of the file called *files* to the Standard Output.

Also consider the *tr* (translate) command, which you can also redirect to operate on a file as input instead of the Standard Input:

```
tutorial% tr a-z A-Z < files > FILES
tutorial% more FILES
-RW-R--R--  1 EVAN        0 AUG  6 11:40 FILES
-RW-R--R--  1 EVAN      959 AUG  6 11:20 PLAYFILE
-RW-R--R--  1 EVAN      959 AUG  6 11:18 START.HERE
/USR/EVAN
tutorial%
```

As you saw above, the *cat* command can use the Standard Input or, with redirection, a file as input. What *cat* really does best is indeed what it says, it 'concatenates' or joins files with the appropriate redirection. For example, to join two files, *files* and *playfile*, to form a third file *mess* that contains both, type:

```
tutorial% cat files playfile > mess
tutorial% more mess
-rw-r--r--  1 evan        0 Aug  6 11:40 files
-rw-r--r--  1 evan      959 Aug  6 11:20 playfile
-rw-r--r--  1 evan      959 Aug  6 11:18 start.here
/usr/evan
/ dimensions
m   *a*
   < etc. >
Xunit   1.00202-13m
k   1.38047-16 erg/degC
tutorial%
```

You then have the *mess* file that contains *files* followed by *playfile*.

To add one file to the end of another, use *cat* and the double chevron redirection symbol '>>'. If you want to add the contents of *playfile* to the end of *files*, type:

```
tutorial% cat playfile >> files
tutorial% more files
-rw-r--r--  1 evan        0 Aug  6 11:40 files
-rw-r--r--  1 evan      959 Aug  6 11:20 playfile
-rw-r--r--  1 evan      959 Aug  6 11:18 start.here
/usr/evan
   < etc. >
Xunit   1.00202-13m
k   1.38047-16 erg/degC
tutorial%
```

You then have two files, the original *playfile* file and a new *files* file that contains *files* followed

by *playfile.*

Be careful that you do not use the same filename when redirecting both Standard Input and Standard Output. The first thing that happens when you use '>' is that the file it points to is created. If that file already exists and if there's something in it, the contents are lost. For example:

```
tutorial% cat < now > then
tutorial%
```

works, but in the following cases, the files are clobbered:

```
tutorial% cat < thisdata  > thisdata
cat: input – is output
tutorial%
```

or

```
tutorial% cat thisdata > thisdata
cat: input thisdata is output
tutorial%
```

The *cat* command warns you with 'cat: input – is output' that something is wrong, but continues to overwrite or clobber your file anyway. Another such problem occurs if you try:

```
tutorial% cat a b > B
tutorial%
```

which clobbers *b* before running *cat.* See the *csh* 'noclobber' option to prevent this.

3.2. Connecting Processes with Pipes

You can run the Standard Output of one process (or program) as the Standard Input of another process when you form a 'pipeline.'

Using the *wc* and *who* commands, and the '|' (vertical bar) 'pipe' symbol, find out how many people are logged in:

```
tutorial% who | wc –l
    1
tutorial%
```

Here *who* feeds its Standard Output to the Standard Input of the *wc* command, which counts lines with the –l option. And yes, that number 1 is you, if you are the only one on your system.

A pipeline is a flow of data that several programs (or processes) operate on in turn. If you write the sequence:

```
tutorial% ls –l /usr | grep evan | sort +3nr | lpr
tutorial%
```

you have a string of connected commands or a 'pipeline.' The Standard Output of the command to the left of the '|' becomes the Standard Input to the command on the right of '|'.

Here the long *ls* listing of files and subdirectories in */usr* is passed to the *grep* command instead of to the default Standard Output, the workstation screen. *Grep* selects all lines containing the string 'evan' from its Standard Input, and *sort* sorts the fourth field of all those selected lines in reverse numerical order.

Lpr routes its input to a printer, so you get a print-out instead of seeing the results on the screen. This, in fact, is the most common use of piping. Since all commands except *lpr* provide

output to the Standard Output, the only way to get a hard copy print-out of a file at the end of such a pipe sequence is to pipe it to *lpr*.

You can type all the commands in the pipeline one at a time, but here the system does all the intermediate work for you, and it's much quicker because you don't have to wait for each command to finish.

Some commands are special. *Ls* does not accept the Standard Input because its 'input' is a directory whose contents are to be listed. You can only use *ls* as the first command in a pipeline. The *lpr* command does not write to the Standard Output, instead its 'output' is to a printer somewhere. You can only use *lpr* as the last command in a pipeline.

You can use both redirection of input and output with pipelines.

3.3. Controlling Jobs

Because the Sun operating system is a multi-tasking system, you can run more than one job or 'process' at a time. You can manipulate your jobs by running them in either the foreground or the background, stopping, suspending, or killing them as we describe below.

3.3.1. Foreground and Background Processes

You saw what processes are running with the *ps* (process status) command. Everytime you ask the Shell to run a command, it runs that command as a separate process. The process which converses with your workstation is called a 'foreground' process because it is in the foreground of your attention. Normally, each command you type, or each pipeline of commands, runs as foreground processes, and they run at your workstation 'while you wait.'

This could mean a lot of waiting for processes to finish, so you can also run commands in the 'background.' Your system prompt re-appears immediately after you type the command, and you can continue with another task while it completes. This is particularly useful for commands that take a long time to run.

3.3.1.1. Running Jobs in the Background with '&'

To run a command in the background, type an ampersand '&' at the end:

```
tutorial% sort files &
[1] 2042
tutorial%
```

The '[1]' is the job number and the '2042' number reply is the PID (or process identity number you saw earlier with the *ps* command) that the operating system associates with the command you typed. Sometimes the process you put in the background calls up other processes which you don't know about, but only the number for the primary (or parent) process is shown when you type the command. In this example, *sort* displays the sorted *files* files on your screen.

You can then type another command to have the Sun system doing several things for you at the same time. Or you can log off, and the background command continues running.

Avoid having too many things going on at the same time; your system has to play ringmaster in your multi-ringed circus, so it might end up taking just as long to complete all your tasks as if you had done them one by one at the keyboard waiting for the prompt each time in the foreground.

One word of caution: if your background process is taking input from a file, don't start another command, either in the background or in the foreground, which will modify the contents of that file. Results can be unpredictable.

To ensure that your output goes to the right place and is not mixed up with other files or lost, redirect the output of the background command to a file:

```
tutorial% sort files > files.sorted &
[1] 2043
tutorial%
```

Now the output of your background command will not disturb you, or anyone else, who is executing commands in the foreground. And you can view it at your leisure with *more*:

```
tutorial% more files.sorted
-rw-r--r--  1 evan      959 Aug  6 11:18 start.here
-rw-r--r--  1 evan      959 Aug  6 11:20 playfile
/ constants
    < etc. >
wey   40 bu
weymass   252 lb
tutorial%
```

Or you can pipe the results to the printer:

```
tutorial% sort files | lpr &
[1] 1043 1044
tutorial%
```

3.3.2. Stopping and Resuming Jobs

If you start a foreground job, you can stop it temporarily with a ^Z and resume it later:

```
tutorial% spell playfile > mispell
^Z
Stopped
tutorial%
```

If you put a job in the background, you can stop it with a *stop* command:

```
tutorial% sort playfile &
[2] 2345
tutorial% stop %2
[2]+  Stopped (signal) sort playfile
```

You need to type the job number prefixed by the '%' after the *stop* command to indicate which job to suspend if you're running more than one. You can resume a stopped job later.

3.3.3. Placing Jobs in the Background

If you start a job in the foreground, and then decide you want to work on something else while it is completing, use the following sequence:

```
tutorial% spell playfile > mispell
^Z
Stopped
tutorial% bg
[1] spell playfile > mispell &
tutorial%
```

Here you start running the *spell* utility, which finds those ubiquitous spelling errors, then you decide to work on something else, and stop it with ^Z. Typing **bg** then puts that job in the background for completion.

3.3.4. Bringing Jobs to the Foreground

If you put a job in the background, and then decide you need it back in the foreground, use the following sequence:

```
tutorial% spell playfile > mispell &
[1] 321
tutorial% ls -F > mydir &
[2] 322
tutorial% who | wc
^Z
Stopped
[3] 323 324
tutorial% jobs
[1] - Running       spell playfile > mispell
[2]   Running       ls -F > mydir
[3] + Stopped        who | wc
tutorial% fg %2
ls -F > mydir
tutorial% more mydir
```

Typing the *jobs* command displays a job table to remind you which job is which. It also gives you the status: '+ ' for the current job and '-' for the previous job.

The *fg* 'foreground' command brings the background job *ls -F* to the foreground. Note that to bring a particular command to the foreground, you precede the job number with the '%' sign. Note that you can also type the command you want to bring to the foreground; that is, in this example '%ls' would mean the same thing as '%2'. You only need to type a unique part of the full command, hence the **ls**, and not the whole sequence.

You can also use *fg* to restart a job previously suspended with '^Z'.

3.3.5. Killing Jobs and Processes with 'kill'

If you start a command running in the background, and then change your mind for some reason, you can stop the process with the *kill* command and the PID. From the other example, we got 321, so to kill it you type:

```
tutorial% spell playfile > mispell &
[1] 321
tutorial% kill 321
[1] Killed      spell playfile > mispell
tutorial%
```

You can also type **kill %1** to kill the process running as job number 1.

A process can use several subprocesses called *children*. To kill them one by one may take some time. To stop them all, find the subprocess' PIDs with the *ps* command and stop them all with the same *kill*. Some processes are clever enough to ignore the stop signal number that the system sends them. If this is the case, but you really want to kill a certain command, type:

```
tutorial% kill -9  321
[2] Killed        spell playfile > mispell
tutorial%
```

The signal number 9 is a sure kill signal option. You can also use **kill 0**, to kill everything that's either running in the background or stopped.

3.4. Recalling Previous Commands with 'history'

The C-Shell has a built-in history mechanism to keep track of some number of the commands you type. This is handy for re-executing long commands and for changing parts of and re-executing previously typed commands,

You determine the number of commands by setting a variable called 'history' in your *.cshrc* file with an editor:

```
set history=30
```

After reading the chapter on the editor, come back to this part and create a *.cshrc* file with the ***history*** option. A typical display from the history buffer is:

```
tutorial% history
  1  ls -l
  2  cd /intro/lessons
  3  history
tutorial%
```

Here you clearly have not been logged on very long because you've only typed three commands. Note that the ***history*** command also appears as the last command.

If the 'history' variable is set to 30, the ***history*** command displays the last 30 commands you typed during the current login session.

The first thing to look at is correcting the previous command. If you mistype something, such as changing directories, and get the 'No such file or directory' message, you don't need to retype the whole *cd* command and pathname. Simply use the '^' character to bracket the incorrect and correct spellings and make the substition as follows:

```
tutorial% cd /usr/evan/intro/lesons
/usr/evan/intro/lesons: No such file or directory
tutorial%
```

To correct this 'lesons' typo, type:

```
tutorial% ^lesons^lessons^
cd /usr/evan/intro/lessons
tutorial%
```

or even just:

```
tutorial% ^so^sso^
cd /usr/evan/intro/lessons
tutorial%
```

History echoes the command it is executing.

The '^' characters act as delimiters to surround the two strings. The first string is the thing you want changed. The second string is the thing you want to change it to. The corrected command is echoed back at you.

Use the *history* facility to see what happened to the commands:

```
tutorial% history
  1  ls -l
  2  cd intro/lessons
  3  history
  4  cd  /usr/evan/intro/lesons
  5  cd /usr/evan/intro/lessons
  6  history
tutorial%
```

The two *cd* commands from the examples above, the erroneous one and the corrected one, now appear in the history buffer.

To run the previous command again, simply type:

```
tutorial% !!
history
  1  ls -l
  2  cd intro/lessons
  3  history
  4  cd  /usr/evan/intro/lesons
  5  cd /usr/evan/intro/lessons
  6  history
  7  history
tutorial%
```

To avoid typing out a long command line that you have run previously, type an exclamation mark in front of a unique number of characters from the previous command. For example, assume you want to re-run the *cd* command noted before. Use the '!' character and *cd* to ask the Shell to search backwards through the history file for a command beginning with the letters 'cd'.

```
tutorial% !cd
cd /usr/evan/intro/lessons
tutorial%
```

You don't have to to type the whole command name, just enough of the previous command to make it unique. So you can also type the *cd* command like this:

```
tutorial% !c
cd /usr/evan/intro/lessons
tutorial%
```

and it will do just as well.

If you ask for a command that the Shell cannot find, an error message is displayed:

```
tutorial% !xd
xd: Event not found.
tutorial%
```

You can also refer directly to a previous command in the history file, by typing the exclamation mark followed by the number of the command in the history buffer. For example, you can re-run command number 1 in the buffer like this:

```
tutorial% !1
ls -l
    < output from the ls command >
tutorial%
```

The number must immediately follow the exclamation mark.

Another version of 'repeat a previous command' is to use the dollar sign '$' to refer to the last argument of the previous command:

```
tutorial% mv /usr/evan/intro/lessons/notes extra
tutorial% pr !$ | lpr
pr extra | lpr
tutorial%
```

As well as simply repeating a previous command, you can make changes to the command at the same time. Consider the history buffer after the last few changes:

```
tutorial% history
  1  ls -l
  2  cd intro/lessons
  3  history
  4  cd  /usr/evan/intro/lesons
  5  cd /usr/evan/intro/lessons
  6  history
  7  history
  8  cd /usr/evan/intro/lessons
  9  cd /usr/evan/intro/lessons
 10  ls -l
 11  mv /usr/evan/intro/lessons/notes extra
 12  pr extra | lpr
 13  history
tutorial%
```

You can see the substitutions that were made in previous commands.

Assume that the file you want to move from Evan's directory was actually the *data* file, not the *notes* file. To avoid all the tiresome typing of the whole pathname again, use the Shell substitution capability:

```
tutorial% !11:s/notes/data
mv /usr/evan/intro/lessons/data extra
tutorial%
```

Note that this same syntax is used in the editors for making text substitutions.

3.5. Substituting with 'alias'

The Shell also provides a method of making shorthand names or *aliases* for frequently used but long-winded commands.

Assume that you are updating a series of plans for a large software project. Each plan and its related meeting minutes are in a separate directory named */misc/master/project/documents*. The last element in the pathname is *plana* for project plan A, *planb* for project plan B, and so on. Instead of typing these long pathnames everytime your group makes changes in a plan, define an alias in your *.cshrc* file in your home directory:

```
alias plana cd /misc/master/project/documents/plana
alias planb cd /misc/master/project/documents/planb
alias planc cd /misc/master/project/documents/planc
alias pland cd /misc/master/project/documents/pland
alias plane cd /misc/master/project/documents/plane
```

Now you have some new commands, so all you have to do to change directory to one is type:

```
tutorial% planb
tutorial% pwd
/misc/master/project/documents/planb
tutorial%
```

You don't even have to define five separate alias lines; one suffices if you use '\!$':

```
alias plan cd /misc/master/project/documents/plan\!$
```

The '\!$' notation refers to the last argument typed on the command line. The ' \ ' (escape) character prevents the '!$' from being expanded until a *plan* command is actually typed. Now you can change to the *planc* directory like this:

```
tutorial% plan c
tutorial% pwd
/misc/master/project/documents/planc
tutorial%
```

The C-Shell constructs *planc* out of the argument you typed, *c*, and the *plan* that was defined in the alias.

4. CREATING AND EDITING TEXT FILES — THE 'vi' EDITOR

The Sun system supports several editors, *vi* (pronounced 'vee-eye') *ex* (pronounced 'ee-ex'), *ed*, and *sed*. This chapter provides the basics for learning to use the 'display' or 'screen' editor *vi*, which is a screen-oriented version of *ex*. Many of the more useful operations that can be performed in *vi* call upon *ex* functions, so in learning *vi*, you're also gaining an understanding of *ex*. In addition, *ex* is based on the *ed* editor, but has many extensions and additional features. Here you learn about the top of the line, and a little bit about the others too. Refer to the *Editing and Text Processing on the Sun Workstation* for more details.

Vi displays a portion of your file on your workstation screen. You can move the cursor around on the screen to make changes by adding, deleting, or replacing text, and you can move the screen itself around to edit different parts of the file.

Almost every key on the keyboard is a *vi* command. There are also combinations of the SHIFT key and the other keys, and combinations of the CTRL key and other keys. Note that when we say 'A,' we mean uppercase 'A'. Lowercase 'a' means something different.

4.1. Command and Insert Modes

There are two modes in *vi*, *command* mode and *insert* mode. In command mode, you can move the cursor around the screen, scroll the screen, search for patterns, save the file, and do other operations which don't involve entering fresh text. To enter new text into the file you must be in insert mode, which you get with the 'a', 'i', and 'o' commands. You get out of insert mode by typing the ESC (ESCAPE) key (or ALT on some keyboards). The significant characteristic of insert mode is that commands can't be used, so *anything* you type (except ESC) is inserted into the file. If you change your mind anytime using *vi*, pressing ESC cancels the command you started and reverses to command mode. Also if you are unsure of which mode you are in, press ESC until the screen flashes or the bell rings; this means that you are back in command mode.

Start working on your *playfile*. To use the *vi* editor, type:

```
tutorial% vi playfile
```

The screen displays something like:

```
/ dimensions
m   *a*
kg   *b*
     < etc. >
spat   4 pi sr
"playfile" 60 lines, 959 characters
```

with the cursor at the upper lefthand corner.

Now practice using the following keys to move the cursor, scroll the screen, and so on.

4.2. Moving the Cursor

There are several ways of changing the position of the cursor. You can move it character by character, word by word, sentence by sentence, forward, backward, from one screen to another, and on and on. We present the more useful ways here, but you'll want to read further on *vi* to learn all the facilities.

4.2.1. l, h, k, j — Forward, Backward, Up, and Down

To position the cursor a character at a time, use the four keys h, j, k, l, which move the cursor as follows:

h move the cursor one character to the left
l move the cursor one character to the right
k move the cursor up one line
j move the cursor down one line

After a little practice, you'll find these easy to use because they're right under your fingers.

The BACKSPACE key on your workstation keyboard has the same function as ^H; you can use it to move the cursor to the left. Typing 'h' without holding the control key works too. Both ^J and ^N move the cursor down to the next line. The LINEFEED key has the same function as ^J. Test out all these possibilities on your workstation or terminal.

You can move the cursor several lines or characters by pressing the key repeatedly, or you can hold down the key as you go for an automatic repeat. (Some terminals have a key marked REPEAT. If you hold this key down while typing some other character, that character is repeated until you let go.) Use these features when you have long distances to move the cursor.

You can give the cursor positioning commands a preceding count to move the cursor a specified number of characters or lines. For instance, '8l' moves the cursor eight characters to the right.

If the cursor is towards the end of a line, and you move it down to a shorter line, then the cursor is placed at the end of that shorter line. However it reverts to its original horizontal position on any line that is long enough.

Moving the cursor down one line past the last line of the screen scrolls the screen up one line. You cannot go past the end of the file, there is no wrap around to the beginning. Moving the cursor up one line from the top line of the screen scrolls the screen down one line.

Some of the *vi* commands use the bottom line of the screen, to display the command or its result. The bottom line initially shows the filename and length.

4.2.2. ^, 0 and $ — Move to Beginning or End of Line

For horizontal motions along the lines, three useful functions are:

^ move cursor to first non-blank character of line
0 move cursor to the real beginning of line
$ move cursor to end of line

4.2.3. H, M, L — Move to Home, Middle, and Last Line on Screen

Three more basic cursor movements are:

H home
M middle
L last

The H command homes the cursor onto the top line of the screen, M moves the cursor to the middle line of the screen, and L moves it to the last line of the screen. H and L take preceding counts, for example, 3H moves to the 3rd line on the screen, 2L moves to the second line from the bottom. If you have a key marked HOME on your terminal, this usually achieves the same as the H command.

4.2.4. w, b, e — Moving by Words

To move over the text a word at a time instead of only character-by-character, use the commands:

w move forward to beginning of next word
e move to end of this word
b move back to beginning of word

If you are already at the beginning of a word when you type a 'b', the cursor goes to the beginning of the previous word. If you are at the end of a word when you type an 'e', the cursor moves to the end of the next word.

A 'word' is anything consisting of letters, digits and underscores, surrounded by anything which is not a letter, a digit or an underscore. This definition covers identifiers in most programming languages, so the commands are useful for editing both programs and documentation.

If your file is shorter than the screen, on-screen lines not present in the file are indicated by the tilde character (~) to avoid confusion with blank lines in the file.

All the commands 'w', 'b', and 'e', move past the end of a line. If you use them to move past the upper or lower limits of the screen, the lines scroll up the screen accordingly.

You can use a preceding count with these commands, so '5w' moves the cursor forward five words, for example.

4.2.5. (,), {, } — Moving by Sentences and Paragraphs

The '(' and ')' commands move the cursor over sentences, ')' to the beginning of the next sentence, and '(' to the beginning of the previous sentence. These commands take counts too, so '2)' moves forward 2 sentences; '3(' moves back three sentences. A sentence is defined as string of words ending with one of the characters '.', '!' or '?' followed by two spaces, or occurring at the end of a line.

The '}' (beginning of next paragraph) and '{' (beginning of previous paragraph) commands work similarly to the '(' and ')' commands except they jump the cursor by paragraphs instead of sentences. However, if your file is text to be input to the *nroff* text formatter using the macro packages *-ms* or *-mm*, *vi* recognizes the 'start of paragraph' macros available in these packages, and positions the cursor accordingly.

If the file you are editing contains text to be formatted using one of the *nroff* macro packages described in *Printing and Formatting Documents*, you can also tell *vi* to move forward or backward whole sections by using ']]' to move forward a section, and '[[' to move back a section.

If the text you are editing is C programming language source text, '[[' and ']]' move forward or backward over whole procedures.

4.3. Scrolling the Screen

Try the following commands to scroll the screen:

^D scroll down half screen
^U scroll up half screen
^F scroll down a full screen
^B scroll up a full screen

With ^F and ^B, two lines from the 'old' screen are retained in the 'new' screen for continuity. These are the same commands you use to scroll the screen with *view*.

4.3.1. Moving to Specific Lines in the File

When editing, you may wish to position the cursor at some specific line. The command you use to do this is 'G' or 'go to' command. For example, to move to the beginning, type '1G'. To move to line 45, type '45G'. Typing the G command with no preceding line number moves the cursor to the last line of the file.

To find a particular character string anywhere in the file, you use the standard '/' command. It is echoed at the bottom line of the screen where you then type the string you want to search for, either as a fixed character string, or as a regular expression. *Vi* uses the same regular expressions you learned about with *grep*. To signal the end of the character string, type either ESC, or RETURN.

Vi places the cursor at the start of the next string that matches what you typed, going forward through the file.

If the string is not in the current screen, the screen is changed to display that part of the file which contains the string. Try this now with a string that is not on the screen, and notice how the screen moves to track it.

Searches wrap around the file, so if the string isn't found between the current position and the end of the file, it wraps around and continues from the beginning of the file. If the string is not found, 'Pattern not Found' is displayed on the bottom line of the screen.

Use the '?' command plus a string to search backward through the file instead of forward. Again, what you type is echoed on the bottom line of the screen.

To repeat a forward search, type '//'. To repeat a backward search, use '??'. Or use two other commands:

n to find next occurrence of same string in same direction
N to find next occurrence of same string in reverse direction

4.4. Inserting New Text

The two basic commands to enter new text into a file are **a**, which appends text after the current cursor position, and **i** which inserts text before the current cursor position. When you give either of these commands, any following text you type gets put in the file. To stop text entry, press the ESC key.

If you are using a dumb terminal, when you type in the additional text, it might look as if you are over-typing the existing text. Typing ESC straightens it out.

If you make a mistake while entering new text, correct it in the usual way with the DEL, BACKSPACE or ^H keys.

You can also use **s** for 'substitute' to enter new text; it replaces a single character with lots of characters, until you type ESC.

There are two other simple commands for entering additional text:

I insert text at the beginning of the line
A append text at the end of the line

Regardless of where the cursor is positioned on a line, the **I** command inserts text at the beginning of that line;

Similarly, **A** adds text at the end of the line, regardless of the cursor position. Blank lines of text can be inserted by using the **A** command and making the first character you enter a RETURN.

Two other commands that can be used to add new lines of text are the 'open lines' commands, **o** and **O**. The **o** command opens up a blank line following the current line, and the cursor is placed at the beginning of the new line. Anything you type from now on, until you type ESC, becomes the new line or lines. To get just a blank line, type ESC immediately following the **o**.

The **O** command works like the **o** command, except that it opens a line above the current one, instead of below. This is the easiest way to add lines before the beginning of a file.

The uppercase **R** command replaces (overtypes) characters until you tell it to stop by pressing the ESC key. The **R** command is useful when you have a fixed format, for example, where you want to preserve column alignment.

Lowercase **r** 'replaces' a single character. For example, if you misspelled 'character' as 'cheracter,' place the cursor on the 'e' typo, type the **r**, then type the correct 'a'. The 'r' command does not require typing ESC, since you are replacing only one character.

4.5. Creating a New File

To create a new file, simply call up *vi* with the name of the file you want to create:

```
tutorial% vi stuff
```

The screen goes almost completely blank:

```
~
~
~
~
~
~
~
~
~
~
"stuff" [New file]
```

The cursor is in the top left hand corner with a row of tildes '~' down the left hand edge of the screen. The '~'s indicate that there are no lines in the file corresponding to these lines on the display. The bottom line displays the name of the file you are creating, and an indication that it is a new file.

Type **a** or **i** and type in all the text you want.

When you have finished and want to write the file away and quit the editor, type **:wq**. A message like:

```
"stuff" 50 lines, 3172 characters
```

appears at the bottom of the screen. After you have quit the editor, you see the 'tutorial%' hostname prompt again.

You can now use the editor to create a *.cshrc* file in your home directory to set the *history* and to contain *aliases*.

4.6. Deleting or Changing Text

There are several way to delete and change characters and text.

4.6.1. Deleting Text with 'x'

The simplest command to use is the 'delete character' command **x**. If you want to x-out five characters, you can do it in two different ways. You can either repeat the **x** five times, or you give the command a leading count and say **5x**.

However, if you change your mind, and want the characters replaced, the command you gave in the first place matters a lot. If you gave the **5x** command, you can get them all back with an undo command of **u** (described later). However, if you said **xxxxx**, the **u** command only undoes the last **x** command. To restore the line to its original state, use **U**.

The difference between **xxxxx** and **5x** is also important if you are going to use '.' (also described later) to repeat changes.

You cannot delete characters beyond the end of a line.

4.6.2. Deleting Words and Lines with 'dw' and 'dd'

To delete a whole word, position the cursor at the beginning of the word, and type **dw**. Again, you can give this command a count and delete more than one word at a time by typing **5dw**, for instance.

To delete a whole line, position the cursor anywhere on that line and type **dd**. The **dd** command also takes a count, **5dd** for example, to delete five lines.

Uppercase **D** deletes from the cursor position to the end of a line.

4.6.3. Changing Text

To change just one word of text, 'dimensions' from *playfile* for example, put the cursor on the beginning of the word and type **cw**. You see:

```
/ dimension$
```

with the cursor on the first letter of 'dimensions' for example. The '$' at the end of the word marks which word you are changing. Then type the word you want to substitute and ESC. This exchanges one word for the other and adjusts the space accordingly.

4.7. Writing Your File and Quitting the Editor

Consider the following ways to save all your good work. Remember that you have to be in command mode to do this, otherwise you'll find yourself entering these characters into the file.

To save your changes, type:

:w

We recommend typing **:w** every few minutes while editing to insure that your changes are saved.

If you want to save your changes and exit the editor, type any of the following:

:wq
or
:x
or
ZZ

Note the subtle differences here: **:wq** always writes out the file and then quits. The others **:x** and **ZZ** only write the file out if changes are made, and then quit. If no changes are made, they quit without writing.

Note also that with **ZZ**, you do not have to type RETURN.

If you want to exit the editor without having made any changes, type:

:q

If you want to exit the editor without saving your changes, type:

:q!

The '!' overrides any warning. If you are going to edit a file but might want to get back to the previous version, make a back-up copy of the file before running *vi*.

4.8. Correcting Mistakes with 'u' and 'U'

When you make mistakes while making changes or adding text, there are two 'undo' commands, namely **u** and **U**, which make corrections easy. The lowercase **u** simply undoes the last change you made. If you have moved the cursor away from the position at which you did the change, the cursor is repositioned to its original place after the change has been undone. Similarly, if the screen has been moved since you did the change, it is moved back to its original display. The **u** command undoes any change to the edit buffer, even if that change affected many lines.

Be careful. Giving another **u** command undoes the original 'undo' command, thus applying the change all over again, but you can undo *that* also if necessary.

The uppercase **U** command can undo several changes, but only those made on the same line. Once you move the cursor off the line, however, you can no longer use a **U** on that line.

Beware of using **u** and **U** in succession; the one does not undo the effects of the other. Remember that you can also use **:q!**.

4.9. Repeating a Command with '.'

Use '**.**', the period character, called 'dot,' to repeat a change, either at the same position or at the new position.

4.10. Running Sun Commands from Inside the Editor

When you need to return to the Shell temporarily to process a job or get some information, you don't have to leave the editor. Simply type **:!*command*** where ***command*** is the name of the command you want. For example, to display your current directory from *vi* type:

```
:!pwd
/usr/evan
Hit return to continue.
```

The system displays the results of the command at the bottom of the screen, then prompts you with 'Hit return to continue.'

You can also use the *ex* editor commands from *vi*. The 'r' (read) command, for example, adds or 'reads' another file into the one you are editing with *vi*. The 'r' command adds the second file to the first after the *current* line, that is, the line where the cursor is in command mode. To be sure you know where the current line is, type a **z<CR>** on the line after which you want the second file to be placed. This marks the current line and repositions it at the top of the screen. When you type the colon ':' character, it is echoed at the bottom of the screen. Whatever you type up to RETURN is interpreted as an *ex* command, and also echoed at the bottom

of the screen as you type it.

Use the *ex* 'r' command plus the name of the file you want to read in. For instance, to read in the file *files* into the existing *playfile*, type:

:r files

This puts *files* after the current line, that is, the line on which the cursor was before you typed **:r files**. Be sure you know what your current line is. The bottom line of the screen displays the number of lines read in. This is a handy way to join files.

To read the results of a Sun system command into the edit buffer, type:

:r !ls –l

for instance. This reads the current directory listing into the current file.

You now have quite a bit of information in your *playfile*, so you can play around with several screens full to become more facile with *vi*.

4.11. A Bit About the 'ex' Editor

For the most part, you'll probably use *vi* once you become handy with its facilities. However, the *ex* editor does have local and global text substitution capabilities that you can use from *vi*. For instance, if you decide to change the string 'dollar' in *playfile* to 'greenback', type:

:/dollar/s/dollar/greenback/

The first 'dollar' finds the string to change, and 's' says to substitute for 'dollar' the string 'greenback'. This only changes the first occurrence of the string in a file.

To change the first occurrence of the string 'dimension' to 'measure' in each line in the file, use the global substitution command ':g':

:g/dimension/s/dimension/measure/

Check through the *playfile* to see the result. Note that not only is the first 'dimension' now 'measure', but the string 'dimensionless' is now 'measureless'. Be careful how you specify the strings. This command does a global search for any string with the characters 'dimension' in it and substitutes 'measure' for the *first* occurrence in every line.

To substitute 'lint' for *every* occurrence of 'fuzz' in *every* line, use:

:g/fuzz/s/fuzz/lint/g

The first ':g' command searches globally, and the second 'g' makes the substitution called by 's/fuzz/lint/' global as well.

4.12. Other Text Editors

The Sun system also supports the *ed* and *sed* editors. *Ed* is the interactive line editor that forms part of the basic UNIX system. *Sed* is the 'stream editor' that is similar to *ed* in the operations it performs, but it is not interactive; it cannot move backwards in the edit file. You don't use *sed* to make permanent changes in a file, but to 'filter' parts of a file as you do with the *grep*, *sort*, and *awk* utilities for example.

Read more about these editors in ***Editing and Text Processing on the Sun Workstation***.

5. PRINTING AND FORMATTING DOCUMENTS

The most commonly used formatter on the Sun system is *nroff*. *Nroff* prepares neatly formatted output for a variety of printers. *Nroff* reads an input file containing unformatted text, interspersed with formatting requests and produces a neatly laid out document. The *troff* text formatter is an advanced program that produces output for a phototypesetter. *Troff* has fine control over sizes of characters, multiple fonts, and so on. *Troff* and *nroff* are compatible, so it's possible to prepare input acceptable to both.

Basically, *nroff* makes all output lines the same length, and adjusts the spacing to justify the left and right margins. You can also select page size, number of lines on the page, length of the lines, size of margins, control indentation, center headings, and underline things. With the help of 'macros,' either one you make yourself or a package like *-ms* described later in this chapter, you can get footnotes, automatic page numbering and titles, automatic paragraph numbering, and automatic generation of table of contents. This manual was formatted using *-ms* macros with *troff*.

When you want to print something without any fancy formatting, headers or footers, use the *lpr* command. You used this before in piping the sorted *playfile* to the line printer with:

```
tutorial% sort playfile | lpr
tutorial%
```

The result is raw, unformatted text.

The *pr* command 'prepares' a file for printing with running headers and footers, the filename, and the date and time the job is run.

The *fmt* command is a simple text formatter that just fills lines up to about 72 columns, so your text then looks more like something on a standard 8-1/2 x 11 piece of paper.

Brief descriptions of these follow, but if you want something fancier, use *nroff* and *troff*.

5.1. Printing a File with 'pr' and 'lpr'

The contents of a file can be printed using the *pr* and *lpr* commands. The *pr* or 'print file' command prints to the Standard Output, which is normally your workstation screen. An example of running *pr* on your *playfile* file is and piping the output through *more* is:

```
tutorial% pr playfile | more
Aug  6 11:20 1983  playfile Page 1
/ dimensions
m   *a*
kg   *b*
sec   *c*
   < etc. >
township  36 mi2
tun   8 barrel
water   .22491|2.54 kg/m2-sec2
   < etc. >
Aug  6 11:20 1983  playfile Page 2
wey   40 bu
weymass   252 lb
Xunit   1.00202-13m
k   1.38047-16 erg/degC
tutorial%
```

Pr separates the output into pages and puts a five-line header at the top and a five-line trailer at the foot of each page. The trailer at the foot of the page consists of blank lines. In the header, one of the lines forms a title consisting of a date, the filename and a page number. The date shown in the header is the date the file was last modified. You can change the header by various options to the *pr* command.

What *pr* really does is 'prepare' a file for printing. Generally you use *pr* in conjunction with the *lpr* 'line printer' command, which routes files to the printer. You call *pr* with the –p option of the *lpr* command as follows:

```
tutorial% lpr –p music
tutorial%
```

Lpr routes the named file(s), or the Standard Input if there are no named files, to the printer. It does nothing else, so your file is printed beginning at the very top of the page. Also, if your file takes more than one page, the printing carries over the perforations (assuming you are using continuous form paper) without a break. This is why you use *lpr* in conjunction with *pr*.

Only error messages are displayed on the workstation screen. The exact workings of this command and its options vary from one installation to another, since they largely depend on the number and the types of printers available. See the *User's Manual for the Sun Workstation* for details on *lpr*.

Your system has variations of the *lpr* command, or options to the command, which give you the capability of displaying the printer queue (usually the command variation *lpq*) and removing requests from the queue (usually the command variation *lprm*).

The *pr* command has many options for preparing a file. See the *pr* details in the user's manual.

5.2. Simple Text Formatting with 'fmt'

Fmt is simple and fast. Try it on your *playfile*:

```
tutorial% fmt playfile
/ dimensions m   *a* kg   *b* sec   *c* coul   *d* candela   *e*
dollar   *f* radian   *g* bit   *h* erlang   *i* degC   *j*
                  < etc. >
m2/sec2 tonne   1+6 gm torr   mm hg township   36 mi2 tun   8 barrel
water   .22491|2.54 kg/m2-sec2 wey   40 bu weymass   252 lb Xunit
1.00202-13m k   1.38047-16 erg/degC
tutorial%
```

Fmt makes the lines of text reasonably even in length.

5.3. Running 'nroff'

After creating your file containing text to be formatted interspersed with formatting requests, format the output with *nroff* and pipe it through *more* for viewing on your workstation screen:

```
tutorial% nroff file | more
```

Now you can proofread it. Or you can print it by piping the output to the printer.

5.4. A Package Deal — the '–ms' Macros

The *–ms* macro package provides an easier way to format documents than basic *nroff* if you want something more than straight paragraphs. It provides indented block and itemized paragraphs, page headers and footers, and page numbering.

Some examples of calls on *–ms* macros are:

```
.DS
.PP
.NH 2
.IP 'first stanza' 14
.DE
```

As you will see, the *–ms* calls look very similar to *nroff* formatting requests. Each consists of a 'dot' followed by two characters, optionally followed by arguments to the macro. All letters are upper case, and each macro call must be on a line of its own.

Create a new file with the editor, perhaps using the *king.art* text below, and sample the *–ms* macros. Then format your file with *nroff* and the *ms* macro option:

```
tutorial% nroff –ms filename | more
tutorial%
```

Here you have the most useful calls for basic document formatting. For the full selection, see the *Editing and Text Processing on the Sun Workstation*.

5.4.1. Paragraphs — '.PP' and '.LP'

Ms provides two basic paragraph forms, an indented paragraph (that is, the first line is indented), and a block paragraph. A standard paragraph example, unformatted and formatted is:

```
tutorial% more king.art
.PP
King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.
In the centre of the chamber King Arthur sat,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.
tutorial%
```

And the formatted version looks like:

```
tutorial% nroff –ms king.art
```

King Arthur was at Caerleon upon Usk; and one day he sat in his chamber, and with him were Owain, the son of Urien, and Kynon, the son of Clydo, and Kay, the son of Kyner, and Guenever and her handmaidens at needlework by the window. In the centre of the chamber King Arthur sat, upon a seat of green rushes, over which was spread a covering of flame-covered satin, and a cushion of red satin was under his elbow.

```
tutorial%
```

For a left block paragraph, use the '.LP' macro:

```
tutorial% more king.art
.LP
King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.
In the centre of the chamber King Arthur sat,
upon a seat of green rushes,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.
tutorial%
```

The formatted result is:

```
tutorial% nroff -ms king.art
```

King Arthur was at Caerleon upon Usk; and one day he sat in his chamber, and with him were Owain, the son of Urien, and Kynon, the son of Clydo, and Kay, the son of Kyner, and Guenever and her handmaidens at needlework by the window. In the centre of the chamber King Arthur sat, upon a seat of green rushes, over which was spread a covering of flame-covered satin, and a cushion of red satin was under his elbow.

```
tutorial%
```

If you don't like these paragraph layouts, you can change them. Refer to the *Editing and Text Processing on the Sun Workstation* for details.

5.4.2. Quoted Paragraphs — '.QP'

If you want a paragraph or paragraphs indented from the surrounding text at both the left and right edges, use the '.QP' (quotation) macro:

```
tutorial% more mark.twain
As Mark Twain wrote:
.QP
We should be careful to get out of an experience only the wisdom that is
in it — and stop there; lest we be like the cat that sits down on a
hot stove lid.
She will never sit down on a hot stove like again — and that is
well; but also she will never sit down on a cold one any more.
```

When formatted, this yields the following:

```
tutorial% nroff -ms mark.twain
As Mark Twain wrote:
```

> We should be careful to get out of an experience only the wisdom that is in it — and stop there; lest we be like the cat that sits down on a hot stove lid. She will never sit down on a hot stove like again — and that is well; but also she will never sit down on a cold one any more.

```
tutorial%
```

Try this now.

5.4.3. Lists and Descriptions — '.IP'

Use the '.IP' macro to create the so-called 'hanging indent' type of paragraph, most commonly used for lists. You can give '.IP' an argument, for instance, the number of a list item. To produce a list, format your text as follows:

```
.IP 1.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
.IP 2.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
.IP 3.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
The '.IP' macro creates lists that you can use in many documents.
```

The formatted version looks like:

1. The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.
2. The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.
3. The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.
   The '.IP' macro creates lists that you can use in many documents.

You can also make a description list, where you see the name of something on the left of the page, and a paragraph describing it on the right. In this case, you give '.IP' two arguments, the first argument is the name that is to appear on the left, and the second argument is how far to indent the text on the right. The unformatted version looks like:

```
.IP Monday 12
Finish debugging program.
.IP Tuesday 12
Meet with customers for demonstration.
.IP Wednesday 12
Discuss documentation plans.
.IP Thursday 12
Outline training class.
.IP Friday 12
Lunch with manager.
```

When formatted, this looks like:

Monday	Finish debugging program.
Tuesday	Meet with customers for demonstration.
Wednesday	Discuss documentation plans.
Thursday	Outline training class.
Friday	Lunch with manager.

5.4.4. Relative Indents — '.RS' and '.RE'

When you need to indent text in relation to previously indented text, use the '.RS' and '.RE' (relative indent start and end) macros. For example:

```
.IP 1.
pigs cows chickens ducks
.RS
.IP * 3
pigs cows chickens ducks
.IP * 3
pigs cows chickens ducks
.RE
.IP 2.
pigs cows chickens ducks
.RS
.IP * 3
pigs cows chickens ducks
.IP * 3
pigs cows chickens ducks
.RE
```

when formatted looks like:

1. pigs cows chickens ducks
    * pigs cows chickens ducks
    * pigs cows chickens ducks
2. pigs cows chickens ducks
    * pigs cows chickens ducks
    * pigs cows chickens ducks

This is particularly useful for outlines.

5.4.5. Section and Paragraph Headings

You can have numbered headings up to five levels and un-numbered headings. All headings are underlined by default (headings are made **boldface** if you are using *troff*), and may occupy several lines if required.

5.4.5.1. Un-numbered Headings — '.SH'

Use the '.SH' (section heading) macro to introduce an un-numbered heading. For instance, the output:

text and more text at the end of a paragraph.

A Section Heading

A new paragraph of text and more text
that continues and continues and continues.

was generated by using the formatting macros:

```
text and more text at the end of a paragraph.
.SH
A Section Heading
.PP
A new paragraph of text and more text
that continues and continues and continues.
```

Put the macro on one line and the actual heading on the following line or lines of the input text. Begin the first paragraph following the heading with a '.LP' or '.PP' macro to signal the end of the heading.

5.4.5.2. Numbered Headings — '.NH'

To introduce a numbered heading, use '.NH'. '.NH' implies a level 1 heading, and '.NH n' where 'n' is a number calls the corresponding level number. For instance, consider the formatting of this chapter in outline form:

```
.NH
Printing and Formatting Documents
.LP
.NH 2
Printing a File with 'pr' and 'lpr'
.LP
.NH 2
Simple Text Formatting with 'fmt'
.LP
.NH 2
Running 'nroff'
.LP
.NH 2
A Package Deal — the '-ms' Macros
.LP
.NH 3
Paragraphs — '.PP' and '.LP'
.LP
```

When formatted, it looks like:

5.4.6. The Date — '.ND' and '.DA'

To avoid having the current date at the bottom of every page of your document, put the macro call '.ND' at the beginning of your input text.

To change the date shown on the bottom of every page, put the '.DA' macro at the beginning of your input text:

```
.DA 8 June 1982
```

This also allows you to specify the exact format of date that you want, for example:

```
.DA 1982-6-8
```

5.4.7. Displays — '.DS' and '.DE'

A 'display' is some text that you want to appear as you typed it without any filling of lines, indented from the surrounding text, and kept together on the same page. Use the two macros '.DS' (display start) and '.DE' (display end); the text that appears between these forms the display. Consider the following formatted text:

```
King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.
In the centre of the chamber King Arthur sat,
upon a seat of green rushes,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.
```

which was formatted by:

```
.DS
King Arthur was at Caerleon upon Usk;
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.
In the centre of the chamber King Arthur sat,
upon a seat of green rushes,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.
.DE
```

You can further format your displays with the macros '.DS B', which justifies the left margin, then centers the whole display, '.DS L', which displays the text as is without indenting, and '.DS C', which centers each line of text individually on the page.

We used the '.DS' and '.DE' macros throughout this manual to offset the screen displays.

5.4.8. Keeping Text Together — '.KS' '.KF' and '.KE'

To keep lines of text together on one page, for a quotation, for example, use '.KS' to mark the start of the text to be kept together (keep start), and '.KE' (keep end) to mark the end of the text to be kept together. This is particularly useful for keeping the text of a table or list together for example. If there is not sufficient room on the current page for the formatted version of the text between these two macro calls, *-ms* starts a new page, leaving the remainder of the current page blank. An example looks like:

```
.KS
.IP Monday 12
Finish debugging the compiler.
.IP Tuesday 12
Meet with customers for demonstration.
.IP Wednesday 12
Discuss documentation plans.
.IP Thursday 12
Outline training class.
.IP Friday 12
Lunch with manager.
.KE
```

If you are working with text that must be kept together, but that need not immediately follow the reference to it, use the floating keep '.KF' and '.KE' macro pair. The text is kept together, and the remainder of the current page is filled with the following text.

```
.KF
.IP Monday 12
Finish debugging the compiler.
.IP Tuesday 12
Meet with customers for demonstration.
.IP Wednesday 12
Discuss documentation plans.
.IP Thursday 12
Outline training class.
.IP Friday 12
Lunch with manager.
.KE
```

5.4.9. Titles and Cover Sheets

If you want to include a title and an author, or maybe even several authors on your document, use the '.TL' macro to specify the title of the document and the '.AU' macro to show the author or authors. Only the '.DA' or '.ND' macro calls to change or suppress the date may appear before these macros.

Here is an example of calling the macros:

```
.TL
Tutorial for Beginners
.AU
Sonny Systems
.NH
GETTING STARTED
.LP
Sit down at your Sun Workstation.
```

The first page of formatted text produced looks like:

Tutorial for Beginners

Sonny Systems

1. GETTING STARTED

Sit down at your Sun Workstation.

5.4.10. Overall Page Layout

By default, *-ms* provides page numbers in the center at the top of each page of the form '-2-'. The run date appears at the bottom of each page when run with *nroff*. The top and bottom page margins are set to one inch. The line length is set to six inches, and there is no page offset. Many of these default parameters can be changed. See *Formatting Document with the -ms Macro Package* in the *Editing and Text Processing* manual.

5.5. Laying Out Tables with 'tbl'

Assume you have material that you want to lay out in tabular format, in rows and columns of text with captions, headers, and such. Numeric material requires column alignment different from alphabetic material. For this, use the *tbl* utility, which is a pre-processor for *troff* and for *nroff*. In general, the *tbl* capabilities can only be fully exploited when used in conjunction with *troff*, but some of the simpler aspects work with *nroff*.

Use the '.TS' and '.TE' macros just as you do any other set of begin and end macros to bracket the desired text:

```
.TS
Description of the Table Layout
Data to be Laid Out
.TE
```

Leaving out one or the other of this set can have all sorts of weird results on your output.

The *tbl* processor sees a table in terms of three distinct parts:

1. The overall layout or form of the table. For instance, whether the table is centered on the page, or whether the table is to be enclosed in a box.
2. The layout of each line of data in the table. This part determines how each column in the table is laid out. For instance, whether it is left-adjusted, or centered, or whether numeric data must be aligned on the decimal point.

3. The actual data (the textual material) of the table itself.

Study the following simple example:

```
.TS
tab (/) ;
l l l .
Franz Joseph Haydn/1732/104
Wolfgang Amadeus Mozart/1756/41
Ludwig van Beethoven/1770/9
Franz Schubert/1797/9
Johannes Brahms/1833/4
Gustav Mahler/1860/9
.TE
```

The line with *tab (/) ;* on it is the so called 'options' part of the table. This is the first part in the list above. In this particular case, the only option to *tbl* is to tell it that the 'tab' character is to be a slash character. Normally, *tbl* expects to see the columns of data in the data part of a table separated by real tab (^I) characters. Our example uses a visible character that is not part of the data, the slash character /, in this case because it's easier to see what's going on. Terminate the options part of the table by a semicolon.

The next part of the table header is the description of how the actual columns of data are to be laid out, part 2 from the list above. In this case, there are three left-adjusted columns, indicated by the *l* format letters. You can have many lines of format descriptions. Each line of format description in part 2 of the table corresponds to a single data line in part 3, the data part of the table. If, however, there are more lines of data in the data part of the table than there are format description lines, the last line of the format description part applies to all the remaining lines of the data. Here, the three letter *l* format letters apply to every line in the data part of the table. Terminate the format descriptions with a period at the end of the last one.

Then add the actual data of the table, each field of which is separated from the next by the / character. Here you learn the birthdates and number of symphonies composed by several well-known classical composers.

Now create a file called *music* with this text, format the table and display it for viewing by typing:

```
tutorial% tbl music | nroff -ms | more
```

Franz Joseph Haydn	1732	104
Wolfgang Amadeus Mozart	1756	41
Ludwig van Beethoven	1770	9
Franz Schubert	1797	9
Johannes Brahms	1833	4
Gustav Mahler	1860	9

You can of course also redirect the output to a file with '>', or give *tbl* a list of files, which are processed one by one in the order in which you specified them on the command line. Here the table appears on the left hand side of the page. You probably would prefer it centered in running text, in which case you add a center option to the options part of the text:

```
.TS
center tab (/) ;
l l l .
Franz Joseph Haydn/1732/104
Wolfgang Amadeus Mozart/1756/41
Ludwig van Beethoven/1770/9
Franz Schubert/1797/9
Johannes Brahms/1833/4
Gustav Mahler/1860/9
.TE
```

When formatted, this looks like:

tutorial% **tbl music | nroff -ms | more**

Franz Joseph Haydn	1732	104
Wolfgang Amadeus Mozart	1756	41
Ludwig van Beethoven	1770	9
Franz Schubert	1797	9
Johannes Brahms	1833	4
Gustav Mahler	1860	9

To align numerical entries, you must treat numbers as numbers, not just as any character. Change the format specification letters for the birthdate and number of symphonies:

```
.TS
center tab (/) ;
l n n .
Franz Joseph Haydn/1732/104
Wolfgang Amadeus Mozart/1756/41
Ludwig van Beethoven/1770/9
Franz Schubert/1797/9
Johannes Brahms/1833/4
Gustav Mahler/1860/9
.TE
```

The format specification part indicates that the second and third columns are numerically ('n') aligned columns so the data in the second and third columns of each line is aligned properly to the right. The formatted version looks like:

tutorial% **tbl music | nroff -ms | more**

Franz Joseph Haydn	1732	104
Wolfgang Amadeus Mozart	1756	41
Ludwig van Beethoven	1770	9
Franz Schubert	1797	9
Johannes Brahms	1833	4
Gustav Mahler	1860	9

To add captions for the individual columns, see what has been included in the following:

```
.TS
center tab (/) ;
c c c
l n n .
Composer/Birthdate/No. of Symphonies
.sp
Franz Joseph Haydn/1732/104
Wolfgang Amadeus Mozart/1756/41
Ludwig van Beethoven/1770/9
Franz Schubert/1797/9
Johannes Brahms/1833/4
Gustav Mahler/1860/9
.TE
```

The example above now shows an extra line in the format description part, and some extra data in the data part. The first line of the format descriptions indicates that there are to be three columns of data, each one centered within its column. This format applies to the very first line of the data. The second (and last) line of the format description part is the same as before, and it applies to all the remaining data lines in the table.

This new layout looks like:

tutorial% **tbl music | nroff -ms | more**

Composer	Birthdate	No. of Symphonies
Franz Joseph Haydn	1732	104
Wolfgang Amadeus Mozart	1756	41
Ludwig van Beethoven	1770	9
Franz Schubert	1797	9
Johannes Brahms	1833	4
Gustav Mahler	1860	9

By the way, when you describe a table, the format description part, part 2 of the table, must always describe the largest number of columns which that table has. If there are some lines which have fewer columns of data, you must indicate what to do with those specific lines.

You can also print a table with a centered overall heading line for the entire table. This is called a 'spanned heading,' meaning that that a column of data spans across into the next column.

```
.TS
center tab (/) ;
c s s
c c c
l n n .
Music Trivia
.sp
Composer/Birthdate/No. of Symphonies
.sp
Franz Joseph Haydn/1732/104
Wolfgang Amadeus Mozart/1756/41
Ludwig van Beethoven/1770/9
Franz Schubert/1797/9
Johannes Brahms/1833/4
Gustav Mahler/1860/9
.TE
```

Here the result is:

```
tutorial% tbl music | nroff -ms | more
```

Music Trivia

Composer	Birthdate	No. of Symphonies
Franz Joseph Haydn	1732	104
Wolfgang Amadeus Mozart	1756	41
Ludwig van Beethoven	1770	9
Franz Schubert	1797	9
Johannes Brahms	1833	4
Gustav Mahler	1860	9

The first line of the format description part now shows a centered, spanned column. A spanned element can span as many or as few columns as you like.

Read the *Editing and Text Processing on the Sun Workstatin* for the details of *tbl*'s other capabilities.

5.6. Formatting Mathematical Equations with 'eqn'

The *eqn* and *neqn* packages aid preparation of documents containing mathematical equations. *Eqn* is a preprocessor for *troff*, and *neqn* is a preprocessor for *nroff*. The preprocessors set the mathematics while *troff* and *nroff* do the text of a document. These turn English-like descriptions of an equation into the formatting request for generating the mathematical symbols for that equation. The capabilities are restricted by the limitations of typewriter-like printers. Enclose equations with '.EQ' and '.EN' to tell *eqn* where the equations begin and end. A simple example is:

```
.EQ
x=y+ z
.EN
```

which produces:

$$x=y+z$$

You can print mathematical symbols and names and the Greek alphabet with *eqn*.

```
.EQ
x=2 pi int sin ( omega t)dt
.EN
```

which produces:

$$x=2\pi\int\sin(\omega t)dt$$

(Note that your printer may not handle these fancy fonts.) You can also produce superscripts and subscripts for example with the words *sup* and *sub*:

```
.EQ
y = c sub 1 x sup 2 + c sub 2 x + c sub 3
.EN
```

which produces:

$$y=c_1x^2+c_2x+c_3$$

Eqn does not process '.EQ' and '.EN' other than to take care of the equation between them. So you have to center, number, and justify the equations yourself or use *eqn* in conjunction with the *-ms* macro package, which as you know, takes care of those things for you. See *Editing and Text Processing* for details.

5.7. Formatting with 'nroff' or 'troff'

Prepare the input file, using an editor, and embed the *nroff* or *troff* requests in the text of the document to be formatted. Put each request itself at the beginning of a line. A request cannot appear on the same line as the text to be formatted, although sometimes part of the text to be formatted can be given as an argument to a request. A formatting request consists of a basic *nroff* instruction, (a period or 'dot' followed by one or two characters), or a call to an *nroff* macro, optionally followed by one or more arguments separated by spaces.

Some examples look like:

```
.sp
.po 8
.bp
.in 5
.ti -3
.in + 5
.ce 4
'ul
```

Putting a space between the instruction and any numerical value that an argument may take makes the requests easier to read, although it's not necessary.

An argument may also be a number preceded by a plus sign or a minus sign. This means a change relative to the existing value of whatever it is you're altering. For example, '.in + 5' means indent the margin 5 spaces more than what it is now, and similarly '.in -4' means back off that indent by 4 spaces.

5.7.1. Page Breaks — '.bp'

To do your own page breaks, insert the '.bp' (break page) at strategic points in your document. The macro package automatically does this, but sometimes you may want to force a new page.

5.7.2. Blank Lines — '.sp'

You can put blank lines in the output by using the '.sp' request. The appropriate number of blank lines are left in the output text. For example, to put three blank lines between sentences, use '.sp 3':

```
King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.
.sp 3
In the centre of the chamber King Arthur sat,
upon a seat of green rushes,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.
```

which when formatted is:

King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.

In the centre of the chamber King Arthur sat,
upon a seat of green rushes,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.

An '.sp' request with no argument leaves one blank line in the output. Or you can produce a blank line by just leaving blank lines in the input text. Using '.sp' is better because it is easier to change later if you decide to add more or fewer blank lines.

5.7.3. Centering and Underlining — '.ce' and '.ul'

Use the '.ce' request to center lines of text. The '.ce' request without an argument centers one line, for example:

```
.ce
Tutorial for Beginners
```

centers the following line of text:

Tutorial for Beginners

To center more than one line of text, type:

```
.ce 3
ducks chickens
cows ducks chickens
pigs cows ducks chickens
```

centers the following three lines of text:

ducks chickens
cows ducks chickens
pigs cows ducks chickens

Filling is temporarily turned off when lines are centered, so each line in the input appears as a line in the output, centered between the left and right margins.

If you don't want to count how many lines you want centered, say:

```
.ce 100
Some random number of text lines
     < etc. >
.ce 0
```

The '.ce 0' request simply stops the centering process.

Note that the argument to the '.ce' request only applies to following text lines in the input. *Nroff* request lines are not counted.

Underlining is somewhat misleading, for *nroff* underlines and *troff italicizes* the words. If you want to 'underline' a heading for example, and format the text with *nroff*, use the '.ul' *nroff* request, and type:

```
.ul
ducks
```

which underlines as:

<u>ducks</u>

You can use the same numbering count with '.ul' as with '.ce'.

You can of course also this to emphasize single words:

```
The best way to learn a new system is
not just to read about it, but to
.ul
use
the facilities it provides.
```

Troff produces: 'The best way to learn a new system is not just to read about it, but to *use* the facilities it provides.' Notice that you have to arrange your input so that the word you want underlined appears on a line of its own.

5.7.4. Indentation — '.in'

To indent lines of text, use the '.in' request. For example, '.in 5' indents all following lines five spaces.

```
.in 5
King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.
.in 0
In the centre of the chamber King Arthur sat,
upon a seat of green rushes,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.
```

Use '.in 0' to turn off the indent.

```
    King Arthur was at Caerleon upon Usk;
    and one day he sat in his chamber,
    and with him were Owain, the son of Urien,
    and Kynon, the son of Clydo, and Kay, the son of Kyner,
    and Guenever and her handmaidens at needlework by the window.
In the centre of the chamber King Arthur sat,
upon a seat of green rushes,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.
```

5.7.5. Temporary Indents — '.ti'

For a temporary indent in relation to the current, use the '.ti' (temporary indent) request. This works on the following line only. For example:

```
.ti 5
King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.
```

produces:

```
    King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were Owain, the son of Urien,
and Kynon, the son of Clydo, and Kay, the son of Kyner,
and Guenever and her handmaidens at needlework by the window.
```

5.7.6. Filling — '.nf' and '.fi'

If you don't want lines filled in, use the '.nf' (no fill) request. Lines are still left justified. To turn filling back on after you've entered text, type the '.fi' (filling) request. The formatted version follows the unformatted version here.

King Arthur was at Caerleon upon Usk;
and one day he sat in his chamber,
and with him were
.nf
Owain, the son of Urien,
and Kynon, the son of Clydo,
and Kay, the son of Kyner,
and Guenever
and her handmaidens at needlework by the window.
.fi
In the centre of the chamber King Arthur sat,
upon a seat of green rushes,
over which was spread a covering of
flame-covered satin,
and a cushion of red satin was under his elbow.

King Arthur was at Caerleon upon Usk; and one day he sat in his chamber, and with him were
Owain, the son of Urien,
and Kynon, the son of Clydo,
and Kay, the son of Kyner,
and Guenever
and her handmaidens at needlework by the window.
In the centre of the chamber King Arthur sat, upon a seat of green rushes, over which was spread a covering of flame-covered satin, and a cushion of red satin was under his elbow.

6. COMMUNICATIONS

The Sun system provides several facilities for communicating with local and remote hosts. You can use *mail* to send messages, which your friends can read at their leisure, save and respond to as necessary. For a quick message, there's the *write* command that sends the message to another user immediately. The network news provides a widely distributed network for communicating news items. The *tip* utility provides dial-up phone access to other systems. For more information on *mail* and the network news, refer to the *Mail User's Guide* and the *Network News User's Guide* in Part Two of this manual. For information on *write* and *tip*, see the *User's Manual for the Sun Workstation*.

For set-up instructions on either of these two facilities, refer to the *System Manager's Manual for the Sun Workstation*.

6.1. The Electronic 'mail' System

When you send *mail* to another user, the messages pile up in a 'mailbox.' If your recipient is logged in and has *set mail* set in his *.login* file, he is notified that mail has arrived when he completes whatever command he is using at the time. If he is not logged in, your message is saved in the mailbox file, and the recipient is notified that he has mail the next time he logs in.

The *mail* program keeps track of what you do with your mail; it records whether you throw a message away, save it in your mailbox, write it to a separate file, or respond to it, for example.

When you log in, or sometime during a work session, you will receive the message:

```
You have mail.
```

or

```
You have new mail.
```

indicating that you have mail in your 'mailbox.'

6.1.1. Reading Your Mail

To read your mail, type:

```
tutorial% mail
Mail version 2.17 12/26/82. Type ? for help.
"/usr/spool/mail/evan": 1 message 1 unread
→U 1 lori@tutorial Tue Oct 29 12:43
&
```

The system responds with a numbered list of messages. To read your mail, type either RETURN for the next message or:

```
& p
From lori Fri May  4 21:20:03 1983
Date: 4 May 83 21:19:54 PDT (Fri)
From: lori (Lori  Rosen)
Message-Id: <8307020419.AA01512@sun.uucp>
Received: by sun.uucp (3.320/3.14)
                        id AA01512; 4 May 83 21:19:54 PDT (Fri)
To: evan
Status: R

Are you going to the birthday party this evening?
I need a ride.
&
```

for the current message, or **p***n* where the number *n* is for the message of that number. The example above is an approximate representation as the message heading varies with who sent you the mail, from what system, and so on.

To save the message in a file for future reference, type:

& **s** *filename*

where *filename* is your chosen filename. Note that this appends the message to the named file and does not overwrite any existing contents.

To quit the *mail* program and have your *mail* correspondence updated automatically on what messages you have and have not read, type:

& **q**

To delete the message you just read, type:

& **d**

To delete a specific message, number 5 for instance, type:

& **d5**

Your mail is erased, unless you 'undelete' a specified message with the ***u*** command before you leave *mail.*

And if you want to leave *mail* and get back to the Shell without making any changes to your messages or reading any of your mail, type:

& **x**

for 'exit.' Any deleted messages are undeleted.

You can get help at any time during this correspondence by typing a '?' sign, and *mail* displays a quick and dirty summary of the most helpful responses.

& **?**

```
q         quit
x         exit without changing mail
p         print
s[file]   save (default mbox)
w[file]   same without header
```

```
-          print previous
d          delete
+          next (no delete)
m user     mail to user
! cmd      execute cmd
?
```

6.1.2. Replying to Mail

There are two ways to reply to mail. You can type an 'r', which sends a response to everyone who received the original message. Or, you can type a 'R' to respond only to the sender of the message and not to any of the names listed as Cc:'s (more on this later).

To reply to everyone, type:

```
& reply
```

at the mail prompt or just 'r' for short.

```
& r
To: lori
```

You see the message header 'To: lori' for example, and you can type in your response, terminated by a ^D.

6.1.2.1. Your Own Mailbox or 'mbox'

To save the mail in your *mbox* (mailbox) file, type:

```
& q
```

for 'quit' when you are done reading your mail. This appends any messages you have read, but not deleted during the current session to the *mbox* file in your home directory or creates the file the first time you use 'quit'.

6.1.3. Sending Mail

Assuming that you are using the Sun system network, and that you are sending mail to one of your colleagues on another host system, use the *mail* command and indicate the recipient and his hostname. For instance, to send mail to 'kathy' whose hostname is 'venus,' type:

```
tutorial% mail kathy@venus
We are moving the project due date up one week.
Let me know if this causes a problem.
^D
EOT
tutorial%
```

Ask your system administrator for the names of other users' hosts.

To send mail to someone with an account on the same system is even easier. Assuming you are logged in to 'angel' and want to send mail to Steve, who also has an account on 'angel', simply type:

angel% **mail steve**
Will you be out of town on business next week?
^D
angel%

Try sending mail to yourself. Be patient; mail delivery is not instantaneous, and it often takes a few seconds for you to be notified that 'You have new mail.' (*Mail* is handled by a background process.)

6.1.4. Personalizing Your Mail in Your .mailrc File

Just as you have a *.login* and a *.cshrc* file in which you can customize your account, you can also create a *.mailrc* file in your home directory to prompt you for additional information in messages and to provide the same kind of *alias* shorthand as in your *.cshrc* file.

If you would like to be prompted for a 'Subject:' header, include the 'ask' option in your *.mailrc* file:

set ask

Now try sending yourself mail as follows:

tutorial% **mail lori**
Subject: **Sending Mail**
This shows how to get a 'Subject:' header
when sending mail.
^D
tutorial%

If you want to send copies of a letter to a distribution list, edit your *.mailrc* file to include:

set askcc

which will then prompt you with 'Cc:' when you terminate a message with ^D. Again, try it on yourself:

tutorial% **mail lori**
Subject: **Sending Copies**
This shows how to send copies
of messages.
^D
Cc: **Chris**
tutorial%

6.1.4.1. Distribution Lists and Aliases

As you learned in the chapter on *Using the Shell*, you can set aliases for long lists of commands or names. This is particularly useful here for distributing copies of letters to the various members of a project group for instance.

Put an 'alias' in your *.mailrc* file, for example:

alias gang jon tom marty steve evan@venus

which specifies the members of a particular project.

```
tutorial% mail gang
Subject: Priorities
It is critical to get the product completed by November.
^D
tutorial%
```

Remember that all these message recipient names are really login names, but the alias may include 'evan@venus' for mailing to remote hosts.

6.2. Writing to Other Users with 'write'

The *write* command immediately sends a message to a specified user, when you type the message, provided that the recipient is logged in to the same system at the time. Use *write* only when it's a real emergency because it does bother some people when messages start appearing while they're trying to type at the workstation.

If your recipient is logged in over a phone line, their phone will be continuously busy, so *write* is the easiest way to reach them.

A typical example of how to use *write* is:

```
tutorial% write joann
Is the customer demo ready?
^D
tutorial%
```

You are not prompted after you type **write** and the login name, so simply type in the message you want to send, as many lines as you like, and end it with the end-of-text character, ^D. You don't get any indication that your message has actually been received.

What the recipient, "joann", sees is:

```
Message from tutorial!lori on tty08 at 10:42 ...
Is the customer demo ready?
EOF
```

'tutorial' is the hostname of the system that Joann and Brad are using. The EOF indicates that the message is finished, and Brad has quit writing.

This is a simple one-line message, but if you send more than one line, your recipient doesn't get it all at once. He sees each line only after a RETURN is typed, so the EOF is the only indication that the message is complete. This causes problems for a two-way conversation, which is usually what you'll use. For example, suppose Brad tries to get in touch with Jay:

```
tutorial% write jay
```

but doesn't immediately enter the message to see if Jay is logged on. Jay receives a message that says:

```
Message from tutorial!brad on tty08 at 10:45 ...
```

At this point Jay gives his own *write* command:

```
titan% write brad
Hi. What?
```

Jay doesn't type ^D, so now Jay and Brad are 'talking' to each other, until one of them types ^D to drop out of the conversation.

At the end of the conversation, Brad's screen might look like this:

```
tutorial% write jay
Message from tutorial!jay on tty04 at 10:45 ...
Hi. What?
How about discussing the project this afternoon?
There's a department meeting. Sorry.
Maybe tomorrow
morning?
Sounds good.
^D
tutorial%
```

What Brad has typed is shown boldface, Jay's replies are in normal type. Jay's screen looks like:

```
tutorial% Message from tutorial!brad tty08 at 10:45 ...
write brad
Hi. What?
How about discussing the project this afternoon?
There's a department meeting. Sorry.
Maybe tomorrow
morning?
Sounds good.
EOF
^D
tutorial%
```

Here we have shown what Jay typed boldface, and Brad's contributions in normal type. Here the dialogue consisted of simple one-line questions, but if you are going to provide several lines of information rather than ask questions, use the protocol, which suggests that you terminate each message with the character **o**, for 'over', and when you are about to quit the conversation, type **oo**, for 'over and out'. You can also set up your personal protocol with your colleagues.

If you try to write to someone who is not logged in, you get a message:

```
tutorial% write sylvia
sylvia not logged in.
tutorial%
```

and *write* terminates. You get the same response if you try to write to someone who is not a user, because *write* doesn't check for known system users, only for users logged in. Remember that you can always use the *who* command to see who is logged in:

```
tutorial% who
susan   tty00   08:30
henry   tty03   08:31
jay     tty04   09:05
susan   tty06   09:15
hank    tty07   09:15
brad    tty08   09:10
tutorial%
```

If you have already started typing your message when the system finds that the user is not logged in, and decides to ignore your command, cancel what you have typed using your line kill

character (^U). This is not critical, but it can be disconcerting when the system tries to interpret your partial message as a command next time you press RETURN.

You may try to write to someone and get the response:

```
tutorial% write hank
Permission denied.
tutorial%
```

This means that the recipient is using one of the system commands that 'locks out' the *write* command to prevent messages from messing up nicely formatted output. This is the same message you get if you try to write to a user who has used *mesg* to prevent you writing to his workstation. *Mesg* is described later.

What's really happening here is that you are writing to the user's workstation or terminal, for example, to tty08, where our *who* example shows Brad is logged in.

The example of the *who* command shows that the user 'susan' is logged in to the system twice, once on tty00, and again on tty06. Choose one, and if you don't get a response, quit that *write* and try to get the user on the other:

```
tutorial% write susan tty06
Are you there?
^D
tutorial% write susan tty00
Are you there?
Message from tutorial!susan tty00...
Yes. What's up?
Working late?
Yes. See you tomorrow.
EOF
^D
tutorial%
```

If you don't specify a terminal, *write* chooses one for you:

```
tutorial% write susan
susan logged more than once
writing to tty00
```

Write chooses the lowest number terminal, so if you don't get a response there, specify the particular tty number to get messages sent to the other terminal.

If you have a long message to communicate, use a text editor to prepare the message in a file and correct any mistakes before anyone else sees them. Send this message by redirecting the Standard Input:

```
tutorial% write jay < message
tutorial%
```

The recipient receives the message all at once. There is no waiting between lines, as there is when the message is typed following the *write* command. Use the *mail* command described above to send very long messages so your recipient can choose his own time to read them.

Here as in the editor, you can use the exclamation mark character *!* to call a system command. For instance, suppose you want to find the location of some files and let someone else know. You change to the desired directory and start writing. If you forget the name of the directory, you can find out in the middle of writing by saying:

```
tutorial% write joe
The files you want are in the directory:
!pwd
/dd/project/brad/docs/memos
!
/dd/project/brad/docs/memos
^D
tutorial%
```

You still have to type in the output of the command called by '!'. Of course, the command you give following '!' may have nothing to do with the message you are sending. You may also redirect the Standard Output of the command, so you don't see any output on the screen. The second '!' signals command termination.

6.3. Preventing Message Interruptions with 'mesg'

If you dislike being interrupted by your friends' messages while you are programming at your workstation and especially when you are using a text editor, set the *mesg* command to 'n' to stop incoming messages:

```
tutorial% mesg n
tutorial%
```

This 'no' tells the system to prevent someone else from writing to you. The message sender sees:

```
Permission denied.
```

The default is 'yes', and you can check how yours is set by typing:

```
tutorial% mesg
is y
tutorial%
```

As a first-time user, yours is undoubtedly still enabled. Setting *mesg* to 'no' only lasts for your current login session unless you set it permanently. To do this, edit your your *.login* file to include:

```
mesg n
```

6.4. Local Area Network Facilities

Your local network lets you log in to other systems or 'rhosts' (remote hosts) to do work, copy files, or whatever.

6.4.1. Making Connections with 'rlogin' and 'rsh'

Use the *rlogin* (remote login) command to log in to another system. For example, to do a remote login to the 'angel' system from your host 'tutorial,' use the *rlogin* command:

```
tutorial% rlogin angel
Last login: Mon Jul 11 22:23:40 am ttyp0
Sun UNIX 4.2 (Berkeley beta release) (GENERIC) #8: Wed Oct 23 13:45:52 PDT 1983
angel%
```

You can then login as you normally do, although a password is not necessary if you log in as the same user on an equivalent host. In both cases, your local user name must exist on the remote

host to allow remote command execution as there isn't any password prompt. Your system administrator sets this up for you.

You can write to another user on the system or send mail without having to provide a hostname if you're on the same system.

Log out as you normally do:

```
angel% logout
Connection closed.
tutorial%
```

You can go on with your local work as before.

If you want to execute a single command on another system, use the *rsh* (remote Shell) command. For instance, to find out who is logged in on 'angel', say:

```
tutorial% rsh angel who
lori console  Jul 28  09:48
tutorial%
```

6.4.2. Copying Files From Other Systems with 'rcp'

Your local network also allows you to copy files from one system to another with the *rcp* (remote copy) command. For example, if you need one of your colleague's files from another system, type:

```
tutorial% rcp krypton:/usr/henry/misc/plan .
tutorial%
```

This copies from */usr/henry/misc* on system 'krypton' the *plan* file into your current (or dot '.') directory on your host 'tutorial'. If you need to copy a directory with its contents, use the **–r** (recursive) option:

```
tutorial% rcp –r krypton:/usr/henry/misc .
tutorial%
```

This copies all of the *misc* directory, including any subdirectories and files, into your current directory.

Note that if you're using '[ ]', '*', '$', '~', '<', '>', '`' or '?', as with *rsh*, you must enclose the path in quotes. For example:

```
tutorial% rcp –r 'venus:/usr/kathy/misc/chap*' .
tutorial%
```

This quotation prevents some strange and undesirable filename expansion.

6.5. Additional Communication Facilities

The Sun system supports several additional communication facilites for which detailed descriptions are out of the scope of this manual. We introduce these facilities to you here and recommend that you read the indicated documentation and user's manual pages for instructions on how to use them if you are interested.

The *uucp* (UNIX-to-UNIX copy) utility provides networking of machines over phone lines. The *mail* facility uses it to send mail to users at remote sites and the network news facility described below uses it to transmit news articles. The *tip* facility permits file transfers between machines.

6.5.1. Network News

The network news, or simply *netnews*, provides access to the USENET (User's Network). *Netnews* is a communication facility for sharing information among a large number of users. It can be described as an electronic bulletin board. You can send articles from one machine to another for limited or very wide distribution, post an article to interested persons, browse through old news, post follow-up articles, and send direct electronic mail replies to the author of an article. You can select the articles, which are arranged in categories called *newsgroups*.

See the *Network News User's Guide* in Part Two of this manual for details.

6.5.2. Dialing to Remote Systems with 'tip'

With the *tip* utility, you can dial in with a phone hook-up to a remote system to transfer files from one machine to another. You must have a login name on the remote machine. You can use either the name of the remote system or the phone number with the *tip* command to make the connection.

A typical example is:

```
tutorial% tip tymnet
dialing ... connected
```

Or you can use the phone number of the desired system by typing:

```
tutorial% tip XXX XXXXXXX
dialing ... connected
```

where *XXX XXXXXXX* is the phone number. Ask your system administrator for specific details.

Occasionally, you may receive the response:

```
dialing ... no answer
EOT
```

or

```
all ports busy
```

which can mean a number of things, such as all outgoing lines are busy.

A tilde (~) at the beginning of a line is the escape character.

To logout, type ~.

For a full description of *tip*, refer to the user's manual.

7. SUN SYSTEM SUMMARY

This summary provides brief descriptions of the Sun system facilities.

General Characteristics

Device-independent I/O and redirection

Highly efficient buffered stream I/O is integrated with formatted input and output.

Virtual memory

Supports processes up to 16 megabytes (given adequate disk space for paging) for greatly enhanced amount of available main memory and reduced delays when running programs as only the parts of the program needed to be loaded in core are in fact loaded.

Hierarchical File System

The file system uses large blocks on the disk and incorporates knowledge of disk geometry to maximize throughput and minimize seek time.

Job Control Facilities

Support for multiplexing of terminals between jobs; running several jobs at once, some in the background and others in the foreground and moving running jobs from background to foreground and vice-versa.

Interprocess Communication

Network communication using standard protocols. Inter-process communications integrated into UNIX. User access to interprocess and network communication through sockets. Arbitrary processes in the system may communicate in either a message or stream oriented fashion. Provide remote logins, copies, and Shells over the local network.

Diskless Operation

The diskless workstation allows larger and more cost effective drives to be used by centralizing disk storage. In a clustered configuration, file-sharing is promoted by the existence of file servers, which support all disk and paging traffic. Access protocol protects clients of the server against server crashes, a server crash resulting only in interrupted service.

Networking

The 4.2 system includes an ISO-OSI model local networking subsystem. Fully supported is the DARPA internet family of protocols and associated addressing. The datagram (UDP) and stream (TCP) protocols are supported, as well as the error message protocol (ICMP) and packet forwarding at the internet layer (IP). A routing information protocol allows hosts to determine the shortest route to a destination within the local network.

C-Shell

A powerful interactive command interpreter, which provides foreground-background type job control, a history mechanism to greatly reduce the amount of typing at the command level, and a macro-like aliasing facility for personalizing the command environment. It supports string variables, trap handling, structured programming, user profiles, settable search paths, and multilevel filename generation.

Mail

User-friendly interactive mail facility that makes it easy to deal with large volumes of mail.

SunCore

An implementation of the ACM Siggraph standard package of graphics software, plus extensions. *SunCore* is implemented to level 3C of the ACM Core specification for output primitives and to level 2 of the ACM Core specification for input primitives. Extensions to the Core include textured polygon fill algorithms, including 2D and 3D operations, raster primitives, rasterop attributes, shaded surface polygon rendering and hidden surface elimination.

SunWindows

The *SunWindows* system emphasizes extensibility, accessibility at multiple layers, and provision of appropriate parts and development tools. Specific applications are provided as examples, and the system is designed to be expanded by clients. There is open access to lower levels, and convenient and powerful facilities for common requirements at higher levels. Standard tool windows are available, such as shell tools and graphics tools. These windows may overlap and be manipulated via pop-up menus.

Languages

Compilers for C, Pascal, and Fortran 77; symbolic debugger *dbx* for C and F77 programs.

Editing and Text Processing

The *vi/ex* pair of editors for use with either line-oriented or full-screen terminals; include regular expression searching. *Nroff/troff* document formatting facilities for line printer and phototypeset output.

Program Development

The Sun operating system provides a solid base for a flexible program development environment. A powerful set of utilities allows full use of the UNIX-based (Berkeley 4.2) operating system.

diff Compares two files and report differences. Checks C programs for syntax errors, type violations, and portability problems.

make Indispensable tool for making sure that large programs are properly compiled with minimal effort through a control file specifying source file dependencies; knows about *cc*, *yacc*, *lex* and so on. Source Code Control System for maintaining and controlling multiple versions of text files.

dbx A source level debugger for C programs.

adb Low-level symbolic debugger; examine arbitrary files with no limit on size; interactive breakpoint debugging with the debugger as a separate process; symbolic reference to local and global variables; patching; stack trace for C programs; and output formats of: 1-, 2-, or 4-byte integers in octal, decimal, or hex; single and double floating point; character and string; and disassembled machine instructions.

yacc A parser generator for BNF grammars.

ar Maintain archives and libraries; combine several files into one for housekeeping efficiency; create new archive; update archive by date; replace or delete files; print table of contents; and retrieve from archive.

as Call assembler to create object program consisting normally of read-only and sharable code, initialized data or read-write code, uninitialized data; relocatable object code is directly executable without further transformation; object code normally includes a symbol table; 'conditional jump' instructions become branches or branches plus jumps depending on distance; searching for integer, character, or floating patterns.

od Dump any file; output options include any combination of octal or decimal or hex by words, octal by bytes, ASCII, opcodes, hexadecimal; range of dumping is controllable.

ld Link edit; combine relocatable object files. Insert required routines from specified libraries; resulting code is sharable by default.

nm Print the namelist (symbol table) of an object program; provide control over the style and order of names that are printed.

size Report the memory requirements of one or more object files.

strip Remove the relocation and symbol table information from an object file to save space.

time Report timing information on a command execution.

prof Construct a profile of time spent per routine from statistics gathered by time-sampling the execution of a program; subroutine call frequency and average times for C programs.

Graphics Tools and the Window System

Pixrects Device-independent interface to pixel operations.

SunWindows
A tool for overlapping windows, including imaging control, creation and manipulation of windows and distribution of user inputs.

Suntools A multi-window executive and application environment supporting pop-up menus, icons and combined text and graphics.

Icon Tool Simple bit-map editor.

Communication Facilities

The Sun system provides an electronic mail facility, access to the USENET network, and facilities for transferring files to and from remote machines. Some of these facilities include:

mail Mail a message to one or more users; read and dispose of each message individually; the presence of mail is announced by *login* and optionally by *csh*; save messages in files or forward them; and support for items such as 'Subject:' and 'Cc:' fields.

network news
Access to USENET news articles.

tip Utility to establish full-duplex connection for logging in to remote UNIX systems via dialup lines; provide transparent interface to remote machine; transmit files; take remote input from local file or put remote output into local file.

uucp Perform spooled file transfers between two UNIX machines; provide automatic queuing until line becomes available and remote machine is up; copy between two remote machines.

write Establish direct workstation or terminal communication with another user.

wall Write to all users.

mesg Inhibit receipt of messages from *write* and *wall*.

calendar Provide automatic reminder service for events of today and tomorrow.

User Access Control Facilities

Commands include:

login Sign on as a new user; verify password and establish user's individual and group (project) identity; adapt to characteristics of a terminal; establish working directory; announce presence of mail; publish message of the day; execute user-specified profile; start command interpreter or other initial program.

rlogin Log in to another machine.

rwho See who is logged in on the local network.

passwd Change a password; user can change his own password; passwords are encrypted for security.

File Manipulation Facilities

Commands include:

cat Concatenate one or more files onto standard output; used for unadorned printing, for inserting data into a pipeline, and for buffering output that comes in dribs and drabs; and works on any file regardless of contents.

cp Copy one file to another, or a set of files to a directory; works on any file regardless of contents; and can also copy directory hierarchies.

rcp Copy files and directories from other machines.

cmp Perform binary comparison.

pr Prepare files for printing by a printer program; place title, date, and page number on every page; provide multicolumn output and parallel column merge of several files.

lpr Spool arbitrary files to printer for off-line printing; usually used in conjunction with *pr*.

head Display first 'n' lines of input.

tail Display last 'n' lines of input.

split Split a large file into more manageable pieces.

dd Translate physical file format for exchanging data with foreign systems.

sum Sum the words of a file, providing convenient checksum.

Directory and Filename Manipulation Facilities

Commands include:

rm Remove a file; only the name goes away if any other names are linked to the file; step through a directory deleting files interactively; and delete entire directory hierarchies.

ln 'Link' another name or 'alias' to an existing file.

mv Move a file or files; rename files or directories; and move whole directory hierarchies.

chmod Change permissions on one or more files; executable by files' owner.

chgrp Change group (project) to which a file belongs.

mkdir Make a new directory.

rmdir Remove a directory.

cd Change working directory.

find Prowl the directory hierarchy finding every file that meets specified criteria; perform specified command on each file found; criteria include: name matching a given pattern, creation date in given range, date of last use in given range, given permissions, given owner, given special file characteristics, or boolean combinations of above. Any directory may be considered to be the root.

Running Programs

Commands include:

csh A flexible user interface, featuring a C-like command syntax, macro facilities, a *history* facility for reissuing previous commands, and job control. The C-Shell, the command language interpreter written at the University of California, Berkeley; supply arguments to and run any executable program; redirect standard input, standard output, and standard error files; execute simultaneously the output of one process connected to the input of another through pipes; compose compound commands using:

if ... then ... else,
case switches,
while loops,
for loops over lists,
break, **continue** and **exit**,
parentheses for grouping.

initiate background processes; perform Shell programs, that is, command scripts with substitutable arguments; construct argument lists from all filenames satisfying specified patterns; take special action on traps and interrupts; provide user-settable search path for finding commands; execute user-settable profile upon login; optionally announce presence of mail as it arrives; and provide variables and parameters with default setting.

rsh Execute command on another system.

sh The Bourne Shell, the UNIX version 7 command language interpreter.

test Test for use in Shell conditionals; string comparison; file nature and accessibility; and boolean combinations of the above.

expr Calculate command arguments with string computations; integer arithmetic; and pattern matching.

wait Wait for termination of asynchronously running processes.

echo Display remainder of command line; useful for diagnostics or prompts in Shell programs, or for inserting data into a pipeline.

sleep Suspend execution for a specified time.

nohup Run a command immune to hanging up the workstation.

nice Run a command in low or high priority.

kill Terminate named processes.

at Schedule a one-shot action for an arbitrary time.

tee Pass data between processes and divert a copy into one or more files.

System Manager's Tools

Automatic boot procedures to bring up Sun UNIX. Automatic reboot and file consistency checks and repair in the event of system crash.

Commands include:

config Configure and create bootable UNIX kernels with non-standard hardware configurations.

su Become the super-user temporarily with all the rights and privileges thereof.

chown Change the ownership of one or more files.

cron Schedule regular actions at specified times; actions are arbitrary programs; and times are conjunctions of month, day of month, day of week, hour and minute; ranges may be specified for each.

mount Attach a device containing a file system to the tree of directories; protect against nonsense arrangements.

umount Remove the file system contained on a device from the tree of directories; protect against removing a busy device.

mkfs Make a new file system on a device.

newfs Provide front-end to *mkfs*.

mknod Make an i-node (file system entry) for a special file; special files are physical devices, virtual devices, physical memory, etc.

tar Manage file archives on magnetic tape; collect files into an archive; update tape archive by date; replace or delete tape files; print table of contents; and retrieve from archive.

dump Dump the file system stored on a specified device, selectively by date, or indiscriminately.

restore Replaces the old *restor* for restoring a dumped file system, or selectively retrieving parts thereof.

fsck Interactive file system check and repair program; supersedes *dcheck*, *icheck*, and *ncheck*; print gross statistics: number of files, number of directories, number of special files, space used, and free space; report duplicate use of space; retrieve lost space; report inaccessible files; check consistency of directories; and list names of all files.

sync Force all outstanding I/O on the system to complete; used to shut down gracefully.

Status Inquiry Commands

Commands include:

ls List the names of one, several, or all files in one or more directories; alphabetic or temporal sorting, up or down; and optional information: size, owner, group, date last modified, date last accessed, permissions, i-node number.

file Determine what kind of information is in a file by consulting the file system index and by reading the file itself.

date Display today's date and time; considerable knowledge of calendric and horological peculiarities; used to set system date and time.

df Report amount of free space on file systems.

du Display a summary of total space occupied by all files in a hierarchy.

quota Display summary of disk usage and limits by user id.

who List presently logged in users, ports and login times; provide optional history of all logins and logouts.

ps Report on active processes; list your own or everybody's processes; and provide optional status information: state and scheduling info, priority, attached terminal, what process is waiting for, and size.

iostat Display statistics about system I/O activity.

tty Display name of your terminal.

pwd Display name of your working directory.

System Accounting Facilities

Commands include:

ac Publish cumulative connect time report; connect time by user or by day and for all users or for selected users.

sa Publish Shell accounting report; give usage information on each command executed, number of times used, total system time, user time and elapsed time, optional averages and percentages, and sorting on various fields; note that the timing information on which the reports are based can be manually cleared or shut off completely.

Workstation Handling Facilities

Commands include:

tset Establish terminal characteristics for the environment.

termcap Facility for customizing terminal parameters by terminal type.

tabs Set tab stops appropriately for specified terminal type.

stty Set up options for optimal control of a terminal; determines half vs. full duplex, carriage return plus line feed versus newline, tabs setting, parity, mapping of upper case to lower, raw versus edited input, and delays for tabs, newlines and carriage returns.

Supported Languages and Related Programs

C Programming Language

The Sun operating system and most of the subsystems are written in C; (for a full description of C, read *The C Programming Language,* Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978); general purpose language designed for structured programming; generalized initialization, block structure, long integers, unions, and explicit type conversions; enhanced to take arbitrary length variable names, and has a substantially faster loader; supports definable data types, which include character, integer, float, double, pointers to all types, functions returning above types, arrays of all types, structures and unions of all types; operations intended to give machine-independent control of full machine facility, including to-memory operations and pointer arithmetic; macro preprocessor for parameterized code and inclusion of standard files; all procedures recursive, with parameters by value; machine-independent pointer manipulation; object code uses full addressing capability of the Sun Workstation; and runtime library gives access to all system facilities.

cc Compile and/or link edit programs in the C language; C compiler has been enhanced to take arbitrary length variable names, allowing readable names and supporting other languages such as Pascal. It allows, for instance, the names from the standard to be used to implement core graphics. A substantially faster loader is also included, as well as tools to aid correction of errors which occur in programs by inserting the error messages as comments into the source code so that the source can easily be edited to remove the errors;

lint Verifier for C programs; reports questionable or nonportable usage such as mismatched data declarations and procedure interfaces, nonportable type conversions, unused variables, unreachable code, no-effect operations, mistyped pointers, and obsolete syntax; full cross-module checking of separately compiled programs.

cb A beautifier for C programs; does proper indentation and placement of braces.

Fortran

f77 A full compiler for the new ANSI Standard Fortran 77; compatible with C and supporting tools at object level; optional source compatibility with Fortran 66; free format source; optional subscript-range checking, detection of uninitialized variables; all widths of arithmetic: 2- and 4-byte integer; 4- and 8-byte real; 8- and 16-byte complex.

Ratfor Ratfor adds rational control structure like C's to Fortran; compound statements; if-else, do, for, while, repeat-until, break, next statements; symbolic constants; file insertion; free format source; translation of relationals like >, >=; produces genuine Fortran to carry away; may be used with F77.

struct Converts ordinary Fortran into Ratfor, a structured dialect usable with *f77*, using statement grouping, if-else, while, for, and repeat-until.

dc Interactive programmable desk calculator; has named storage locations as well as conventional stack for holding integers or programs; unlimited precision decimal arithmetic; appropriate treatment of decimal fractions; arbitrary input and output radices, in particular binary, octal, decimal and hexadecimal; reverse Polish operators:

+ - * /

remainder, power, square root, load, store, duplicate, clear,
print, enter program text, execute.

bc A C-like interactive interface to the desk calculator *dc*; all the capabilities of *dc* with a high-level syntax; arrays and recursive functions; immediate evaluation of expressions and evaluation of functions upon call; arbitrary precision elementary functions exp, sin, cos, atan; go-to-less programming.

Pascal An ANSI Pascal compiler and interpreter system.

pz Execution profiler

pxref Cross-reference program for making cross-referenced listings of Pascal programs.

Macroprocessing Utility

m4 A general purpose stream-oriented macroprocessor that recognizes macros anywhere in text; syntax fits with functional syntax of most higher-level languages; can evaluate integer arithmetic expressions.

Compiler-compilers

yacc An LR(1)-based compiler writing system; during execution of resulting parsers, arbitrary C functions may be called to do code generation or semantic actions; BNF syntax specifications; precedence relations; accepts formally ambiguous grammars with non-BNF resolution rules.

lex Generator of lexical analyzers; converts specification of regular expressions and semantic actions into a recognizing subroutine; arbitrary C functions may be called upon isolation of each lexical token; full regular expression, plus left and right context dependence; resulting lexical analyzers interface cleanly with *yacc* parsers.

Text Editing and Document Formatting Tools

vi The screen-oriented display editor, providing 'what you see is what you get editing' for either line-oriented or full screen terminals. Capabilities include regular expression searching and user-specific settings.

ex The line-oriented parent of *vi*, based on the original *ed* editor; subsumes all functions of *ed*.

awk A pattern scanning and processing language that makes it easy to specify many data transformation and selection operations. Pattern scanning and processing language; searches input for patterns, and performs actions on each line of input that satisfies the pattern; patterns include regular expressions, arithmetic and lexicographic conditions, boolean combinations and ranges of these; data treated as string or numeric as appropriate; can break input into fields; fields are variables; variables and arrays (with non-numeric subscripts); full set of arithmetic operators and control flow; multiple output streams to files and pipes; output can be formatted as desired; multi-line capabilities.

sed A non-interactive stream text editor version of *ed* for processing large files; can perform a sequence of editing operations on each line of an input stream of unbounded length; lines may be selected by address or range of addresses; provides control flow and conditional testing, multiple output streams, and multi-line capability.

ed Interactive context editor; random access to all lines of a file; find lines by number or pattern; patterns may include specified characters, don't care characters, choices among characters, repetitions of these constructs, beginning of line, and end of line; add, delete, change, copy, move or join lines; permute or split contents of a line; replace one or all instances of a pattern within a line; combine or split files; escape to the Shell command language during editing; do any of above operations on every pattern-selected line in a given range; optional encryption for extra security.

ptx Make a permuted (key word in context) index.

spell Look for spelling errors by comparing each word in a document against a 25,000-word list that includes proper names; handles common prefixes and suffixes; collects words to help tailor local spelling lists.

look Search for words in dictionary that begin with specified prefix.

crypt Encrypt and decrypt files for security.

troff and *nroff*

Troff drives a phototypesetter and can be used with appropriate conversion utilities to drive other types of devices; *nroff* drives ASCII terminals of all types; *troff* and *nroff* accept the same input language and are capable of elaborate formatting feats when appropriately programmed; completely definable page format keyed to dynamically planted 'interrupts' at specified lines; maintains several separately definable typesetting environments (for example, one for body text, one for footnotes, and one for unusually elaborate headings); arbitrary number of output pools can be combined at will; macros with substitutable arguments, and macros invocable in mid-line; computation and printing of numerical quantities; conditional execution of macros; tabular layout facility; positions expressible in inches, centimeters, ems, points, machine units or arithmetic combinations thereof; access to character-width computation for unusually difficult layout problems; overstrikes, built-up brackets, horizontal and vertical line drawing; dynamic relative or absolute positioning and size

selection, globally or at the character level; can exploit the characteristics of the terminal being used, for approximating special characters, reverse motions, proportional spacing, etc; typesetter has a vocabulary of several 102-character fonts (4 simultaneously) in 15 sizes; *troff* provides terminal output for rough sampling of final output; *nroff* produces multicolumn output on the workstation (or terminal capable of reverse line feed), or through the *col* postprocessor.

-ms A standardized manuscript layout package of canned requests for use with *nroff* and *troff*; provides page numbers and draft dates, automatically numbered subheads, footnotes, single or double column, paragraphing, display and indentation, and numbered equations.

eqn A mathematical typesetting preprocessor for *troff*; translates easily readable formulas, either in-line or displayed, into detailed typesetting instructions; formulas are written in a style like:

sigma sup 2 ~=~ 1 over N sum from i=1 to N (x sub i - x bar) sup 2

to produce:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i-\bar{x})^2$$

automatic calculation of size changes for subscripts, sub-subscripts, etc.; full vocabulary of Greek letters and special symbols, such as 'gamma', 'GAMMA', 'integral'; automatic calculation of large bracket sizes; vertical 'piling' of formulae for matrices, conditional alternatives, etc.; integrals, sums, etc., with arbitrarily complex limits; diacriticals: dots, double dots, hats, bars, etc.; easily learned by nonprogrammers and mathematical typists.

neqn A version of *eqn* for *nroff*; accepts the same input language; prepares formulas for workstation or terminal display; same facilities as *eqn* within graphical capability of workstation.

tbl A preprocessor for *nroff* and *troff* that translates simple descriptions of table layouts and contents into detailed typesetting instructions; computes column widths; handles left- and right-justified columns, centered columns and decimal-point alignment; places column titles; table entries can be text, which is adjusted to fit; can box all or parts of table.

col Canonicalize files with reverse line feeds for one-pass printing.

deroff Remove all *troff* commands from input.

checknr Check document for possible mismatched opening and closing delimiters and unknown commands.

-me Another package of canned formatting requests.

Information Handling Utilities

Commands include:

sort Sort or merge ASCII files line-by-line; no limit on input size; sort up or down; sort lexicographically or on numeric key; multiple keys located by delimiters or by character position; may sort upper case together with lower into dictionary order; optionally suppress duplicate data.

tsort Topological sort converts a partial order into a total order.

uniq Collapse successive duplicate lines in a file into one line; publish lines that were originally unique, duplicated, or both; may give redundancy count for each line.

tr Do one-to-one character translation according to an arbitrary code; may coalesce selected repeated characters; may delete selected characters.

diff Report line changes, additions and deletions necessary to bring two files into agreement; may produce an editor script to convert one file into another; a variant compares two new versions against one old one.

comm Identify common lines in two sorted files; output in up to 3 columns shows lines present in first file only, present in both, and/or present in second only.

join Combine two files by joining records that have identical keys.

grep Display all lines in a file that satisfy a pattern as used in the editor *ed*; may display all lines that fail to match, count of matches, and first match in each file.

look Binary search in sorted file for lines with specified prefix.

wc Count the lines, 'words' (blank-separated strings) and characters in a file.

Novelties, Games, and Miscellaneous

Commands include:

backgammon
Provides competition against a player of modest ability.

bcd Convert ASCII to card-image form.

cal Display a calendar of specified month and year.

canfield Game of solitaire with betting.

fortune Presents a random fortune cookie on each invocation; limited jar of cookies included.

units Convert amounts between different scales of measurement; knows hundreds of units.

arithmetic
Speed and accuracy test for number facts.

quiz Test your knowledge of Shakespeare, Presidents, capitals, etc.

wump Hunt the wumpus, a thrilling search in a dangerous cave.

hangman Word-guessing game using a dictionary supplied with *spell*.

fish Children's card-guessing game.

Sun Workstation Manuals

System Manager's Manual for the Sun Workstation — Models 100U/150U

Includes system installation, configuration, boot, and maintenance procedures, mail system and networking facility set-up information, and a reference manual for commands and utilities of use to system managers.

System Manager's Manual for the Sun Workstation — Model 120

Includes system installation, configuration, boot, and maintenance procedures, mail system and networking facility set-up information, and a reference manual for commands and utilities of use to system managers.

Beginner's Guide to the Sun Workstation

A tutorial to the Sun system basics, along with user's guides to the Shells, the *mail* and *news* systems, a glossary, and a bibliography for additional UNIX references.

Editing and Text Processing on the Sun Workstation

Contains user's guides and reference material for the text editors and utilities, and document formatting programs and macro packages.

User's Manual for the Sun Workstation

Includes a documentation overview, user-oriented commands, demos, and games.

Programmer's Reference Manual for the Sun Window System

Contains reference material for programmers of applications which use window system facilities.

Programmer's Reference Manual for SunCore

Contains reference material for the SunCore graphics program.

Fortran and Pascal on the Sun Workstation

Fortran I/O libraries, and descriptions of Fortran interfaces to the UNIX system (section 3F). Also includes the Pascal User's Guide.

System Interface Manual for the Sun Workstation

Oriented towards programmers writing C-language programs. Contains descriptions of system calls, subroutines from various libraries, characteristics of special files (devices), and formats of files.

Programming Tools for the Sun Workstation

Information of general interest to anyone using the Sun system to write programs.

System Internals Manual for the Sun Workstation

Contains papers on Sun system internals, including kernel debugging, network implementation, and a device driver tutorial.

PART TWO — USER'S GUIDES

Part Two of the *Beginner's Guide to the Sun Workstation* includes user's guides to the Shells, the mail facility, the network news, a glossary and an annotated bibliography. The user's guides provide details, examples and explanations of many of those commands and facilities presented in Part One.

The Sun system supports two Shells, the C-Shell and the Bourne Shell. These Shells are more or less the same in basic essentials, but they vary a lot in detail. To provide complete, basic descriptions of both Shells, the Shell user's guides in Part Two contain some material that is similar and even repetitious. For specific information on the Shells, see the *csh* and *sh* pages in the *User's Manual for the Sun Workstation*. For detailed information on how to program the Shells, see the *Programming Tools for the Sun Workstation*.

The chapters in Part Two are:

1. Using the C-Shell — Introduces the C-Shell command interpreter and some commonly used Sun system commands.
2. Using the Bourne Shell — Introduces the Version 7 UNIX Shell, the Bourne Shell.
3. Mail User's Guide — Provides details on the electronic mail facilities.
4. Network News User's Guide — Describes what the network news is, how to establish newsgroups, how to read the news, and how to post your own news.
5. Glossary — Provides brief definitions of terms and common commands.
6. Bibliography — Provides an annotated list of Sun system and UNIX reference material.

For additional details on any of the information presented in Part Two, refer to the *User's Manual for the Sun Workstation* and to the *System Interface Manual for the Sun Workstation*.

Table of Contents

1. USING THE C-SHELL

Using the C-Shell† introduces the basics of the C-Shell first to give you a broad understanding its operation. Second, this guide provides more detailed information for learning to use the different C-Shell facilities.

1.1. What is a Shell?

A *Shell* is a program which provides you with interactive access to the operating system via a combined command and programming language. A Shell's primary purpose is to translate command lines typed at the workstation into system actions, such as the invocation of other programs. Because a Shell is a user program, just like any you might write, there is more than one available. The Shell you get when you log in is specified in your password file.

Shell features include control-flow primitives, parameter passing, and variable and string substitution. The Shell supports constructs such as *while*, *if-then-else*, *case*, and *for*. Two-way communication is possible between the Shell and commands. String-valued parameters, typically filenames or flags, may be passed to a command. Commands set a return code that may be used as Shell input.

You can use the Shell to modify the environment in which commands run. You may redirect input and output to files, call processes that communicate through *pipes*, and define a directory searching sequence in the file system to call commands. Commands can be read either from the workstation or from a file, so command procedures can be stored in a file for later use.

A Shell in the Sun operating system acts mostly as a medium through which other programs are invoked. While it has a set of *built-in* functions that it performs directly, most commands cause execution of programs that are external to the Shell. The Shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

The Sun system supports two Shells, the C-Shell (*csh*) developed by William Joy at the University of California at Berkeley and the Bourne Shell (*sh*) developed by S. R. Bourne at Bell Laboratories. After you log in to the Sun system, the C-Shell displays the '%' prompt to indicate it is waiting for input. The Bourne Shell displays the '$' prompt; this is an easy way to tell which Shell your system is running. The C-Shell permits its prompt to be modified and, in this guide, we show the prompt as 'tutorial%'.

1.2. C-Shell Commands

Commands in the Sun system consist of a list of strings or *words*. They are interpreted as a *command name* followed by *arguments*. Thus the command:

```
tutorial% mail sam
```

consists of two words. The first word *mail* names the command to be executed, in this case the *mail* program, which sends messages to other users. The C-Shell uses the name of the command in attempting to execute it for you. It looks in a number of *directories* for a file with the name *mail*, which contains the *mail* program.

The rest of the words of the command are given as *arguments* to the command itself when it is executed. In this case, the argument is *sam*, which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. You can use the *mail* command as follows:

† The material in this chapter is derived from *An Introduction to the C-Shell*, William N. Joy.

```
tutorial% mail sam
Is the project meeting at 3:00?
I may have another appointment.
        Joe
^D
EOT
tutorial%
```

Here Joe sent the user 'sam' a message and ended his message with a ^D,† which sent an end-of-file to the *mail* program. The *mail* program then echoed the characters 'EOT' and transmitted Joe's message. The C-Shell displays the 'tutorial% ' prompt before and after the *mail* command to indicate that it is awaiting input.

After displaying the 'tutorial%' prompt, the C-Shell reads command input from your workstation. When you type a complete command such as *mail sam*, the C-Shell executes the appropriate program, *mail* in this case, with an argument, *sam*. It then hands control over to the *mail* program and waits for *mail* to complete. The *mail* program reads input from your workstation until you signal an end-of-file by typing a ^D. This causes *mail* to complete; the Shell notices that *mail* has completed and displays another 'tutorial%' prompt to signal you that it is ready to read another command from the workstation again.

This is the essential pattern of all interaction with the Sun system through the C-Shell. You type a complete command that the C-Shell executes. When this execution completes, the C-Shell prompts for a new command. If you run the editor for an hour, the C-Shell waits patiently for you to finish editing and obediently prompts you again when you finish.

An example of a useful command you can execute now is the *tset* command, which sets the *erase* and *kill* characters on your terminal — the erase character erases the last character you typed, and the kill character erases the entire line you have entered so far. By default, the erase character is 'DEL' or 'BACKTAB', and the kill character is '^U'. You may prefer to use the backspace (^H) character as your erase character. You can make this change by using the *tset* command with the **–e** option:

```
tutorial% tset –e
Erase set to Ctrl—H
Kill is Ctrl—U
tutorial%
```

This tells the program *tset* to set the erase character to ^H.

1.2.1. Specifying Optional Capabilities with Flag Arguments

While many arguments to commands specify filenames or user names, some arguments called *flag* arguments specify an optional capability of a command that you wish to use. By convention, such flag arguments begin with the character '–' (hyphen). So, to produce a simple list of the files in the current *working directory*, use the *ls* command:

```
tutorial% ls
bin     dead.letter     disk.usage     mbox    misc
tutorial$
```

The flag option **–s** is the size option, which gives the size of each file in blocks of 512 characters; for example:

†The notation ^D is read 'control-D' and means that you should hold down the CONTROL (or CTRL) key while pressing the D key. The shift key is ignored so that '^d' and '^D' are equivalent.

```
tutorial% ls -s
total 9
  1 bin       1 dead.letter     1 disk.usage
  5 mbox      1 misc
tutorial%
```

shows the number of 512-character blocks in each file. Refer to the user's manual for available options for each command. Some commands like the *ls* command have a large number of useful options, while other commands have either no options or only one or two.

1.2.2. C-Shell Metacharacters

The C-Shell has a number of special characters called *metacharacters* that have special functions. In general, most characters which are neither letters nor digits have special meaning to the C-Shell. There is a method of *quoting* that prevents the C-Shell from treating these metacharacters in any special way. This notation is described in *Quoting Away the Metacharacters.*

Metacharacters normally have effect only when the C-Shell is reading your input. You need not worry about placing C-Shell metacharacters in a letter you are sending with *mail* or when you are typing in text or data to some other program, for example. Note that the C-Shell is only reading input when it has prompted with 'tutorial%'. See the *C-Shell Metacharacters Summary* for a complete list with meanings.

1.2.3. Redirecting Output to Files with '>'

Commands that normally read input from or write output to the workstation can instead be executed using a file rather than the workstation for input and output. The *date* command normally displays the current date on your workstation screen because your screen is the default *standard output* for the *date* command:

```
tutorial% date
Thu Aug  4 10:58:37 PDT 1983
tutorial%
```

Suppose you wish to save the current date in a file called *today.* You can *redirect* the standard output of a command through a notation using the metacharacter '>' to the *today* file rather than to the screen:

```
tutorial% date > today
tutorial%
```

This command places the current date and time into the file *today.* Note that the *date* command does not know that its output is going to a file rather than to the workstation. The C-Shell sets up this *redirection* before the command begins executing.

One other thing to note here is that the C-Shell creates the file if it does not exist. The file *today* need not have existed before *date* was executed. And if the file does exist, its previous contents are discarded. You can set the C-Shell option *noclobber* to prevent this from happening accidentally; see the *C-Shell Variables* section on *noclobber.*

The system normally keeps files that you create with '>' permanently. If you wish to create a file which will be removed automatically, begin its name with a '#', the 'scratch' character, to denote that the file will be a scratch file. The system removes such files after a couple of days, or sooner if file space becomes very tight. So if you don't really want to save the output in the example above permanently, use the notation:

```
tutorial% date > #today
tutorial%
```

1.2.4. Redirecting Input from Files with '<'

In the same way that you can redirect the standard output of a command to a file with '>', you can also redirect the *standard input* of a command from a file with the '<' character. This is not often necessary, however, since most commands read from a file whose name is given as an argument. Redirection of input to the *sort* command looks like:

```
tutorial% sort < fruit

apples
bananas
blueberries
lemons
limes
nectarines
oranges
peaches
pears
plums
strawberries

tutorial%
```

where the command reads its input from the file *fruit*. You would more likely let *sort* open the file *fruit* for input itself since this is less typing:

```
tutorial% sort fruit

apples
bananas
  < etc. >
plums
strawberries

tutorial%
```

Note that if you just type *sort* and do not redirect the standard input, as in:

```
tutorial% sort
```

the *sort* program sorts lines from its standard input, the workstation, taking what you type as data, until you type a ^D to indicate an end-of-file. The default standard inpur for programs comes from the workstation keyboard.

1.2.5. Chaining Commands in a Pipeline

In the C-Shell, you can connect the standard output of one command to the standard input of another; that is, you can run the commands in a sequence known as a *pipeline*. For instance, the *ls* command with the **-s** option normally produces a list of the files in your directory with the size of each in 512-character blocks:

```
tutorial% ls -s
total 388
  1 Makefile     286 doc.tbl  40 gloss
 56 mail.all       5 refs.all
tutorial%
```

If you are interested in learning which of your files is largest, you want to sort the list by size. You can look at the many *ls* options to see if there is an option to do this, but you would eventually discover that there is not.

Instead you can use a couple of simple *sort* options and combine them with *ls* with the '|' notation to invoke the *pipe* mechanism to get what you want. Thus, you can pipe *ls* to *sort* by typing:

```
tutorial% ls -s | sort -n
total 388
  1 Makefile     5 refs.all     40 gloss
 56 mail.all 286 doc.tbl
tutorial%
```

This runs the *ls* command with the option **-s** and *pipes* this *ls* output to the *sort* command with the numeric option **-n**. Your list of files is sorted by size with the smallest first. You can then use the **-r** reverse *sort* option and the *head* command in conjunction with the previous command:

```
tutorial% ls -s | sort -n -r | head -3
286 doc.tbl
56 mail.all
40 gloss
tutorial%
```

Here you take a list of your files sorted alphabetically, each with the size in blocks. You *pipe* this to the standard input of *sort* asking it to sort numerically in reverse order, that is, largest first. This output is then *piped* into the *head* command, which shows you the first few lines. In this case you ask *head* for the first three lines. Thus this pipeline gives you the names and sizes of your three largest files.

The C-Shell connects commands separated by '|' characters, and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline normally takes its standard input from the workstation keyboard, and the rightmost normally sends its standard output to the workstation screen. Other examples of pipelines are provided later in the description of foreground and background jobs.

1.2.6. Pathnames and Filenames

Sun system *pathnames* consist of a number of *components* separated by a slash '/'. Each component, except the last, names a directory in which the next component resides, in effect specifying the *path* of directories to follow to gain access to the file. Thus the pathname:

/etc/motd

specifies a file in the directory */etc*, which is a subdirectory of the *root* directory '/'. Within this directory the file named is *motd*, the 'message of the day' file. A pathname that begins with a slash is said to be an *absolute* pathname since it specifies a complete path from the absolute top of the directory hierarchy of the system, the *root*. Pathnames which do not begin with '/' are interpreted as starting in the current *working directory*, which is by default, your *home* directory and which you can change dynamically with the *cd* (change directory) command. Such pathnames are said to be *relative* to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each component of the pathname. If the pathname does not contain any slashes at all, the file is contained in the working directory itself, and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and '.'s (dots). In fact, filenames can have all printing characters except '/'. Remember that it is inconvenient to have most non-alphabetic characters in filenames because many of these characters are metacharacters that have special meaning to the C-Shell. The character '.' (dot) is not a C-Shell metacharacter and often separates the *extension* of a filename from the *base* of the name. Consider the following four related files:

```
data.c data.o data.errs data.output
```

The files share a base portion, 'data', of a name and have different extensions, 'c', 'o', 'errs', and

'output'. The file *data.c* might be the source for a C program, the file *data.o* the corresponding object file, the file *data.errs* the errors resulting from a compilation of the program, and the file *data.output* the output of a run of the program.

1.2.7. Filename Expansion — '*' '?' '[ ]' '~' '{ }'

If you want to refer to all four of these files in a command, use the '*' notation, which the C-Shell expands to match any sequence, including the empty sequence, of characters in a filename. For example, if you use:

```
data.*
```

the C-Shell expands this word into a list of names which begin with 'data' before executing the command to which it is an argument. The names that match *data.** are alphabetically sorted and placed in the *argument list* of the command. Thus the *echo* command and this '*' notation display the four related files as:

```
tutorial% echo data.*
data.c data.errs data.o data.output
tutorial%
```

Note that the names are in sorted order here and a different order than you listed them above. The *echo* command receives four words as arguments, even though you only directly type one word as an argument. *Filename expansion* of the one input word, *data.** generates the four words.

As '*' matches any sequence of characters in a filename, the character '?' matches any *single* character in a filename. So, to echo a line of filenames, type:

```
tutorial% echo ? ?? ???
```

This echoes first those with one-character names, then those with two-character names, and finally those with three- character names. The names of each length are independently sorted.

Another mechanism matches any single character from a sequence of characters between '[ ]'. So to match:

```
data.c data.o
```

in the example above, use:

```
tutorial% echo data.[co]
data.c data.o
tutorial%
```

You can also place two characters around a '–' in the '[ ]' notation to denote a range. Thus to match:

```
chap.1 chap.2 chap.3 chap.4 chap.5
```

if they exist, use:

```
tutorial% echo chap.[1-5]
chap.1 chap.2 chap.3 chap.4 chap.5
tutorial%
```

This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

Note that if a list of arguments to a command, that is, an argument list, contains filename expansion syntax that fails to match any existing filenames, the C-Shell considers this to be an error and displays the diagnostic:

No match.

and does not execute the command.

Another important point is that files with the character '.' (dot) at the beginning of their names are specially treated. Neither '*', '?' nor the '[ ]' mechanism matches it. This special treatment prevents accidental matching of the filenames '.' and '..' in the working directory; these files have special meaning to the system.

Another filename expansion mechanism gives access to the pathname of the home directory of other users. This notation consists of the character '~' (tilde) followed by another user's login name. For instance, *~sam* maps to the pathname */usr/sam* if the home directory for 'sam' is */usr/sam*. Use this notation when you need to gain access to other users' files in directories with different prefix directory names. It's an easier and more reliable method than typing out the entire pathname.

A special case of this notation consists of a '~' alone, such as *~/mbox*. The C-Shell expands this notation into the file *mbox* in your home directory, that is, into */usr/sam/mbox* for your fellow user Sam. This is very useful if Sam uses *cd* to change to another directory and finds a file he wants to copy to his home directory using *cp*. The C-Shell expands '~' into */usr/sam*, Sam's home directory, and copies the file *thatstuff* there if 'sam' types:

```
tutorial% cd programs
tutorial% pwd
/usr/sam/programs
tutorial% cp thatstuff ~
tutorial% cd
tutorial% ls
thatstuff
tutorial%
```

Another form of filename expansion uses the characters '{ }'. Braces specify that the contained strings, separated by a comma (,) are to be consecutively substituted into the containing characters and the results expanded left to right. So, you can abbreviate a set of words that have common parts but cannot be abbreviated by the above mechanisms because they are not files, or because they are the names of files which do not yet exist. Thus:

A{str1,str2,...strn}B

expands to:

Astr1B Astr2B ... AstrnB

The contained strings 'str1,str2...strn' are consecutively substituted into the containing characters 'A' and 'B' and expanded left to right. This expansion occurs before the other filename expansions, and may be applied recursively, that is, nested. The results of each expanded string are sorted separately, left to right order being preserved. If the resulting filenames don't exist, they are created if you don't use other expansion mechanisms. You can use this mechanism to generate arguments which are not filenames, but which have common parts. A typical use below makes subdirectories *docs*, *memos* and *letters* in your home directory:

```
tutorial% mkdir ~/{docs,memos,letters}
tutorial% ls
docs letters memos
tutorial%
```

This mechanism is most useful when the common prefix is longer than in this example, for instance, to list all the directories below without typing each individually, use the '{ }' mechanism:

```
tutorial% ls {/bin/, /usr/ucb/} {pi, whereis}
/bin/pi
/bin/whereis
/usr/ucb/pi
/usr/ucb/whereis
tutorial%
```

See the *C-Shell Metacharacters Summary* for a quick reference list of these characters.

1.2.8. Quoting Away the Metacharacters

Because the C-Shell uses these metacharacters for special purposes, you cannot use them directly as parts of words. If you try to use the *echo* command and '*' as its argument, it doesn't work properly because of the special significance of '*'. Thus the *echo* command does not show the character '*':

```
tutorial% echo *
```

It either echos a sorted list of filenames in the current working directory, or displays the message 'No match.' if there are no files in the working directory.

To place characters that are neither numbers, digits, '/', '.' nor '–' in an argument word to a command, enclose them with single quotation characters ' '. For example, to quote away the special meaning of '*', type:

```
tutorial% echo '*'
*
tutorial%
```

Here the *echo* command displays the '*' character, and ignores any special meaning.

There is one special character '!' that the *history* mechanism uses and that you cannot escape by placing within ' ' characters. Precede '!' and the character ' ' itself by a single ' \' to prevent their special meanings. So to echo ''!', use:

```
tutorial% echo \'\!
'!
tutorial%
```

With these two mechanisms, you can place any printing character into a word which is an argument to a C-Shell command. You can combine the two mechanisms, as in:

```
tutorial% echo \''*'
'*
tutorial%
```

The first '\' escapes the first ' ', and the '*' was enclosed between ' ' characters, so neither retained its special meaning to the C-Shell.

1.2.9. How to Terminate C-Shell Commands

When the C-Shell is waiting for an executing command to complete, there are several ways to stop that command. For instance, if you list all system users with the *cat* command:

```
tutorial% cat /etc/passwd
bugs:nologin:7:10:bug reporting:/usr/bugs:/dev/null
prot:ZSMXceOkDv9hw:11:10:Vic Prot:/usr/prot:/bin/csh
                < etc. >
sam:Iu2nX.wzcjYBo:953:10:Sam Brown:/usr/sam:/bin/csh
rjb:9rYbUmD9JrJvw:954:20:Robert Baker:/usr/rjb:/bin/csh
tutorial%
```

this list is likely to continue scrolling off your screen for several seconds unless you stop it. You

can send an INTERRUPT signal to the *cat* command by typing ^C (or the DEL or RUBOUT key if that's the way your keyboard is set up). Since the *cat* command does not take any precautions to avoid or otherwise handle this signal, the ^C terminates *cat*. The C-Shell notices that *cat* has terminated and prompts you again with 'tutorial%'. If you type a ^C again, the Shell just repeats its prompt since it ignores INTERRUPT signals and continues executing commands rather than terminating like *cat* did. Terminating at this point would otherwise have the effect of logging you out.

Many programs terminate when they get an end-of-file from their standard input. Thus, you terminated this *mail* program in the first example above by typing a ^D, which generates an end-of-file from the standard input. The C-Shell also terminates when it gets an end-of-file and displays 'logout'; the Sun operating system then logs you off the system. Since this means that typing too many ^D's can accidentally log you off, the C-Shell provides a mechanism to prevent this. See the *ignoreeof* description in the *C-Shell Variables* section.

If you redirect a command's standard input from a file, the command normally terminates when it reaches the end of that file. So if you redirect input to the *mail* command with:

```
tutorial% mail sam < doc.text
tutorial%
```

mail terminates without your typing a ^D because it reads to the end-of-file of your file *doc.text*. You can also use the pipe mechanism to pipe the standard output of the *cat* command to the *mail* command:

```
tutorial% cat doc.text | mail sam
tutorial%
```

The *cat* command then writes the text through the pipe to the standard input of the *mail* command. When *cat* completes, it terminates, closing down the pipeline, and the *mail* command receives an end-of-file from it and terminates also. Using a pipe here is more complicated than redirecting input, so use the first form. Typing ^C also stops both of these commands.

Another way to stop a command is to suspend its execution temporarily, with the possibility of continuing execution later. Do this by sending a STOP signal with ^Z. This signal suspends all running commands, but there may be more than one if a pipeline is executing. The C-Shell notices that the command(s) have been suspended, displays 'Stopped' and then prompts for a new command. The previously executing command has been suspended, but is otherwise unaffected by the STOP signal. Any other commands can be executed while the original command remains suspended. You can then continue the suspended command using the *fg* (foreground) command without any arguments. The C-Shell redisplays the command to remind you which command is being continued, and resumes the command execution. The suspension has no effect whatsoever on the execution of the command unless the input files that the suspended command is using have been changed in the meantime. Suspending commands can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows:

```
tutorial% mail peter
You can copy the source from the directory named
^Z
Stopped
tutorial% ls
data.c
data.o
muchstuff
tutorial% jobs
[1]  + Stopped  Mail peter
tutorial% fg
Mail peter
(continue)
data.c. Let's discuss the project later.
^D
EOT
tutorial%
```

In this example you send a message to Peter but forget the name of the file you want to mention. You stop the *mail* command by typing ^Z. When the C-Shell notices that *mail* is suspended, it displays 'Stopped' and prompts for a new command. You then use the *ls* command to find out the name of the file. You then type the *jobs* command to see which command was suspended, *mail peter* in this case. You type the *fg* command to continue *mail* execution. Input to the *mail* program is then continued and ended with a ^D which indicates the end of the message. *Mail* displays EOT for end-of-transmission.

Type ^Z only at the beginning of a line since everything typed on that current line is discarded when a signal is sent from the keyboard. This also happens with the ^C (INTERRUPT) and ^\ (QUIT) signals. See the section on *Running Jobs* for more information on suspending and controlling jobs.

If you write or run programs which are not fully debugged, it may be necessary to stop them somewhat ungracefully. Send them a QUIT signal by typing a ^\. This usually provokes the C-Shell to produce a message like:

```
Quit (Core dumped)
```

indicating that a file *core* has been created containing information about the program's state when it was terminated by the QUIT signal. You can examine this *core* file yourself using a debugger, or forward information to the maintainer of the program telling him where the *core* file is.

When you run background commands, they ignore INTERRUPT and QUIT signals at the workstation. To stop the background commands, you must use the *kill* command. See the *Running Jobs* section for an explanation and examples.

If you want to examine the output of a command without having it zip off the screen as the output of the *cat* command below does:

```
tutorial% cat /etc/passwd
```

use the *more* paging command to display it a page at a time:

```
tutorial% more /etc/passwd
```

The *more* program pauses after each complete screenful and displays '—More—' at which point you can type a space to get another screenful, press RETURN to get another line, or type a 'q' to end *more*. You can also use *more* as a filter through which to pipe the *cat* command:

```
tutorial% cat /etc/passwd | more
```

This works just like the simple *more* command above.

For stopping output of commands not involving *more*, use the ^S key to stop the typeout. The typeout resumes when you type ^Q. Typing ^S and ^Q works well on low-speed terminals, but use *more* if you find it hard to type ^S and ^Q fast enough to paginate the output nicely.

You can also use the ^O flush output character. Typing ^O quickly throws away or 'flushes' all output from the current command until the next input read occurs or until the next Shell prompt. Use ^O to complete a command's execution without your having to suffer through the output on a slow terminal. Typing ^O toggles output flushing on and off.

1.2.10. Changing Shells

If you are running the C-Shell, log in normally and follow the examples provided in the *C-Shell Details* section.

If you are not running the C-Shell when you log in, you are using the Bourne Shell, */bin/sh* or *sh*. In fact, much of the above discussion is applicable to */bin/sh*, as is noted in *Using the Bourne Shell* in Part Two.

If you are not using the C-Shell now, log in and change to the C-Shell with the *csh* command:

```
$ csh
tutorial%
```

To change back to the Bourne Shell, use the *sh* command:

```
tutorial% sh
$
```

1.3. C-Shell Details

This section describes more advanced features and details of the C-Shell.

1.3.1. Starting and Terminating the C-Shell

When you log in, the system starts the C-Shell running in your home directory. The C-Shell begins by reading commands from the *.cshrc* file in this directory. All Shells which you may start during your workstation session read from this file. You can put specific commands there that are described later. For now, however, you do not need this file, and the C-Shell does not complain about its absence.

This first C-Shell is called the *login Shell*. This login Shell reads commands from *.cshrc*, after which it reads commands from a file called *.login*, also in your home directory. This *.login* file contains commands which you wish to execute once each time you log in to the system. A *.login* file looks something like:

```
set ignoreeof
setenv EXINIT 'set noai wrapmargin=8'
set path=(. /usr/ucb /bin /usr/bin)
set mail=(/usr/spool/mail/sam)
```

The first is a *set* command, which the C-Shell interprets directly. *Set* turns on the C-Shell variable *ignoreeof*, which prevents the C-Shell from logging you out if you type ^D. Rather, you use the *logout* command to log off the system.

The *setenv* command sets the value of an *environment variable*, in this case to use the editors *ex* and *vi* without automatic cursor indentation (*set noai*) and with an automatic cursor return or 'wrap' to the left side of the screen eight columns from the right screen edge (*wrapmargin=8*).

The *path* variable defines the search path through which the C-Shell looks for files and programs. The *mail* variable sets the location of the user Sam's system mailbox. When the *mail* program finishes checking for your mail, the C-Shell finishes processing your *.login* file and begins reading commands from the workstation, prompting for each with 'tutorial%'. When you log off with ^D,

the C-Shell displays 'logout' and executes commands from the file *.logout* if it exists in your home directory. After that the C-Shell terminates, and you are logged off the system. You then receive a new login message. In any case, after you type **logout**, the C-Shell is committed to terminating and will take no further input from your keyboard.

1.3.2. C-Shell Variables

The C-Shell maintains a set of *variables*. Each C-Shell variable has an array of zero or more *strings* as its value. Use the *set* command to assign values to C-Shell variables. *Set* has several forms, the most useful of which was already given above in the *.login* example as:

set *name=value*

C-Shell variables may store values which are used in commands later through a substitution mechanism. However, the most commonly used C-Shell variables are those which the C-Shell itself refers to. By changing the values of these variables called *built-in* variables, you can directly affect the C-Shell's behavior.

One of the most important variables is *path*, which contains a sequence of directory names where the C-Shell searches for commands. The *set* command without an argument shows the value of all variables currently defined or 'set' in the C-Shell. You can see what the default value for *path* is by typing the *set* command:

```
tutorial% set
argv    ()
cwd/usr/sam
history 30
home    /usr/sam
mail    (/usr/spool/mail/sam)
path    (. /usr/ucb /bin /usr/bin)
prompt tutorial%
shell   /bin/csh
status  0
term    sun
usersam
tutorial%
```

This output indicates that the variable *path* points to the current directory, symbolized by '.' (dot) and then */usr/ucb*, */bin* and */usr/bin*. Commands developed at the University of California at Berkeley live in */usr/ucb*, while commands developed at Bell Laboratories live in */bin* and */usr/bin*.

A number of locally developed programs on the system live in the directory */usr/local*. If you want all Shells which you invoke to have access to these new programs, place the command:

```
set path=(. /usr/ucb /bin /usr/bin /usr/local)
```

in your file *.login* in your home directory. Try doing this, and then log out and back in. Now type the *set* command again to see that the value assigned to *path* has changed:

```
tutorial% set
argv    ()
cwd/usr/sam
    < etc.>
path    (. /usr/ucb /bin /usr/bin /usr/local)
    < etc. >
usersam
tutorial%
```

Be aware that the C-Shell initially examines each directory in your path and determines which commands are contained there. Except for the current directory '.', which the C-Shell treats specially, this means that if commands are added to a directory in your search path after you have started the C-Shell, the C-Shell will not necessarily find them. If you wish to use a command which has been added in this way, use the *rehash* command to recompute the C-Shell's internal hash table of command locations so that it finds the newly added command:

```
tutorial% rehash
tutorial%
```

If you do not run *rehash*, the hashing algorithm may tell the C-Shell that the command wasn't in that directory when the hash table was computed. Since the C-Shell has to look in the current directory '.' on each command, it can always find commands in the current working directory.

Other useful built-in variables are the variable *home*, which shows your home directory, *cwd*, which contains your current working directory, and the variable *ignoreeof*, which can be set in your *.login* file to tell the C-Shell not to exit when it receives an end-of-file from your keyboard. The variable *ignoreeof* is one of several variables that only have the value *set* or *unset*. Thus, to set this variable you simply type the following in your *.login* file:

```
set ignoreeof
```

If you type the character ^D accidentally, you get the message 'Use "logout" to logout.' Then use the *logout* command to terminate the login Shell.

To unset the *ignoreeof* option temporarily for that login session, type:

```
tutorial% unset ignoreeof
tutorial%
```

These actions do not give the *ignoreeof* variable a value, but none is desired or required.

Another useful built-in C-Shell variable is the *noclobber* variable. If you use the metasyntax:

```
> filename
```

to redirect the standard output of a command, you normally overwrite and destroy the previous contents (if any) of the named file, here *filename*. Because of this, you may accidentally overwrite a valuable file. If you want to prevent the C-Shell from overwriting files in this way, add the *noclobber* variable to your *.login* file:

```
set noclobber
```

Then try to redirect *date* into the *today* file:

```
tutorial% date > today
today: File exists.
tutorial%
```

Noclobber warns you if *today* already exists. If you really want to overwrite the contents of *today*, you can use the '!' character to force the action:

```
tutorial% date >! today
tutorial%
```

The '>!' is a special metasyntax indicating that clobbering the file is allowed. The space between the '!' and the filename *today* is critical here, as '!today' would be an invocation of the *history* mechanism, and have a totally different effect.

See *csh* in the user's manual for *set* variables.

1.3.3. The History Mechanism

The C-Shell can maintain a *history list* into which it places the words of previous commands. You can reuse these commands or words from them to form new commands. You can also use *history* to repeat previous commands or to correct minor typing mistakes.

To use the *history* mechanism, edit your *.cshrc* file to contain:

```
set history=30
```

Then type:

```
tutorial% source .cshrc
tutorial%
```

to have the change take effect. Then after you have typed several commands, you see that typing just *history* shows the contents of the history list:

```
tutorial% history
    1   ls
    2   mkdir misc
    4   cd misc
    5   vi tut.memo
    6   spell tut.memo > mem.sp &
    7   pwd
    8   cd supplements/tutorial
    9   history > hist.list
tutorial%
```

You can use the numbers given with the history events to refer to previous events, which are difficult to refer to using the contextual mechanisms introduced above. For example, to reuse command number 8, type simply:

```
tutorial% !8
cd supplements/tutorial
tutorial% pwd
/usr/sam/supplements/tutorial
tutorial%
```

Figure 1 gives a sample session involving typical history mechanism commands.

```
tutorial% cat bug.c
main()

{
    printf("hello);
}
tutorial% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
tutorial% ed !$
ed bug.c
29
4s/);/"&/p
      printf("hello");
w
30
q
tutorial% !c
cc bug.c
tutorial% a.out
hello tutorial% !e
ed bug.c
30
4s/lo/lo\\n/p
      printf("hello\n");
w
32
q
tutorial% !c -o bug
cc bug.c -o bug
tutorial% size a.out bug
a.out: 2784+ 364+ 1028 = 4176b = 0x1050b
bug: 2784+ 364+ 1028 = 4176b = 0x1050b
tutorial% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill      3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill      3932 Dec 19 09:42 bug
tutorial% bug
hello
tutorial% num bug.c | spp
spp: Command not found.
tutorial% ^spp^ssp
num bug.c | ssp
   1    main()
   3    {
   4        printf("hello\n");
   5    }
tutorial% !! | lpr
num bug.c | ssp | lpr
tutorial%
```

Figure 1. Sample *history* Use

In this example you have a very simple C program, with a bug (or two) in the file *bug.c*, which you *cat* out on your workstation. You then try to run the C compiler on it, referring to the file again as '!$', meaning the last argument to the previous command. Here the '!' is the history mechanism invocation metacharacter, and the '$' stands for the last argument, by analogy to '$' in the editor, which stands for the end of the line. The C-Shell echoes the command, as it would have been typed without use of the history mechanism, and then executes it. The compilation yields error diagnostics, so you now run the editor on the file you are trying to compile, fix the bug, and run the C compiler again. This time you refer to this command simply as '!c'. This repeats the last command which started with the letter 'c'. If you have used other commands starting with 'c' recently, you have to say '!cc'. Typing '!cc:p' prints the last command starting with 'cc' without executing it, so you can check which previous command you want.

After this recompilation, you run the resulting *a.out* file, and then note that there still is a bug, and run the editor again. After fixing the program you run the C compiler again, but tack onto the command an extra '–o bug' telling the compiler to place the resultant binary in the file *bug* rather than *a.out*. In general, you can use the history mechanism anywhere in the formation of new commands, and you can place other characters before and after the substituted commands.

You then run the *size* command to see how large the binary program images you have created are, and then an *ls –l* command with the same argument list, denoting the argument list '!*'. Finally, you run the program *bug* to see that its output is indeed correct.

To make a numbered listing of the program, you run the *num* command on the file *bug.c*. To remove blank lines in *num* output, you run the output through the filter *ssp*, but misspell it as 'spp'. To correct this you use a C-Shell substitute, placing the old text and new text between '^' characters. Note that the symbols '↑' and '^' are the same thing. This is similar to the substitute command in the editor. Finally, you repeat the same command with '!!', but send its output to the line printer.

There are other mechanisms available for repeating commands. *History* displays a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, ways to select arguments to include in a new command. Refer to the C-Shell pages in the *User's Manual for the Sun Workstation* for a complete description.

1.3.4. The Alias Mechanism

The *alias* mechanism substitutes one string for another before the C-Shell executes it. Use the C-Shell's *alias* mechanism to supply default arguments to commands, or to perform transformations on commands and their arguments. The *alias* facility is similar to a macro facility. Some of the features obtained by aliasing can also be obtained using C-Shell command files, but these take place in another instance of the Shell and cannot directly affect the current Shell's environment or involve commands such as *cd*, which must be done in the current Shell.

As an example, suppose that there is a new version of the *mail* program called *newmail* on the system. You would rather use it than the standard *mail* program, which is called *mail*. If you place the C-Shell command:

```
alias mail newmail
```

in your *.cshrc* file, the C-Shell transforms an input line of the form

```
tutorial% mail sam
```

into a call on *newmail*. Suppose you want the command *ls* to always show which list entries are subdirectories, which are files, and which are symbolic links to other directories, that is to always use a **–F** option. Put the following alias in your *.cshrc* file:

```
alias ls ls –F
```

If you then type *ls*, you actually use *ls –F*:

```
tutorial% ls
bin/     lint.mss mbox    misc/
supplements/
```

You can also use:

```
alias lf ls -F
```

to create a new command syntax *lf* that calls *ls -F*. So, using this alias on the home directory of 'sam', you get:

```
tutorial% lf ~sam
bin/     dead.letter  mbox    misc/
supplements/
tutorial%
```

or a list of files and directories in */usr/sam* with the **-F** indications of '/' for a directory.

Thus the *alias* mechanism creates short names for commands, provides default arguments, and defines new short commands in terms of other commands. You can also define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the *history* mechanism. To call an *ls* command after each *cd* (change directory) command, use the alias:

```
alias cd 'cd \!* ; ls '
```

Enclose the entire alias definition in ' ' characters to prevent most filename expansions from occurring and the character ';' from being recognized as a metacharacter. The '!' here is escaped with a '\' to make it apply to the argument list of the aliased *cd* command itself rather than searching the history list for a previous command. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command without giving an error if there aren't any arguments. The ';' separating commands indicates that one command is to be done and then the next. Remember to run the *source* command on your *.cshrc* file to have any changes you make take effect:

```
tutorial% source .cshrc
tutorial%
```

When you use this *alias*, it looks like:

```
tutorial% cd /usr/games
abuse       bcd     cribbage    mille    scifi     worms
adventure   boggle  fish        monop    snake     wump
arithmetic
                    < etc. >
tutorial%
```

This *cd* command not only changes directories, here to */usr/games*, but it also lists all the games available.

Similarly to define a command which looks up its first argument in the password file, put in your *.cshrc* file:

```
alias whois 'grep \!^ /etc/passwd'
```

Then, when you type *whois* plus a username, the C-Shell calls *grep* to look in the */etc/passwd* file:

```
tutorial% whois alice
alice:IBkUBlXESfxGY:55:20:Alice Smith:/usr/alice:/bin/csh
tutorial%
```

Use the *unalias* command at the 'tutorial%' prompt to remove aliases temporarily for that Shell session.

Warning: the C-Shell currently reads the *.cshrc* file each time it starts up, so if you place a large number of commands there, the C-Shell will tend to start slowly. Try to limit the number of aliases to 10 or 15.

1.3.5. The Redirection Notation '>>' and '>&'

In addition to the standard output, commands also have a *diagnostic output*, which is normally directed to the workstation screen even when the standard output is redirected to a file or a pipe. You occasionally may want to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces, you can type:

```
tutorial% command >& file
tutorial%
```

The '>&' here tells the C-Shell to route both the diagnostic output and the standard output into *file*. Use the command form *command* **>&!** *file* if *noclobber* is set and *file* already exists to overwrite *file*.

Similarly you can route both standard and diagnostic output through the pipe to the line printer *lpr* by typing:

```
tutorial% command |& lpr
tutorial%
```

Finally, to place standard output at the end of an existing file, type:

```
tutorial% command >> file
tutorial%
```

If *noclobber* is set, an error message 'file: No such file or directory.' results if *file* for example, does not exist; otherwise the C-Shell creates the named file. The form *command* **>>!** *file* eliminates the error condition if *file* does not exist when *noclobber* is set.

1.3.6. Running Jobs in the Background, Foreground, or Suspended

When one or more commands is typed together as a pipeline or as a sequence of commands separated by semicolons, the C-Shell creates a single *job* consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the C-Shell creates a job. Some lines that create jobs (one per line) are

```
tutorial% sort < data
tutorial% ls -s | sort -n | head -5
tutorial% mail harold
```

The job is started as a *background* job if you type the metacharacter '&' at the end of the commands. This means that the C-Shell does not wait for the command to complete but immediately prompts for another. The job runs *in the background* at the same time that the C-Shell continues to read and execute normal jobs, called *foreground* jobs. Thus, to redirect the output of the *du* program to a file called *disk.usage*, for instance, type:

```
tutorial% du > disk.usage &
[1] 503
tutorial%
```

Du reports on the disk usage of your working directory, as well as any directories below it. This command sequence puts the output into the file *disk.usage*, and the Shell returns immediately with a prompt for the next command without waiting for *du* to finish. The *du* program continues executing in the background until it finishes, even though you can type and execute more commands in the meantime. When a background job terminates, the C-Shell displays a message just

before the next prompt telling you that the job has completed. In the following example, the *du* job finishes sometime during the execution of the *mail* command and its completion is reported just before the prompt after the *mail* job is finished.

```
tutorial% du > disk.usage &
[1] 503
tutorial% mail sam
How do you know when a background job is finished?
^D
EOT
[1] - Done  du > disk.usage
tutorial%
```

If the job did not terminate normally, the 'Done' message might say something else like 'Stopped.' If you want the terminations of background jobs to be reported at the time they occur, which may interrupt the output of other foreground jobs, you can set the *notify* variable in your *.cshrc* file. In the previous example this would mean that the 'Done' message might have come right in the middle of the message to Sam. The STOP, INTERRUPT, or QUIT signals mentioned earlier, when typed on the keyboard, do not affect background jobs.

Until they terminate, jobs are recorded in a table inside the C-Shell. The C-Shell remembers the command names, arguments and the *process numbers* of all commands in the job in this table as well as the working directory where the job was started. Each job in the table is either running *in the foreground* with the C-Shell waiting for it to terminate, running *in the background*, or *suspended*. Only one job can be running in the foreground at one time, but several jobs can be suspended or running in the background at once. As each job is started, it is assigned a small identifying number called the *job number* which you can use later to refer to the job in the commands described below. Job numbers remain the same until the job terminates and then are reused.

Before prompting you for another command, the C-Shell displays the background job's number, as well as the process numbers of all its top-level commands. For example, if you run the following command in the background by typing the ampersand '&' character at the end:

```
tutorial% ls -s | sort -n > file.list &
[2] 2034 2035
tutorial%
```

the *ls* program runs with the **-s** option, pipes this output to the *sort* program with the **-n** option, which puts its output into the file *file.list*. The '&' at the end of the line starts these two programs together as a background job. After starting the job, the C-Shell displays the job number in brackets, 2 in this case, followed by the process number of each program started in the job. Then the C-Shell immediately prompts for a new command, leaving the job running simultaneously.

As mentioned in the *How to Terminate C-Shell Commands* section, typing ^Z suspends currently running foreground jobs. You can suspend a background job by using the *stop* command described below. When jobs are suspended, they merely stop any further progress until started again, either in the foreground or the background. The C-Shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. Stopping a foreground job looks like:

```
tutorial% du > disk.usage
^Z
Stopped
tutorial%
```

The C-Shell displays the 'Stopped' message when it notices that the *du* program stopped. For background jobs, using the *stop* command, it is:

```
tutorial% sort disk.usage &
[1] 2345
tutorial% stop %1
[1] + Stopped (signal) sort disk.usage
tutorial%
```

The '(signal)' indicates that the job has been stopped by an indirect signal, as opposed to being stopped by ^Z. Suspending background jobs can be very useful when you need to temporarily change what you are doing, that is, execute other commands, and then return to the suspended job. Also, you can suspend foreground jobs and then continue them as background jobs using the *bg* command. Thus, you can continue other work and stop waiting for the foreground job to finish. For example:

```
tutorial% du > disk.usage
^Z
Stopped
tutorial% bg
[1] du > disk.usage &
tutorial%
```

starts *du* in the foreground, stops it before it finishes, then continues it in the background so you can execute more foreground commands. The *bg* command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the ^Z STOP signal. The combination of the STOP signal and the *bg* command changes a foreground job into a background job. This is especially helpful when a foreground job ends up taking longer than you expected, and you wish you had started it in the background in the beginning.

All *job control* commands can take an argument that identifies a particular job. Begin all job name arguments with the character '%', since some of the job control commands also accept process numbers, as displayed by the *ps* command. If you do not specify a job, a job control command uses the default job, that is, the *current* job. This current job is identified by a '+' in the output of the *jobs* command, which shows which jobs you have. When only one job is stopped or running in the background as is the usual case, it is always the current job, so no argument is needed. If a job is stopped while running in the foreground, it becomes the current job and the existing current job becomes the previous job, identified by a '-' in the *jobs* output. When the current job terminates, the previous job becomes the current job. When given, the argument is either '%-', which indicates the previous job, '%#' where # is the job number, '%pref' where pref is some unique prefix of the command name and arguments of one of the jobs, or '%?' followed by some string found in only one of the jobs.

The *jobs* command displays the table of jobs, giving the job number, status ('Stopped' or 'Running') and command name for each background or suspended job:

```
tutorial% du > disk.usage &
[1] 3398
tutorial% ls -s | sort -n > myfile &
[2] 3405
tutorial% mail bill
^Z
Stopped
tutorial% jobs
[1] - Running    du > disk.usage
[2]   Running    ls -s | sort -n > myfile
[3] + Stopped    mail bill
tutorial%
```

With the -l option the process numbers are also displayed:

```
tutorial% jobs -l
[1] 3398 - Running     du > disk.usage
[2] 3405  Running      ls -s | sort -n > myfile
[3] + Stopped    mail bill
tutorial%
```

Continuing with the same series, you can use the *fg* command to bring the *ls* job to the foreground:

```
tutorial% fg %ls
ls -s | sort -n > myfile
tutorial% more myfile
```

The *fg* (foreground) command runs a suspended or background job in the foreground. It restarts a previously suspended job or changes a background job to run in the foreground, allowing signals or input from the workstation. In the above example you use *fg* to change the *ls* job from the background to the foreground since you want to wait for it to finish before looking at its output file.

The *stop* command suspends a background job.

```
tutorial% stop %1
[1]- Stopped (signal) du > disk.usage
tutorial%
```

You can use the *kill* command to terminate a background or suspended job immediately. In addition to job numbers, you can give it process numbers as arguments, as displayed by *ps*. Thus, in the above example, you can terminate the running *du* command with *kill*:

```
tutorial% kill %1
[1]  Terminated  du > disk.usage
tutorial%
```

The *notify* command (not the *set* command variable mentioned earlier) reports termination of a specific job at the time the job finishes instead of waiting for the next prompt.

If a job running in the background tries to read input from the workstation, it is automatically stopped. You can give input to the job, when you return such a job to the foreground. If desired, you can return the job to the background until it requests input again. This is illustrated in the following sequence where the *s* (substitute) command in the text editor might take a long time:

```
tutorial% ed bigfile
120000
1,$s/thisword/thatword/
^Z
Stopped
tutorial% bg
[1] ed bigfile &
tutorial%
 . . . some foreground commands
[1] Stopped (tty input)ed bigfile
tutorial% fg
ed bigfile
w
120000
q
tutorial%
```

After you called the *s* command, you stopped the *ed* job with ^Z, and then put it in the

background using *bg*. Sometime later when the *s* command was finished, *ed* tried to read another command and was stopped because jobs in the background cannot read from the workstation. Typing the *fg* command returned the *ed* job to the foreground where it could once again accept commands from the terminal.

To stop all background jobs when they are about to write output to the workstation, use the *stty* (set terminal output) command:

```
tutorial% stty tostop
tutorial%
```

This prevents messages from background jobs from interrupting foreground job output so you can run a job in the background without losing workstation output. You can also use it for interactive programs that sometimes have long periods without interaction. Thus each time a background job prompts for more input, it stops before the prompt. You run the job in the foreground using *fg*, give it more input and, if necessary, stop and return it to the background. This *stty* command is a good thing to put in your *.login* file if you do not like output from background jobs interrupting your work. It can also reduce the need for redirecting the output of background jobs if the output is not very big:

```
tutorial% stty tostop
tutorial% wc hugefile &
[1] 10387
tutorial% ed text
. . . some time later
q
[1] Stopped (tty output)     wc hugefile
tutorial% fg %wc
wc hugefile
  13371   30123   302577
tutorial% stty -tostop
tutorial%
```

Thus after some time the *wc* command, which counts the lines, words and characters in a file, had one line of output. When it tried to write this to the workstation it stopped. By restarting it in the foreground, it writes on the workstation exactly when you are ready to look at its output. *Stty tostop* allows *bg* job output to go to the workstation. Programs which attempt to change the mode of a terminal will also stop, whether or not **tostop** is set, as it would be very unpleasant to have a background job change the state of a terminal.

Since the *jobs* command only displays jobs started in the currently executing C-Shell, it knows nothing about background jobs started in other login sessions or within C-Shell files. Use the *ps* command in this case to find out about background jobs not started in the current C-Shell.

1.3.7. The C-Shell's Working Directory

As mentioned in *Filenames and Pathnames*, the C-Shell is always in a particular *working directory*. The *cd* (change directory) command changes the working directory of the C-Shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The *mkdir* command (make directory) creates a new directory. The *pwd* (print working directory) command reports the absolute pathname of the working directory of the C-Shell, that is, the directory you are located in. Thus in the example below, Sam creates and moves to the directory *newpaper*, where he might place a group of related files:

```
tutorial% pwd
/usr/sam
tutorial% mkdir newpaper
tutorial% cd newpaper
tutorial% pwd
/usr/sam/newpaper
tutorial%
```

No matter where you move to in a directory hierarchy, you can return to your home login directory by typing *cd* without any arguments:

```
tutorial% cd
tutorial% pwd
/usr/sam
tutorial%
```

The name '..' (dot dot) always means the directory above the current one in the hierarchy, so to change the C-Shell's working directory to the one directly above the current one, use:

```
tutorial% cd ..
tutorial% pwd
/usr
tutorial%
```

The name '..' can be used in any pathname. To change to the directory *programs* contained in the directory above the current one, type:

```
tutorial% cd ../programs
```

If you have several directories for different projects under, say, your home directory, use this shorthand notation to switch easily between them.

The C-Shell always remembers the pathname of its current working directory in the variable *cwd*. You can also request that the C-Shell remember the previous directory when you change to a new working directory. If you use the *pushd* (push directory) command in place of the *cd* command, the C-Shell saves the name of the current working directory on a *directory stack* before changing to the new one. You can see this directory stack at any time by typing the *directories* command *dirs*:

```
tutorial% pushd newpaper/references
~/newpaper/references ~
tutorial% pushd /usr/lib/tmac
/usr/lib/tmac ~/newpaper/references ~
tutorial% dirs
/usr/lib/tmac ~/newpaper/references ~
tutorial%
tutorial%
```

The list is displayed in a horizontal line, reading left to right, with a tilde (~) as shorthand for your home directory — in this case */usr/sam*. The directory stack is displayed whenever there is more than one entry in it and it changes. *Dirs* is usually faster and more informative than *pwd* since it shows the current working directory as well as any other directories remembered in the stack. The *pushd* command without any arguments alternates the current directory with the first directory in the list.

The *popd* (pop directory) command without an argument returns you to the directory you were in prior to the current one, discarding the current directory from the stack and forgetting it. Typing *popd* several times in a series takes you backward through the directories you had changed to with the *pushd* command:

```
tutorial% popd
~/newpaper/references ~
tutorial% popd
~
tutorial%
```

There are other options to *pushd* and *popd* to manipulate the contents of the directory stack and to change to directories not at the top of the stack; see the *csh* user's manual page for details.

Regardless of what directory changes you make, the C-Shell remembers the working directory in which each job was started. It warns you if you try to restart a job in the foreground which has a different working directory than the current C-Shell working directory. Thus if you start a background job, change the C-Shell's working directory, and then run the background job in the foreground, the C-Shell warns you that the working directory of the currently running foreground job is different from that of the C-Shell:

```
tutorial% dirs -l
/doc/sam
tutorial% ed myproject
tutorial% dirs
~/myproject
tutorial% ed prog.c
1143
^Z
Stopped
tutorial% cd ..
tutorial% ls
myproject
textfile
tutorial% fg
ed prog.c (wd: ~/myproject)
```

The C-Shell warns you that the working directory of the *ed* program is *~/myproject*, not the current working directory of *doc/sam*. The *ed* job was still in */doc/sam/project* even though the C-Shell had changed to */doc/sam*.

You get a similar warning when such a foreground job terminates or is suspended using the ^Z STOP signal, since returning to the C-Shell again implies a change of working directory.

```
tutorial% fg
ed prog.c (wd: ~/myproject)
. . . after some editing
q
(wd now: ~)
tutorial%
```

These messages are sometimes confusing if you use programs that change their own working directories. The C-Shell only remembers which directory a job is started in, and assumes the job stays there. The *jobs* -l option displays the working directory of suspended or background jobs when it is different from the current working directory of the C-Shell.

1.3.8. Useful Built-in Commands

This section describes several of the more useful built-in C-Shell commands. For a complete list, see *csh* in the user's manual.

The *echo* command plays back its argument list to the screen. It is often used in *Shell scripts* or as an interactive command to see what filename expansions will produce. We saw this earlier in *Filename Expansion*. To determine the effect of a command such as *rm*, type:

```
tutorial% echo rm [ca]*
rm a aA aB aC c1 c10 ... c5 c6 c7 c8 c9
tutorial%
```

Using *echo* here is a good way to learn the effects of the metacharacters without affecting any files or directories.

The *limit* command restricts the use of resources. With no arguments, it displays the current limitations:

```
tutorial% limit
filesize        unlimited
datasize        1984 kbytes
stacksize       512 kbytes
coredumpsize    unlimited
memoryuse       unlimited
tutorial%
```

Limits can be set, for instance:

```
tutorial% limit coredumpsize 128k
tutorial%
```

for the current login session. Most reasonable units abbreviations work. See the *csh* user's manual page for more details.

Use *repeat* to repeat a command several times. To make four copies of the file *data* in the file *four*, type:

```
tutorial% repeat 4 cat data >> four
tutorial%
```

The *setenv* command sets variables in the environment. For example:

```
setenv TERM sun
```

or

```
setenv TERM adm3a
```

This sets the value of the environment variable TERM to 'sun' or 'adm3a', depending on your terminal. *Unsetenv* removes variables from the environment.

The *printenv* user program displays the environment. It might show:

```
tutorial% printenv
HOME=/usr/lori
SHELL=/bin/csh
PATH=.:/:/bin:/usr/bin:/usr/lori/bin:/usr/local:/usr/local/bin:/usr/ucb:/etc:/usr/hosts
TERM=sun
USER=lori
EXINIT=set noai wrapmargin=8
tutorial%
```

Use the *source* command noted before to force the current Shell to read commands from a file:

```
tutorial% source .cshrc
tutorial%
```

Running *source* on the *.cshrc* file makes any changes you made take effect immediately, that is, before the next time you login.

Use the *time* command to time a command no matter how much CPU time it takes:

```
tutorial% time cp /etc/rc /usr/sam/rc
0.0u 0.1s 0:01 8% 2+ 1k 3+ 2io 1pf+ 0w
tutorial% time wc /etc/rc /usr/sam/rc
    52     178    1347 /etc/rc
    52     178    1347 /usr/sam/rc
   104     356    2694 total
0.1u 0.1s 0:00 13% 3+ 3k 5+ 3io 7pf+ 0w
tutorial%
```

The first indicates that the *cp* command used a negligible amount of user time (u) and about 1/10th of a second system time (s); the elapsed time was 1 second (0:01); 8% of CPU cycles over period when active; there was an average memory usage of 2k bytes of program space and 1k bytes of data space over the CPU time involved (2+ 1k); the program did three disk reads and two disk writes (3+ 2io), took one page fault and was not swapped (1pf+ 0w). The word count command *wc* on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active, *wc* used an average of 13 percent of the available CPU cycles of the machine.

1.4. Programming the C-Shell

You can use the C-Shell to read and execute C-Shell command *scripts*. C-Shell scripts are files that contain a group of *csh* commands.

A command script may be interpreted by saying:

```
tutorial% csh script ...
```

where *script* is the name of the file containing a group of *csh* commands and '...' is replaced by a sequence of arguments. The C-Shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other C-Shell variables.

You can make the file executable with the *chmod* command:

```
tutorial% chmod 755 script
tutorial%
```

If you place a C-Shell comment at the beginning of the Shell script as well, that is, begin the file with a '#' character, then */bin/csh* is automatically invoked to execute *script* when you type:

```
tutorial% script
```

If the file does not begin with a '#' then the standard Shell */bin/sh* executes it. In this way, you can convert your older shell scripts to use *csh*.

1.4.1. Variable Substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism known as *variable substitution* is done on these words. Keyed by the character '$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script echoes the current value of the variable *argv* to the output of the shell script. You must have *argv* set at this point; otherwise, it is an error.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set*. This is the fundamental mechanism used

for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. Thus

```
tutorial% set argv=(a b c)
tutorial% echo $?argv
1
tutorial% echo $#argv
3
tutorial% unset argv
tutorial% echo $?argv
0
tutorial% echo $argv
Undefined variable: argv.
tutorial%
```

It is also possible to access the components of a variable which has several values. Thus

```
$argv[1]
```

gives the first component of *argv* or in the example above 'a'. Similarly

```
$argv[$#argv]
```

would give 'c', and

```
$argv[1-2]
```

would give 'a b'. Other notations useful in shell scripts are:

```
$n
```

where *n* is an integer as a shorthand for

```
$argv[n]
```

the *n*th parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current Shell. Since this process number is unique in the system it can be used to generate unique temporary filenames. The form

```
$<
```

is quite special. It is replaced by the next line of input read from the Shell's standard input, not the Shell script it is reading. This is useful for writing Shell scripts that are interactive, reading commands from the workstation or terminal, or even writing a Shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo -n 'yes or no?'
set a=($<)
```

writes out the prompt 'yes or no?' without a newline and then reads the answer into the variable

'a'. In this case '$#a' would be '0' if either a blank line or end-of-file (^D) is typed.

There is one minor difference between '$*n*' and '$argv[*n*]'. The form '$argv[*n*]' will yield an error if *n* is not in the range '1-$#argv', while '$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form 'n-'; if there are less than *n* components of the given variable then no words are substituted. A range of the form 'm-n' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

See *Programming Tools for the Sun Workstation* for more information on programming the C-Shell.

1.5. C-Shell Metacharacter Summary

The following table lists the special *csh* and Sun system characters. A number of these characters also have special meaning in expressions. See the *csh* manual section for a complete list.

Syntactic Metacharacters

;	separates commands to be executed sequentially
|	separates commands in a pipeline
( )	brackets expressions and variable values
&	follows commands to be executed without waiting for completion

Filename Metacharacters

/	separates components of a file's pathname
?	expansion character matching any single character
*	expansion character matching any sequence of characters
[ ]	expansion sequence matching any single character from a set
~	used at the beginning of a filename to indicate home directories
{ }	used to specify groups of arguments with common parts

Quoting Metacharacters

\	prevents meta-meaning of following single character
'	prevents meta-meaning of a group of characters
"	like ', but allows variable and command expansion

Input/output Metacharacters

<	indicates redirected input
>	indicates redirected output

Expansion/substitution Metacharacters

$	indicates variable substitution
!	indicates history substitution
:	precedes substitution modifiers
^	used in special forms of history substitution
`	indicates command substitution

Other Metacharacters

#	begins scratch file names; indicates Shell comments
-	prefixes option (flag) arguments to commands
%	prefixes job name specifications

2. USING THE BOURNE SHELL

Using the Bourne Shell† describes the UNIX system version 7 Shell called the Bourne Shell (*sh*). The design of the Bourne Shell, here referred to simply as the 'Shell,' is based in part on the original UNIX Shell and the PWB/UNIX Shell. Similarities also exist with the command interpreters of the Cambridge Multiple Access System and of CTSS.

See the *User's Manual for the Sun Workstation* for more details on *sh*. Also see the references in the appendix for more information.

2.1. What is a Shell?

A *Shell* is both a command language and a programming language that provides an interface to the operating system and interprets commands which you type. The Shell's primary purpose is to translate command lines typed at the workstation into system actions, such as the invocation of other programs. Because the Shell is a user program, just like any you might write, there is more than one available. The Shell you get when you log in is specified in your password file entry field, which contains the pathname to it.

The Shell's features include variables, control-flow primitives, parameters passing, subroutines, interrupt handling, and string substitution. The Shell supports control structures such as *while*, *if-then-else*, *case*, and *for*. Two-way communication is possible between the Shell and commands. String-valued parameters, typically filenames or flags, may be passed to a command. Commands set a return code that may be used as Shell input.

You can use the Shell to modify the environment in which commands run by redirecting input and output to files, by calling processes that communicate through *pipes*, and by defining a directory searching sequence in the file system to call commands. Commands can be read either from the workstation or from a file, so command procedures can be stored for later use.

A Shell in the Sun operating system acts mostly as a medium through which other programs are invoked. The Shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

The Sun system supports two shells, the Bourne Shell and the C-Shell, which are available as the programs *csh* and *sh* respectively. The Shell you get when you log in is specified in a field in your password file entry, which contains the pathname to the Shell to be used. If your system is running the Bourne Shell, it displays the '$' prompt. The C-Shell displays the '%' prompt.

2.2. Logging In

The Shell is a program that runs automatically when you log in to the Sun system. When you log in, the Shell reads any commands from the file *.profile*, if you have such a file in your login directory before reading any commands from the workstation.

It reads each command that you type and interprets what you've asked for. The Shell expands any file-matching metacharacters you use. If you redirect the standard input and output, or the diagnostic output, the Shell handles that too. The Shell examines the command you type in, calls up the command from wherever it lives, and passes all the arguments to that program and starts it up.

The Shell is the interface between you the user, and the Sun system utility programs. Because it is just an ordinary program, you can use it by typing the command *sh* followed by an argument, which is the name of a file containing Sun system commands. See *Programming the Shell* for some simple examples.

---

†The material in this chapter is derived from *An Introduction to the Bourne Shell*, S. I. Bourne.

2.3. Changing the Shell Prompt

The Shell displays a prompt before reading a command. By default this prompt is '$ '. Or you can change it to the string 'yesdear', for example:

```
PS1=yesdear
```

If a newline is typed and further input is needed, the Shell displays the prompt '> '. Sometimes mistyping a quote mark causes this. If it is unexpected, an INTERRUPT (^C) returns the Shell to read another command. You can change this '>' prompt by saying, for example:

```
PS2=more
```

2.4. Simple Shell Commands

Simple Shell commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. These arguments can be flag arguments or filenames. For example, to print the names of users logged in, type the *who* command:

```
$ who
lori     console Jul 26 07:40
$
```

This shows user 'lori' is logged in on the console.

To show a detailed list of files in the current directory, use the *ls* command with the –l option:

```
$ ls -l
total 1064
-rw-r--r--  1 lori       181    Jul 25 17:14 Makefile
-rw-r--r--  1 lori    460654    Jul 24 17:16 doc.cat
                < etc. >
-rw-r--r--  1 lori        67    Jul 23 12:31 tabs
-rw-r--r--  1 lori     22980    Jul 17 15:42 uucp
$
```

The argument –l tells *ls* to print status information, size and the creation date for each file.

2.4.1. Background Commands

To execute a command, the Shell normally creates a new *process* and waits for it to finish. You may run a command in the *background* without waiting for it to finish. For example, to put a call to the C compiler in the background, you type the *cc* command line with an ampersand '&' at the end of the line:

```
$ cc pgm.c &
321
$
```

This compiles the file *pgm.c*. The trailing & is the operator that instructs the Shell not to wait for the command to finish. To keep track of a process, the Shell reports its process number, 321 in this case, following its creation. You can obtain a list of currently active processes using the *ps* (process status) command.

2.4.2. Input/Output Redirection

Most commands produce output on the standard output, that is your workstation or terminal. You can send this output to a file instead of the standard output by typing, for example:

```
$ ls -l > file.list
$
```

The Shell interprets the notation *>file.list* and does not pass it as an argument to *ls*. If *file.list* does not exist, the Shell creates it; otherwise the output from *ls* replaces the original contents of *file.list*. You can append output to a file with the '>>' notation:

```
$ ls -l >> file.list
$
```

A second *ls -l* directory contents listing is appended to the first in *file.list*. In this case *file.list* is also created if it does not already exist.

A command can take the standard input from a file instead of the workstation by typing the '<' redirection character as in:

```
$ wc < file.list
    30    234   1546 file.list
$
```

The command *wc* reads its standard input, in this case redirected from *file.list*, and displays the number of characters, words and lines found. If only the number of lines is required, use the -l option:

```
$ wc -l < file.list
    30 file.list
$
```

2.4.3. Pipelines and Filters

Two or more commands connected by the pipe operator '|' form a *pipeline*. A filter is a command that reads its standard input, transforms it in some way, and displays the result as output. Pipelines and filters are often used together.

Use the pipe operator, '|' to connect the standard output of one command to the standard input of another. In the example above, the two commands *ls -l > file.list* and *wc -l < file.list* were run to get one desired result. You can run both together; that is, you can process the *ls* output with the *wc* command by typing:

```
$ ls -l | wc
    19    146    963
$
```

Here, the output of the *ls* command is piped as input to *wc*. Two commands connected in this way constitute a *pipeline*, and the overall effect is the same as:

```
$ ls -l > file.list; wc < file.list
$
```

except that no *file.list* is used. Instead a pipe connects the two processes, which are run in parallel. Pipes are unidirectional and synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* when the pipe is full.

A *filter* command transforms its standard input in some way. One such filter, *grep*, selects from its input those lines that contain some specified string. For example, to list those lines, if any, from *ls* that contain the string 'all', type:

```
$ ls | grep all
mail.all
news.all
refs.all
shell.all
summ.all
$
```

Grep takes the output of *ls* and searches for the string 'all'. Another useful filter is *sort*, which orders or 'sorts' a file in several different ways. For example, to display an alphabetically sorted list of names in the file *name.list*, type:

```
$ sort name.list

Dan
Joe
Mary
Mike
Susie

$
```

Read more about *sort* in the user's manual.

A pipeline may consist of more than two commands. You can pipe *ls* to *grep* and then to *wc*, for example:

```
$ ls | grep all | wc -l
      5
$
```

to display the number of filenames in the current directory containing the string 'all' as in the example above.

2.4.4. Filename Expansion '*' '?' '[ ]'

As described in the *Simple Shell Commands* section, the first word of a command is the name of the command to be executed. Other words on the command line are arguments to that commands. Many commands accept arguments that are filenames. For example, use the *ls* command with the -l option to display information relating to the file *main.c*:

```
$ ls -l main.c
-rw-r--r-- 1 lori     136783 Jul 26 11:10 main.c
$
```

The file *main.c* is the argument to the *ls* command.

The Shell provides a mechanism for generating a list of filenames that match a pattern. For example, to generate, as arguments to *ls*, all filenames in the current directory that end in '.all', type:

```
$ ls -l *.all
-rw-r--r-- 1 lori      57022   Jul 23 12:15 mail.all
-rw-r--r-- 1 lori      25643   Jul 23 12:15 news.all
-rw-r--r-- 1 lori       4965   Jul 25 16:50 refs.all
-rw-r--r-- 1 lori     136783   Jul 26 11:10 shell.all
-rw-r--r-- 1 lori      34121   Jul 25 17:14 summ.all
$
```

The Shell expands '*' to match any string including the null string, that is, all the files whose names end with '.all' in the working directory. In general *patterns* are specified as follows:

*	Matches any string of characters including the null string.
?	Matches any single character.
[ ]	Matches any one of the characters enclosed. A pair of characters separated by a minus matches any character lexically between the pair.

To match all names in the current directory beginning with one of the letters *a* through *z*, use:

```
$ ls -l [a-z]*
mailref:
total 121
-rw-r--r-- 1 lori    460654   Jul 24 17:16 doc.cat
-rw-r--r-- 1 lori    292152   Jul 24 16:47 doc.tbl
             < etc. >
-rw-r--r-- 1 lori     58334   Jul 20 17:42 mail.all
-rw-r--r-- 1 lori       201   Jun 30 23:18 mail0.nr
$
```

The Shell expands the '[a-z]*' argument to all files beginning with any lower-case letter. Be careful; this may display a very long list of files.

The '?' character matches all names that consist of a single character in the directory. So you can use the *ls* command with '?' on */usr/fred/test* and say:

```
$ ls /usr/fred/test/?
/usr/fred/test/a
/usr/fred/test/b
$
```

to match the files *a* and *b*. If no filename is found that matches the pattern, the pattern is passed unchanged as an argument, and 'No match.' is displayed.

This filename expansion notation saves typing and provides name selection according to some pattern. It makes finding a file easy. For example, to find and display the names of all *core* files in subdirectories of */usr/fred*, say:

```
$ echo /usr/fred/*/core
/usr/fred/misc/core  /usr/fred/test/core
$
```

The '*' finds the files *core* in subdirectories *misc* and *test*. As you saw before, *echo* is a standard command that displays its arguments, separated by blanks. This last search feature however, can be expensive, requiring a scan of all subdirectories of */usr/fred*.

There is one exception to the general rules given for patterns. The character '.' ('dot') at the start of a filename must be explicitly matched. For instance, using '*' to match any character does not match a '.' at the beginning of a filename:

```
$ echo *
Makefile doc.tbl file shell.all uucp
$
```

Instead, it echos all filenames in the current directory not beginning with '.' even if the current directory is your home directory, which contains your *.profile* file. To match the '.' character at the beginning of a filename, type:

```
$ echo .*
$
```

This echos all those filenames that begin with '.' including the names '.' and '..' which mean 'the current directory' and 'the parent directory' respectively.

See the *Shell Metacharacter Summary* section for a quick reference list of the metacharacters.

2.4.5. Quoting the Metacharacters with ' ' and ' \'

Characters that have a special meaning to the Shell, such as ' < >', '*', '?' | '&', are called *metacharacters*. Any character preceded by a ' \' is *quoted* or *escaped* and loses its special meaning, if any. The ' \' is called the *escape* character and elided so that to echo a single '?', type:

```
$ echo \?
?
$
```

And to echo a single '<', type:

```
$ echo \<
<
$
```

The ' \' prevents the Shell from using the special meanings of '?' and '<'.

To continue long commands over more than one line, the sequence *newline* is ignored.

The ' \' is convenient for quoting single characters, but when more than one character needs quoting, it is clumsy and error prone. Enclose the string of characters between single quotes. For example, to quote a series of four asterisks so that they lose their special matching capability, type:

```
$ echo xx'****'xx
xx****xx
$
```

The quoted string may not contain a single quote, but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using double quotes prevents interpretation of some but not all meta-characters.

2.5. Programming the Shell

Use the Shell to read and execute commands contained in a file. A file containing commands is called a *command procedure* or *Shell procedure.*

A simple Shell procedure uses the *echo* command:

```
$ cat > welcome
echo Good morning!
^D
$
```

To call the Shell to read commands from *welcome*, use the *sh* command:

```
$ sh welcome
Good morning!
$
```

In general, this format looks like:

$ **sh** *file [args ...]*

in which *sh* calls commands from *file.*

You can either execute a Shell procedure with *sh* or make the procedure executable. The *chmod* (change mode) command can make a file *readable, writable* and *executable.*

For example, to make *welcome* executable, type:

```
$ chmod +x welcome
```

Following this, the command:

```
$ welcome
Good morning!
$
```

is equivalent to:

```
$ sh welcome
Good morning!
$
```

You can also use the *chmod* command in the following format:

```
$ chmod 755 welcome
$ welcome
Good morning!
$
```

which has the same effect as the first. See the user's manual for details on the *chmod* command.

An executable command is noted with an asterisk '*' in an *ls -F* directory listing. So if you were to list the directory contents now, it would show among other things:

```
$ ls -F
  < etc. >
welcome*
$
```

The Shell also has the capability to define a named variable and assign a value to it. The simplest way to set a Shell variable is to use an *assignment statement*:

```
variable=value
```

You can then use the assigned value by preceding the name of the variable with a dollar sign:

```
$variable
```

Now make a file called **bliss** which contains the assignment statements:

```
food=bread
drink=wine
person=thou
echo $food, $drink, $person ....Ah!
```

Make *bliss* executable with *chmod*, and execute it:

```
$ chmod 755 bliss
$ bliss
bread, wine, thou ....Ah!
$
```

As another example, make a string-searching file called *lsg* (for 'ls-grep') that contains:

```
ls | grep $1
```

Make it executable with *chmod* and use it to find filenames containing the string 'all' in the current directory:

```
$ chmod 755 lsg
$ lsg 'all'
allnames
allx
mail.all
$
```

You can thus use Shell procedures and programs interchangeably.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#**. You can refer to the name of the file being executed as **$0**.

There are some predefined Shell variables, some of which are modifiable and some of which are read-only. You have already seen:

HOME — Set to the user's home directory,

PATH — Set of directories that the Shell searches in order to find commands.

PS1 — Primary prompt string, here the '$'.

You can also set a Shell variable from the output of a command. For example:

```
$ now=`date`
$ echo $now
Fri Aug 12 08:23:12 PST 1983
$
```

Note that the characters surrounding the command are grave accents, not apostrophes.

To set a Shell variable equal to a value contained in a file, use:

```
todo=`cat plan`
```

This calls the *cat* command with the argument *plan*, where the *plan* file contains:

```
Eat breakfast.
Go to work.
```

The resulting value of '$todo' is:

```
$ echo $todo
Eat breakfast. Go to work.
$
```

A different type of Shell variable is one which is passed to the Shell procedures when it is called. These arguments called *positional parameters* and are referred to by number, **$1, $2, ...** Consider the following simple Shell procedure:

```
$ cat > reverse
echo $6 $5 $4 $3 $2 $1
^D
$ chmod 755 reverse
$ reverse do re mi fa so la
la so fa mi re do
$
```

You can also use the flow control programming constructs *if.. else, while.. do,* and others to control the action taken by the procedure. See the *Programming Tools for the Sun Workstation* for more information on programming the Shell.

2.6. Shell Metacharacter Summary

Syntactic Metacharacters

|	pipe symbol
;	command separator
&	background commands
( )	command grouping
<	input redirection
>	output creation

Filename Expansion Metacharacters

*	match any character(s) including none
?	match any single character
[...]	match any of the enclosed characters

Substitution Metacharacters

${...}	substitute Shell variable
`...`	substitute command output

Quoting Metacharacters

\	quote the next character
'...'	quote the enclosed characters except for '
"..."	quote the enclosed characters except for $ ` \ "

3. MAIL USER'S GUIDE

The *Mail User's Guide*† describes how to use the *mail* program to send and receive messages. It assumes you are familiar with the C-Shell, a text editor like *vi* or *ex*, and some of the common Sun system commands. If you are not, read the *Introduction to the Sun System* in Part One of this manual. For additional details on Sun system commands, consult the *User's Manual for the Sun Workstation*. Set-up information for the *mail* facility is in the *System Manager's Manual for the Sun Workstation*.

Mail provides a communication facility for sending and receiving mail among users on the same host, users on different hosts linked to your local network, and users linked to the ARPANET, UUCP, and Berkeley networks. You use a set of editing commands to manipulate messages, and to define and send mail to names which label user groups.

Briefly, here is how *mail* handles messages: *mail* divides incoming mail into its constituent messages so you can deal with them in any order you please. The mail system collects the messages for you from other people in a file, called your *system mailbox*. When you login, the system notifies you if there are any messages waiting in this system mailbox. When you read your mail using *mail*, it reads your system mailbox and separates that file into the individual messages that have been sent to you. You can then read, reply to, delete, or save these messages. Each message is marked with its author, the date sent, and the message subject among other things.

3.1. Sending Mail

The *mail* command has three ways to send mail, depending on where your recipient has a login account. If he has a host machine linked to your local network, use the method described in *Sending Mail on the Local Network*. If he has a login account on the same host as yours, use the method described in *Sending Mail on the Same Host*. If your recipient logs in on a machine connected to yours by UUCP, use the method described in *Sending Mail on the Network*.

3.1.1. Sending Mail on the Local Network

To send mail to users on other hosts linked to yours on the local network, use the *mail* command followed by the login name and the host machine name of your recipient; for example, type:

```
tutorial% mail joe@venus
Is the meeting planned for this afternoon?
^D
EOT
tutorial%
```

This sends mail to 'joe', which is the login name of the person you're sending mail to. Joe's hostname is 'venus', which is where he has an account and logs in. End your message with a ^D (an EOT) at the beginning of a line. *Mail* echoes EOT and returns you to the Shell.

The message 'joe' reads consists of the message you typed, preceded by several header lines telling who sent the message (your login name), the date and time it was sent, and various other details.

If, while you are composing the message you decide that you do not wish to send it after all, you can abort the letter with your current interrupt character (the default is ^C). Typing a single INTERRUPT causes *mail* to display:

```
(Interrupt -- one more to kill letter)
```

Typing a second INTERRUPT saves your partial letter in the file *dead.letter* in your home directory and aborts the letter. Once you have sent mail to someone, there is no way to undo the act, so be careful.

---

†The material in this guide is derived from the *Mail Reference Manual*, Kurt Shoens, Craig Leres.

3.1.2. Sending Mail on the Same Host

Sending mail to a plain login name without a hostname sends mail to that person assuming he has an account on your machine. To send a message to 'roger', who has a login account on your host, type:

```
tutorial% mail roger
Let's play tennis this afternoon.
^D
EOT
tutorial%
```

End your message with a ^D (an EOT) at the beginning of a line as before. *Mail* echoes EOT and returns you to the Shell. Later, the user 'roger' to whom you sent mail receives the message:

```
You have mail.
```

or

```
You have new mail.
```

or

```
New mail has arrived.
```

to tell him he has a message waiting.

If you want to send the same message to several other people, you can list their login names on the command line. For instance:

```
tutorial% mail john marty david
Meeting at three o'clock.
Please be on time.
^D
EOT
tutorial%
```

sends the reminder to John, Marty, and David.

3.1.3. Sending Mail on the Network

If your recipient logs in on a machine connected to yours by the telephone line network called *uucp* (unix to unix copy), you must know the list of machines through which your message must travel to arrive at his site. So, if his machine is directly connected to yours, you can send mail to him using his hostname, the '!' or 'bang' character, and his login name. If you are using the C-Shell as our example shows, you must also escape the special '!' character with a backslash ' \'. For example:

```
tutorial% mail venus\!joe
```

sends mail to the user 'joe' whose *uucp* hostname is 'venus'. The general syntax is:

```
tutorial% mail hostname\!name
```

If your message must go through an intermediate system first, use the syntax:

```
tutorial% mail intermediate\!hostname\!username
```

Ask your system administrator about a 'map' of all the systems in the network connected to your site.

There are several advanced facilities to learn about in the section *Sending Mail with Tilde Escapes.*

3.2. Reading Your Mail

If, when you log in, you see the message:

```
Last login: Tue Aug 6 13:42:21 on console
Sun UNIX 4.2 (Berkeley beta release) (GENERIC) #8: Oct 23 13:45:52 PDT 1983
You have mail.
tutorial%
```

you can read the mail by typing simply:

```
tutorial% mail
```

Mail responds by displaying its version number and date and then listing the messages you have waiting. Then it prompts you and waits for your command. The messages are assigned numbers starting with 1, so you refer to the messages with these numbers. Consider the following:

```
Mail version 2.17 12/26/82. Type ? for help.
"/usr/spool/lori": 2 messages 2 new
>N 1 steve      Wed Sep 21 09:21  12/277 "Weekly Meeting"
  N 2 wendy     Tue Sep 20 22:55
```

Mail keeps track of which messages are 'new', that is, have been sent since you last read your mail, and which messages are 'read', that is, that you have read. New messages have an **N** next to them in the header listing and old, but unread messages have a **U** next to them. *Mail* keeps track of new/old and read/unread messages by putting a header field called 'Status' into your messages.

To look at a specific message, use the *print* command, which may be abbreviated to simply *p*. You can examine the first message above by typing:

```
& print 1
Message 1:
From steve Tue Aug  2 10:28:33 1983
Date: 2 Aug 83 10:28:27 PDT (Tue)
From: steve (Steve Smith)
Subject: Weekly Meeting
Message-Id: <8308021728.AA05502@sun.uucp>
Received: by sun.uucp (3.320/3.14)
  id AA05502; 2 Aug 83 10:28:27 PDT (Tue)
To: henry
Status: R

Meeting at three o'clock.
Please be on time.
&
```

Many *mail* commands that operate on messages take a message number as an argument like the *print* command. These commands have a notion of a current message. When you enter the *mail* program, the current message is initially the first new message. Thus, you can often omit the message number and use simple *p* to display the current message:

```
& p
```

Another shorthand method is to display a message by simply giving its message number. To display the first new message, say:

```
& 1
```

Frequently, it is useful to read the messages in your mailbox in order, one after another. You can read the next message in *mail* by simply typing a newline. As a special case, you can type a newline as your first command to *mail* to type the first unread message.

3.3. Replying to Mail

If you wish to send a reply immediately after reading a message, you can do so with either the *reply* or the *Reply* command. There is a distinct difference between the two commands.

Sometimes you will receive a message that has been sent to several people and wish to reply only to the person who sent it. In this case, use a capital *R* to reply to the sender only. *Reply*, like *type*, takes a message number as an argument. To reply to the sender and to everyone who received the original message, use the *reply* command. Type in your reply, followed by a ^D at the beginning of a line, as before. *Mail* displays EOT and the ampersand prompt to indicate its readiness to accept another command. In our example, if you wish to reply to the first message after reading it, use the command:

```
& reply
To: steve
Subject: Re: Weekly Meeting
```

and enter your letter. You are now in the message collection mode, and *mail* gathers up your message up to a ^D. Note that it copies the header from the original message to make it easy to recognize. If there are other header fields in the message, that information will also be used. For example, if your letter has a 'To:' header listing several recipients, *mail* sends your reply to those same people as well. Similarly, if the original message contained a 'Cc:' (carbon copies to) field, your reply goes to those users, too. *Mail* is careful, though, not to send the message to you, even if you appear in the 'To:' or 'Cc:' field, unless you ask to be included explicitly. More on this later.

When you use the *reply* command to respond to a letter, there is a problem of figuring out the names of the users in the 'To:' and 'Cc:' lists 'relative to the current machine.' If the original letter was sent to you by someone on the local machine, then this problem does not exist, but if the message came from a remote machine, the problem must be dealt with. *Mail* uses a heuristic to build the correct name for each user relative to the local machine. So, when you reply to remote mail, the names in the 'To:' and 'Cc:' lists may change somewhat. After typing in your letter, your correspondence looks like:

```
& reply
To: steve
Subject: Weekly Meeting

Thanks for the reminder
^D
EOT
&
```

The *reply* command is especially useful for sustaining extended conversations over the message system, with other 'listening' users receiving copies of the conversation. Abbreviate the *reply* command to *r* once you get the hang of things.

If you wish, while reading your mail, to send a message to someone, but not as a reply to one of your messages, send the message directly with the *mail* command, which takes as arguments the names of the recipients you wish to send to. For example, to send a message to Wendy, type:

```
& mail wendy
Your presentation yesterday was very informative.
EOT
&
```

Normally, each message you read is saved in your mailbox or *mbox* file in your login directory at the time you leave *mail*. Often, however, you will not want to save a particular message you have received because it is only of passing interest. To avoid saving a message in *mbox*, delete it by typing:

& delete 1

This makes message 1 from Steve disappear altogether, along with its number. Abbreviate the *delete* command to *d*.

3.4. Customizing Your Mail

There are several ways to customize the *mail* facility. If you have accounts on several systems and want to direct your mail to a single account, you need a *.forward* file in the home directories of those other accounts. You can also use the *set* and *alias* commands to tailor many *mail* features to your personal uses.

3.4.1. Forwarding Your Mail from Other Accounts

If there is a large number of systems at your site or if you are linked to USENET, you may have accounts on several machines. To forward your mail to a single account where *mail* will notify you that you have mail, create a *.forward* file in the home directories of all the accounts from which you want mail forwarded. For example, you may have an account on 'venus' where other users sometimes send you mail, but you usually log in to your account on 'tutorial'. Create a *.forward* file on your account on 'venus' and add:

sam@tutorial

substituting your login name for 'sam.' Mail sent to your account on 'venus' will then be forwarded to your account on 'tutorial'.

Another way of forwarding mail is to use *aliases*. See *Streamlining Your Mail with 'alias'* for details.

3.4.2. Setting Your Options with 'set'

Set has two forms, one for setting a *binary* option and one for a *valued* option. Binary options are either *on* or *off*. A complete list of *mail* options appears in *Additional Features*.

As a C-Shell (*csh*) user, you will be notified when new mail arrives if you inform the C-Shell of the location of your system mailbox in the directory */usr/spool/mail* in a file with your login name. If your login name is 'karen', you can make *csh* notify you of new mail by including the following line in your *.login* file in your home directory:

set mail=/usr/spool/mail/karen

Another useful option is *ask*, which informs *mail* that each time you send a message, you want *mail* to prompt you for a subject header to be included in the message. To set the *ask* option in *mail*, type:

& set ask

Another useful option is *hold*, which tells *mail* to keep your messages in the system mailbox instead of moving them to your *mbox* file in your home directory as it normally does when you leave *mail*.

Use valued options to adapt *mail* to your personal use. For example, the *SHELL* option tells which Shell you like to use, and is specified by:

& set SHELL=/bin/sh

Note that no spaces are allowed in 'SHELL=/bin/sh.' A default Shell is used if none is specified.

Another important valued option is *crt*, which prevents long messages from flying by too quickly for you to read them. Setting the *crt* option sends any message longer than a given number of lines through the paging program *more*. Try setting this option for your workstation as:

& set crt=34

(or 24 if you are using a terminal) to paginate messages that will not fit on your screen. *More* displays a screenful of information, then shows --MORE--. Type a space to see the next screenful or RETURN to see the next line. It is not necessary to type in these *set* commands each time you run *mail*. See the discussion of *.mailrc* in *Streamlining Your Mail with 'alias'*.

3.4.3. Streamlining Your Mail with 'alias'

Mail has an *alias* adaptation, similar to the C-Shell. An alias is simply a name which stands for one or more real user names. When you send mail to an alias, you are really sending it to the list of real users associated with it. For example, define an alias for the members of a project, so that you can send mail to the whole project by sending mail to just a single name. Suppose that the users in a project are named Dan, Rick, Tom, and John. Define an alias 'project' for their project group by typing:

```
& alias project dan rick tom john
```

You can then send mail to all of them by typing:

```
& mail project
    < etc. >
^D
EOT
tutorial%
```

Use *alias* to provide a convenient name for someone whose user name is inconvenient. For example, if a user named Margaret Cunningham has the login name 'margaret', you can set an alias with:

```
& alias mar margaret
```

so that you can send mail to the shorter name, 'mar'.

The *alias* and *set* commands let you customize *mail*, but you wouldn't want to have to retype them each time you enter *mail*. To make them more convenient to use, put the *set* options and aliases you want in your *.mailrc* file in your home directory. For example, a *.mailrc* file can look like:

```
set ask nosave crt=24 SHELL=/bin/csh
alias project dan rick tom john
```

What happens here is that *mail* always looks for two files when it is invoked. It first reads a system-wide file */usr/lib/Mail.rc*, then your user-specific file, *.mailrc* in your home directory. The system administrator at your site maintains the system-wide file, which contains *set* commands that are applicable to all users of the system.

The *mail* delivery system *sendmail* provides a system-wide *aliases* file called */usr/lib/aliases* which provides a more efficient way to keep a large database of *mail* aliases. For details, refer to the *Sendmail Installation and Operation Guide* in the *System Manager's Manual for the Sun Workstation*.

3.5. More on Reading Mail

You have seen that you can invoke *mail* with command line arguments that name people to send the message to, or with no arguments to read mail. Specifying the **-f** flag on the command line causes *mail* to read messages from a file other than your system mailbox. For example, if you have a collection of messages in the file *letters*, use *mail* to read them with:

```
tutorial% mail -f letters
"letters": 3 messages
& n
→1 lori Tue Jul 26 14:55 11/102 "Company Party"
  <etc.>
& 1
From lori Tue Jul 26 14:55:09 1983
Date: 26 Jul 83 14:55:04 PDT (Tue)
From: lori (Lori Rosen)
Subject: Company Party
Message-Id: <8307262155.AA04140@sun.uucp>
Received: by sun.uucp (3.320/3.14)
  id AA04140; 26 Jul 83 14:55:04 PDT (Tue)
To: steve
Status: R
Please make plans for the company
party in October.
?
```

You can use all the *mail* commands described in this manual to examine, modify, or delete messages from your *letters* file, which is rewritten when you leave *mail* with the *quit* command described below.

Since mail that you read is saved in the file *mbox* in your home directory by default, you can read it by using simply:

```
tutorial% mail -f
"/usr/lori/mbox": 7 messages
&
```

Normally, messages that you examine using the *print* command are saved in the *mbox* file in your home directory if you leave *mail* with the *quit* command described below. If you wish to retain a message in your system mailbox you can use the *preserve* command to tell *mail* to leave it there. *Preserve* accepts a list of message numbers, just like *print* and may be abbreviated to *pre*.

Messages in your system mailbox that you do not examine are normally retained in your system mailbox automatically. If you wish to have such a message saved in *mbox* without reading it, use the *mbox* command to save it. For instance:

```
& mbox 2
& quit
Saved 1 message in mbox
Held 1 message in /usr/spool/mail/lori
tutorial%
```

saves the second message in *mbox* when you give the *quit* command. The *mbox* command is also the way to direct messages to your *mbox* file if you have set the *hold* option described above. You can abbreviate *mbox* to *mb*.

3.6. Quitting Mail

When you have read all the interesting messages, you can finish the mail session with the *quit* command. *Quit* does different things with the messages depending on whether you read a message, skipped over it, or deleted it:

- Messages that you read but didn't delete are appended to *mbox* in your home directory.

- Messages which you simply skipped over and didn't delete are kept in the system mailbox so you can read them the next time you use *mail*.
- Deleted messages are gone forever.

Note that you can retrieve deleted messages with the *u* (undelete) command as long as you are still in *mail*. Once you quit the *mail* program however, deleted messages are irretrievable.

To quit *mail*, type:

```
& quit
Saved 1 message in mbox
Held 1 message in /usr/spool/mail/lori
tutorial%
```

You can abbreviate the *quit* command to *q*.

If you wish for some reason to leave *mail* quickly without altering either your system mailbox or *mbox*, type:

```
& x
tutorial%
```

(short for *exit*) which immediately returns you to the Shell without changing anything.

If, instead, you want to execute a Shell command without leaving *mail*, type the command preceded by an exclamation point, just as in the text editor. For instance:

```
& !date
Tue Jul 26 12:32:12 PDT 1983
!
&
```

displays the current date without leaving *mail*.

Finally, type a question mark '?' to get a brief summary of the *mail* command abbreviations.

3.7. Collecting Groups of Messages in Folders

Mail includes a simple facility for maintaining groups of messages together in folders.

To use the folder facility, put a line like:

```
set folder=letters
```

in your *.mailrc* file to indicate where your folder directory should be kept. Each folder of messages is a single file, and all of your folders are kept in that directory. If, as in the example above, your folder directory does not begin with a '/,' *mail* assumes that your folder directory is to be found starting from your home directory. Thus, if your home directory is */usr/person* the above example puts your folder directory in */usr/person/letters*.

Anywhere a filename is expected, you can use a folder name, preceded with '+ '. For example, to put a message into a folder with the *save* command, use:

```
& save +records
"/usr/username/letters/records" [New file] 13/272
&
```

This saves the current message in the *records* folder. If the *records* folder does not yet exist, it is created. Note that messages which are saved with the *save* command are automatically removed from your system mailbox.

To copy a message into a folder without removing it from your system mailbox, type:

```
& copy +records
"/usr/username/lettters/records" [Appended] 11/282
&
```

This copies the current message into the *records* folder and leaves a copy in your system mailbox. The 'appended' message indicates that you already have a message in the *records* folder. *Copy* is identical in all other respects to *save*.

Use the *folder* command to read the contents of a different folder. For example:

```
& folder +records
```

causes *mail* to close the file it is currently reading (as if you had typed **q**) and direct its attention to the contents of the *records* folder. All of the commands that you can use on your system mailbox are also applicable to folders, including *print*, *delete*, and *reply*. To inquire which folder you are currently editing, use simply:

```
& folder
```

To list your current set of folders, use the **folders** command:

```
& folders
```

If you want to start reading one of your folders, use the **-f** option described previously. For example:

```
tutorial% mail -f +records
```

reads your *records* folder without looking at your system mailbox.

3.8. Sending Mail with Tilde Escapes

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, display the message, execute a Shell command, or do some other auxiliary function. *Mail* provides these capabilities through 'tilde escapes,' which consist of a tilde '~' at the beginning of a line, followed by a single character which indicates the function to be performed.

3.8.1. Displaying the Message Text with '~p'

If you are typing in a long message and want to to display the text of the message so far, use the '~p' escape:

```
tutorial% mail lori
A very long message ...
~p
-------
Message contains:
To: lori
Subject: Company Party
A very long message ...
(continue)
```

The '~p' displays a line of dashes, the recipients of your message, and the text of the message so far. Since *mail* requires two consecutive ^C's (RUBOUT's, INTERRUPTs, DELETE's) to abort a letter, you can use a single ^C to abort the output of '~p' or any other '~' escape without killing your letter.

3.8.2. Editing a Message — '~e' and '~v'

If you are dissatisfied with the message as it stands, you can use a text editor on it. To use the default line editor *ex*, type:

~e

The '~e' escape copies the message into a temporary file for editing. After modifying the message to your satisfaction, write it out and quit the editor. Your screen displays:

(continue)

after which you may continue typing text which will be appended to your message, or type ^D at end the message. *Mail* provides a standard text editor, but you can override this default by setting the valued option 'EDITOR' to something else. For example, you might prefer to use the *vi* display editor from Berkeley instead of the *ex* editor. Edit your *.login* file to contain:

setenv EDITOR /usr/ucb/vi

Mail also defines a default visual display editor, *vi*. To use *vi* for editing your current message, use the escape:

~v

The '~v' escape works like '~e', but invokes the visual rather than the text editor. If it does not suit you, you can set the valued option 'VISUAL' to the pathname of a different editor.

If you want to include the contents of some file in your message, the escape:

~r datafile
"datafile" 14/512

appends the named file, *datafile* in this case, to your current message. If the read is successful, the number of lines and characters appended to your message is printed, after which you may continue appending text. *Mail* complains if the file doesn't exist or can't be read; for example:

tutorial% **mail lori**
Subject: Company Party
~r budget
budget: No such file or directory.

The filename may contain Shell metacharacters like '*' and '?', which are expanded according to the conventions of your Shell.

3.8.3. Using the 'dead.letter' File with '~d'

Similar to the '~r' escape is the '~d' escape which reads the file *dead.letter* in your home directory. For instance:

~d
"/usr/lori/dead.letter" 2/24

reads in *dead.letter* to your current message. You can use '~d' to recycle a message you aborted with ^C, since *mail* copies the text of aborted messages into *dead.letter*.

3.8.4. Saving Message Text in a File with '~w'

To save the current text of your message in a file, use the '~w' escape:

~w *itinerary*
"itinerary": 7/180

Mail displays the number of lines and characters written to the file, after which you may continue appending text to your message. You can use Shell metacharacters in the filename, as with '~r'.

3.8.5. Forwarding a Message with '~m' and '~f'

If you are sending mail from within *mail*'s command mode, you can read a message sent to you into the message you are writing with the escape:

```
& mail lori
Subject: Company Party
& ~m 4
Interpolating: 4
(continue)
```

which reads message 4 into the current message, shifted right by one tab stop. You can name any non-deleted message, or list of messages.

You can also forward messages without having them indented by a tab stop in the text with the '~f' escape.

```
& mail lori
Subject: Company Party
& ~f 4
Interpolating: 4
(continue)
```

3.8.6. Adding People to the Message List with '~t'

If, in the process of composing a message, you decide to add additional people to the list of message recipients, you can do so with the escape:

~t *name1 name2 ...*

You may name as few or many additional recipients as you wish. Note that the users originally on the recipient list will still receive the message; you cannot remove someone from the recipient list with '~t'.

3.8.7. Adding a Message Subject with '~s'

If you wish, you can pick a subject for your message or change the one you originally chose if you have the *ask* option set, by using the '~s' escape:

~s New Message Subject

The '~s' replaces any previous subject with 'New Message Subject.' The subject, if given, is placed in the 'Subject' heading near the top of the message. It's a good idea to use the '~p' escape to see what the message will look like if you make a lot of changes which are not immediately displayed as you type in text.

3.8.8. Sending Copies with '~c' and '~b'

If you want to 'copy' people on a message, use the '~c' escape:

~c *name1 name2 ...*

The '~c' escape adds the named people to the 'Cc:' list.

To add blind copy recipients to a message, that is, people who will receive a copy of your message but whose names will not be listed on the message, use the '~b' escape:

~b *name 1 name2 ...*

Again, use '~p' to see what the message will look like.

```
tutorial% mail lori
Subject: Company Party
text
~c joe dan
~b shelley
~p
-------
Message contains:
To: lori
Subject: Company Party
Cc: joe dan
Bcc: shelley

text
(continue)
```

3.8.9. Editing the Header Fields with '~h'

The recipients of the message together constitute the 'To:' field, the subject the 'Subject:' field, and the carbon copies the 'Cc:' field. If you wish to edit these fields in ways impossible with the '~t', '~s', and '~c' escapes, you can use the '~h' escape, which displays each of the fields in turn:

```
~h
To: lori
Subject: Company Party
Cc:
Bcc:
(continue)
```

The '~h' escape first displays 'To:' followed by the current list of recipients and leaves the cursor at the end of the line. Typing in ordinary characters appends them to the end of the current list of recipients. You can also use your erase character DEL to erase back into the list of recipients, or your kill character ^U to erase them altogether. Typing a newline, advances to the 'Subject' field, where the same rules apply. Another newline brings you to the 'Cc:' field, and so on. Another newline leaves you appending text to the end of your message. You can use '~p' to display the current text of the header fields and the body of the message.

3.8.10. Escaping to the Shell with '~!'

To escape to the Shell temporarily to execute a command, use the '~!' escape:

```
~!pwd
/usr/lori/tutorial
!
```

You can use a Shell command and return to mailing mode without altering the text of your message. If you wish, instead, to filter your message through a Shell command, you can use the '~|' escape:

```
~|command
(continue)
```

which pipes your message through the command and uses the output as the new text of your message. If the command does not produce any output, *mail* assumes that something is amiss and retains the old version of your message. A frequently-used filter is the command *fmt*, which justifies the message text. For example, you can type a message like the following and run it through *fmt*:

```
tutorial% mail steve
Subject: Company Party
Please make plans for
the company party in October.
We have a
lot of scheduling and
budgeting to do.
I will be on vacation
for three weeks in
September,
so we should firm up
the plans before I leave.
~|fmt
(continue)
^D
EOT
tutorial%
```

to send your recipient 'steve' the formatted version:

```
From lori Fri Aug  5 16:18:02 1983
Date: 5 Aug 83 16:17:55 PDT (Fri)
From: lori (Lori Rosen)
Subject: Company Party
Message-Id: <8308052317.AA00232@sun.uucp>
Received: by sun.uucp (3.320/3.14)
  id AA00232; 5 Aug 83 16:17:55 PDT (Fri)
To: steve
Status: R

Please make plans for the company party in  October.  We have a lot
of scheduling and budgeting to do.  I will be on vacation for three
weeks in September, so we should firm up the plans before I leave.
```

3.8.11. Escaping to 'mail' Command Mode with '~:'

To escape to *mail* command mode temporarily, use the '~:' escape:

```
~:mail command
```

This is especially useful for reshowing the message you are replying to by using for example:

```
~:t
```

It is also useful for setting options and modifying aliases.

3.8.12. Changing the Tilde Escape and Using a Tilde as a Tilde

If you want or need to change the escape character to something other than the tilde '~', use the 'escape' option. For instance:

```
set escape=]
```

lets you use a right bracket instead of a tilde. Changing the escape character removes the special meaning of the tilde '~'.

If you wish for some reason to send a message that contains a line beginning with a tilde, you must double it. Thus, for example:

~~This line begins with a tilde.

sends the line:

~This line begins with a tilde.

If you ever need to send a line beginning with your new escape character, such as the right bracket as suggested above, double it, just as for '~'.

3.8.13. If You Need Help — '~?'

If you forget which tilde escape does what and need a quick reminder while you are sending someone a message, type the '~?' escape:

~?

The '~?' displays a brief summary of the available tilde escapes.

3.9. Special Recipients

As described previously, you can send mail to either user names or alias names. Special conventions provide the capability of sending messages directly to files or to programs. If a recipient name has a '/' in it or begins with a '+', it is assumed to be the pathname of a file to which to send the message. If the file already exists, the message is appended to the end of the file. If you want to name a file in your current directory (that is, one for which a '/' would not usually be needed) you can precede the name with './'. So, to send mail to the file *memo* in the current directory, type:

tutorial% **mail ./memo**

If the name begins with a '+', it is expanded into the full pathname of the folder name in your folder directory. Sending mail to files can be used for a variety of purposes, such as maintaining a journal and keeping a record of mail sent to a certain group of users. You can keep a record automatically by including the full pathname of the record file in the *alias* command for the group. Using our previous alias example, you can say:

alias project dan rick tom john /usr/project/mail_record

Then, all mail sent to 'project' is saved in the file */usr/project/mail_record* as well as being sent to the members of the project. You can examine this file using **mail -f**.

When you need to send mail directly to a program, preface the program name with a '|'. *Mail* treats recipient names that begin with a '|' as a program to send the mail to. For example, you might write a project billboard program and want to access it using *mail*. To send messages to the billboard program, send mail to the special name '| billboard'. You can set up an alias to refer to a '|' prefaced name if desired.

Caveats: Because the Shell treats '|' specially, you must quote the '| program' on the command line to present it as a single argument to *mail*. Surround the entire name with quotes. This also applies to use with the *alias* command. For example, if you want to alias '/usr/local/bugfiler' to 'bugfiler', say:

alias bugfiler '| /usr/local/bugfiler'

3.10. Additional Features

This section describes some additional commands for handling lists of messages, receiving your mail, and setting options.

3.10.1. Message Lists

Several *mail* commands accept a *message list* as an argument. Along with *type* and *delete*, described above, there is the *from* command, which displays the message headers associated with the message list passed to it. Use *from* in conjunction with some of the message list features described below.

A message list consists of a list of *message numbers*, *ranges*, and *names*, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters '^', '.' or '$' to specify the first relevant, current, or last relevant message, respectively. 'Relevant' means 'not deleted' for most commands, and 'deleted' for the *undelete* command.

A *range* of messages consists of two message numbers separated by a hyphen. So, to display the first four messages, use:

& **type 1-4**

and to display all the messages from the current message to the last message, use:

& **type .-$**

A *name* is a user name. The user names given in the message list are collected, and each message selected by other means is checked to make sure that one of the named users sent it. If the message consists entirely of user names, then every message that is relevant and sent by one of those users is selected. Thus, to display every message that *peter* sent to you, type:

& **type peter**

As a shorthand notation, you can specify simply '*' to get every 'relevant' (same sense) message. So to display all undeleted messages, type:

& **type ***

To delete all undeleted messages, type:

& **delete ***

And to undelete all deleted messages, type:

& **undelete ***

You can search for the presence of a word in subject lines with '/'. For example, to display the headers of all messages that contain the word 'PASCAL', say:

& **from /pascal**

Note that subject searching ignores upper/lower case differences.

3.10.2. List of Commands for Receiving Mail

This section describes all the *mail* commands available when receiving mail.

& !*command*
: Escape to the Shell to process *command*.

& -
: Go to the previous message and display it.

& ?
: Display a brief help summary about *mail* commands. Same as *help*.

alias
: Define a name to stand for a set of other names. Use this when you want to send messages to a certain group of people and want to avoid retyping their names. For example:

    alias gang jon marty steve evan darryl

  creates an alias *gang*, which expands to the five people 'jon', 'marty', 'steve', 'evan',

and 'darryl'. If the given alias already exists, the listed names are added to it.

cd Change current directory. *Cd* takes a single argument, the pathname of the directory to change to. If no argument is given, *cd* changes to your home directory.

copy Copy messages into a file without deleting them when you quit. See *save*.

delete Delete a list of messages. Reclaim deleted messages with the *undelete* command.

dp Delete the current message and display the next message. The *dp* command is useful for quickly reading and disposing of mail.

dt Same as *dp*.

edit Edit individual messages using the text editor. *Edit* takes a list of messages as described under the *type* command and writes each into the file *Messagex* (where 'x' is the message number being edited) for editing. When you have edited the message, write the message and quit. *Mail* reads the message back and removes the file. You may abbreviate *edit* to *e*.

else Mark the end of the *then*-part of an *if* statement and the beginning of the part to take effect if the condition of the *if* statement is false.

endif Mark the end of an *if* statement.

exit Leave *mail* without updating the system mailbox or the file you were reading. Thus, if you accidentally delete several messages, use *exit* to avoid scrambling your mailbox.

file List the names of the folders in your folder directory. Same as *folder*.

folder or **folders**

Switch to a new mail file or folder. With no arguments, *folder* tells you which file you are currently reading. If you give it an argument name, it writes out changes (such as deletions) you have made in the current file and reads the new file. Use these special conventions for the name:

Name	Meaning
#	Previous file read
%	Your system mailbox
%name	*Name*'s system mailbox
&	Your ~/mbox file
+ folder	A file in your folder directory

from Display header lines for each message in a list. To display all the message headers from 'lori' for example, type:

```
from lori
1 lori  Fri Jul 22 10:10:38 10/128 "Subject"
2 lori  Wed Jul 27 10:15:20 11/120
5 lori  Fri Jul 29 10:16:52 13/150
```

headers Reprint the current list of message headers. When you start up *mail* to read your mail, it lists the message headers that you have. These headers tell you who each message is from, when it was sent, how many lines and characters each message has, and the 'Subject:' header field if present. In addition, *mail* tags the message header of each message that has been the object of the *preserve* command with a 'P'. A '*' flags messages that have been saved or written. Finally, deleted messages are not shown at all. *Headers* (and thus the initial header listing) only lists the first so many message headers. The number of headers listed depends on the speed of your system. You can override this by specifying the number of headers you want with the *screen* command. *Mail* maintains a notion of the current 'window' into your messages for the purpose of displaying headers. Use the *z* command to move forward and back a window. You can change the notion of the current window directly to a particular message by using, for example:

```
& headers 40
```

to move *mail*'s attention to the messages around message 40. You can abbreviate the *headers* command to *h*.

help Display a brief help message about the *mail* commands.

hold Hold a list of messages in the system mailbox, instead of moving them to the *mbox* file in your home directory. If you set the binary option *hold*, this will happen by default. Same as *preserve*.

if

Execute commands in your *.mailrc* file conditionally depending on whether you are sending or receiving mail with the *if* command. For example, you can do:

```
if receive
  commands...
endif
```

An **else** form is also available:

```
if send
  commands...
else
  commands...
endif
```

Note that the only allowed conditions are *receive* and *send*.

ignore Add the list of header fields named to the 'ignore list.' Header fields in the ignore list are not shown on your screen when you display a message so you can suppress the display of certain machine-generated header fields, such as *Via* which are not usually of interest. Use the *Type* and *Print* commands to display a message in its entirety, including ignored fields. If *ignore* is executed without arguments, it lists the current set of ignored fields.

list List the valid *mail* commands.

mail Send mail to one or more people. If you have the *ask* option set, you are prompted for a subject to your message. Then you can type in your message, using tilde escapes as described earlier to edit, display, or modify your message. To send your message, type **^D** at the beginning of a line, or a '.' alone on a line if you set the option *dot*. To abort the message, type two interrupt characters (^C) in a row or use the '~q' escape.

mbox Send a list of messages to *mbox* in your home directory when you quit. This is the default action for messages if you do not have the *hold* option set.

next Go to the next message and show it. If given a message list, *next* goes to the first such message and shows it. For example, to go to the next message sent by *steve* and show it, type:

```
& next steve
```

You can abbreviate the *next* command to simply a newline, which means that you can go to and type a message by simply giving its message number or one of the magic characters '↑', '.', or '$'. So, to display the current message, type:

```
& .
```

And to display message 4, say:

```
& 4
```

preserve Keep a list of messages in your system mailbox when you quit. Same as *hold*.

Print Like *print*, but also displays ignored header fields. See also *print ignore*.

print Display each message your on workstation. Abbreviate to *p*. Same as *type*.

quit Leave *mail* and update the file, folder, or system mailbox you were reading. Messages that you have examined are marked as 'read' and messages that existed when you started are marked as 'old.' If you were editing your system mailbox and if you have set the binary option *hold*, all messages which have not been deleted, saved, or mboxed are retained in your system mailbox. If you were editing your system mailbox and you did not have *hold* set, all messages which have not been deleted, saved, or preserved are moved to the *mbox* file in your home directory.

Reply Reply to a one or more messages. The reply (or replies if you are using this on multiple messages) is sent ONLY to the person who sent you the message (respectively, the set of people who sent the messages you are replying to). You can add people using the '~t' and '~c' tilde escapes. The subject in your reply formed by prefacing the subject in the original message with 'Re:' unless it already begins so. If the original message included a 'reply-to' header field, the reply goes only to the recipient named by 'reply-to.' You type in your message using the same conventions available through the *mail* command. The *Reply* command is especially useful for replying to messages that were sent to enormous distribution groups when you really just want to send a message to the originator.

reply Reply to a single message. The reply is sent to the person who sent you the message to which you are replying, plus *all* the people who received the original message, except you. You can add people using the ~*t* and ~*c* tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with 'Re:' unless it already begins so. If the original message included a 'reply-to' header field, the reply only goes to the recipient named by 'reply-to.' Type in your message using the same conventions available through the *mail* command.

save Save messages on related topics in a file. *Save* takes as an argument a list of message numbers, followed by the name of the file in which to save the messages. The messages are appended to the named file, so you can keep several messages in the file, stored in the order they were put there. You can abbreviate the *save* command to *s*. You can save messages 1 and 2 in *good.mail* for example, by typing:

```
& s 1 2 good.mail
```

Saved messages are deleted and not automatically saved in *mbox* at quit time. They are not selected by the *next* command described above unless explicitly specified.

set Customize *mail* with options or with valued options. See the *Setting Custom Binary and Valued Options* information that follows. Options can be *binary*, in which case they are *on* or *off*, or *valued*. To set a binary option *on*, do:

set *option*

where *option* is the option name. To give a valued option a value, say:

set *option*=*value*

Several options can be specified in a single *set* command.

Shell Escape to the Shell to type commands to it. When you leave the Shell, you return to *mail*. *Mail* assumes the default Shell, but you can override this default by setting the valued option *SHELL*:

set SHELL=/bin/sh

source Read *mail* commands from a file. It is useful when you are changing your *.mailrc* file and you need to read in the changes.

top Display the first five lines of each addressed message in a message list. It may be abbreviated to *t*. If you wish, you can change the number of lines that *top* displays by setting the valued option *toplines*. On a CRT terminal, you might prefer:

set toplines=10

Type Displays each message with header fields. Identical to the *Print* command.

type Display a list of messages on your screen. If you have set the *crt* option to a number, and the total number of lines in the messages you are displaying exceeds that specified by *crt*, the messages are displayed by a paging program such as *more*. (Same as *print*).

undelete

Restore a deleted message. Only messages that have been deleted may be undeleted. This command may be abbreviated to *u*.

unset Reverse the action of setting a binary or valued option.

visual Invoke a display-oriented editor. The operation of the *visual* command is otherwise identical to that of the *edit* command. Both the *edit* and *visual* commands assume some default text editors. You can override these default editors with the valued options *EDITOR* and *VISUAL* for the standard and screen editors. The defaults are:

EDITOR=/usr/ucb/ex VISUAL=/usr/ucb/vi

write Write the message into a file. Just like *save*, except that *write* deletes the first (normally 'From:' line) and last (normally blank) lines. *Write* has the same syntax as *save* and can be abbreviated to simply *w*. Thus, you can write the second message by doing:

& **w 2 file.save**

z Move the message header window forward. Type:

& **z+**

Analogously, you can move to the previous window with:

& **z–**

3.10.3. Setting Custom Binary and Valued Options

This section describes each of the options in alphabetical order, including some that you have not yet seen. To avoid confusion, please note that the options are either all lowercase letters or all uppercase letters. We begin sentences with capitalized option names as a courtesy to English. Unless otherwise stated, the default value of all binary options is false (unset).

EDITOR Define the pathname of the text editor to be used in the *edit* command and '~e'. If not defined, *ex* is used. A valued option.

SHELL Give the pathname of your Shell. This Shell is used for the '!' command and '~!' escape. In addition, it expands filenames with Shell metacharacters like '*' and '?' in them. Default is *csh*. A valued option.

VISUAL Define the pathname of your screen editor for use in the *visual* command and '~v' escape. Invokes *vi* unless otherwise defined. A valued option.

append Appends messages to the end of your *mbox* file rather than to the beginning. Normally, messages are put in *mbox* in the same order that the system puts messages in your system mailbox. A binary option. This option is set in the system file */usr/lib/Mail.rc*. So by default, messages are appended to the end of the system

mailbox and, your *mbox* file. You may override it in your *.mailrc* or your system administrator may remove it, if desired.

ask Prompt you for the subject of each message you send. If you respond with simply a newline, no subject field is sent. A binary option.

askcc Prompt you for additional 'carbon copy' recipients at the end of each message. Responding with a newline shows your satisfaction with the current list. A binary option.

autoprint

Cause the *delete* command to behave like *dt*. After deleting a message, the next one is displayed automatically. This is useful for quickly scanning and deleting messages in your mailbox. A binary option.

debug Display debugging information. Same as using the **–d** command line flag. A binary option.

dot Cause *mail* to interpret a period alone on a line as the terminator of a message you are sending. A binary option. This is set in the default system file */usr/lib/Mail.rc*. It may be removed or overridden in your *.mailrc*.

escape Allow you to change the escape character used when sending mail. Only the first character of the *escape* option is used, and it must be doubled if it is to appear as the first character of a line of your message. If you change your escape character, then '~' loses all its special meaning, and need no longer be doubled at the beginning of a line. A valued option.

folder Determine the name of the directory to use for storing folders of messages. If this name begins with a '/' *mail* considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

hold Hold messages that have been read but not manually dealt with in the system mailbox to prevent them from being automatically swept into your *mbox*. A binary option.

ignore Ignore ^C's (or RUBOUT) characters from your system and echo them as '@'s' while you are sending mail. The ^C characters retain their original meaning in *mail* command mode. Setting the *ignore* option is equivalent to supplying the **–i** flag on the command line. A binary option.

ignoreeof

Make *mail* refuse to accept your current EOF character (^D by default) as the end of a message. *Ignoreeof* also applies to *mail* command mode, and is related to *dot*.

keep Truncate *mail* system mailbox instead of deleting it when it is empty. This is useful if you elect to protect your mailbox, which you do with the Shell command:

tutorial% **chmod 600 /usr/spool/mail/***yourname*

where *yourname* is your login name. If you do not do this, anyone can probably read your mail.

keepsave

Retain all saved messages. When you save a message in a file, *mail* usually discards it when you quit.

metoo

Include yourself as a recipient when sending mail to an alias. Otherwise, *mail* does not send you a copy if you are included in the alias. A binary option.

nosave Prevent *mail* from copying a partial letter to the *dead.letter* file in your home directory when you abort a message with two ^C's (RUBOUT's). A binary option.

quiet

Suppress the display of the *mail* version when *mail* is first invoked and the message number from the *type* command. A binary option.

record Name a record file to save your outgoing mail. Each new message you send is appended to the end of the file. A valued option.

screen Override any terminal speed consideration that may affect how *mail* prints the message headers. Usually, the faster your terminal, the more it displays. The value of *screen* specifies how many message headers you want displayed. This number is also used for scrolling with the *z* command. A valued option.

sendmail

Use an alternate delivery system. Set the *sendmail* option to the full pathname of the program to use. Note: this is not for everyone! Most people should use the default delivery system. A valued option, set to the full pathname of the program to use.

toplines

Define the number of lines that the *top* command displays instead of the default five lines. A valued option.

verbose Invoke *sendmail* with the **–v** flag to use verbose mode and announce expansion of aliases, etc. Equivalent to invoking *mail* with the **–v** flag. A binary option.

3.10.4. Command Line Options

This section describes the use of *mail* command line options.

–N Suppress the initial printing of headers. For example, to get into *mail*, type:

```
tutorial% mail -N
&
```

–d Turn on debugging information. Not of general interest.

–f *file* Show the messages in *file* instead of your system mailbox. If *file* is omitted, *mail* reads *mbox* in your home directory.

–i

Ignore tty interrupt signals while typing in a mail message. You can still use '~q' to interrupt the message. Useful on noisy phone lines, which generate spurious RUBOUT or DELETE characters. This is usually unnecessary if your INTERRUPT character is the default ^C or if you're not logged in over a phone line. (See the *stty* Shell command in the user's manual for details.)

–n Inhibit reading of */usr/lib/Mail.rc*.

–s *string*

Denote the subject of a message when sending mail. If *string* contains blanks, surround it with quote marks.

–u *name* Read *name*'s mail instead of your own. Unwitting others often neglect to protect their mailboxes, but discretion is advised.

–T *file* Arrange to print on *file* the contents of the *article-id* fields of all messages that were either read or deleted. *–T* is for the *readnews* program and should NOT be used for reading your mail.

–v Use the **–v** flag when invoking *sendmail*. This feature may also be enabled by setting the option *verbose*. Useful for diagnosing mail delivery problems.

3.11. Message Format

A sample message format is:

```
From lori Wed Jul 27 10:16:52 1983
Date: 27 Jul 83 10:16:45 PDT (Wed)
From: lori (Lori Rosen)
Subject: Company Cruise
Message-Id: <8307271716.AA05188@sun.uucp>
Received: by sun.uucp (3.320/3.14)
  id AA05188; 27 Jul 83 10:16:45 PDT (Wed)
To: alison
Status: R
```

Messages begin with a *from* line, which consists of the word 'From' followed by a user name, followed by anything (usually null), followed by a date in the format returned by the *ctime* library routine described in the user's manual. The *ctime* date may be optionally followed by a single space and a time zone indication, which should be three capital letters, such as PDT.

Following the *from* line are zero or more *header field* lines. Each header field line is of the form:

```
name: information
```

Name can be anything, but only certain header fields are recognized as having any meaning. The recognized header fields are: *article-id, bcc, cc, from, reply-to, sender, subject,* and *to.* Other header fields are also significant to other systems; see, for example, the current ARPANET message standard for more on this topic. A header field can be continued onto following lines by making the first character on the following line a space or tab character.

If any headers are present, they must be followed by a blank line. The part that follows is called the *body* of the message, and must be ASCII text, not containing null characters. Each line in the message body must be terminated with an ASCII newline character and no line may be longer than 512 characters. If binary data must be passed through the mail system, it is suggested that this data be encoded in a system which encodes six bits into a printable character. For example, you could use the upper and lower case letters, the digits, and the characters comma and period to make up the 64 characters. Then, you can send a 16-bit binary number as three characters. Pack these characters into lines, preferably lines about 70 characters long as long lines are transmitted more efficiently. The file */usr/bin/unencode* and */usr/bin/undecode* are useful for encoding and decoding binary data into the recommended form.

The message delivery system always adds a blank line to the end of each message. This blank line must not be deleted.

The UUCP message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

Note that some network transport protocols enforce limits on the lengths of messages.

3.12. Summary of Commands, Options, and Escapes

This section gives a quick summary of the *mail* commands, binary and valued *set* options, and tilde escapes.

3.12.1. 'mail' Command Summary

The following table describes the commands.

mail Commands

Command	Description
!	Single command escape to Shell
-	Back up to previous message
Print	Show message with ignored fields
Reply	Reply to author of message only
Type	Show message with ignored fields
alias	Define an alias as a set of user names
alternates	List other names you are known by
cd	Change working directory, home by default
copy	Copy a message to a file or folder
delete	Delete a list of messages
dt	Delete current message, type next message
endif	End of conditional statement; see *if*
edit	Edit a list of messages
else	Start of else part of conditional; see *if*
exit	Leave mail without changing anything
file	Interrogate/change current mail file
folder	Same as *file*
folders	List the folders in your folder directory
from	List headers of a list of messages
headers	List current window of messages
help	Print brief summary of *mail* commands
hold	Same as *preserve*
if	Conditional execution of *mail* commands
ignore	Set/examine list of ignored header fields
list	List valid *mail* commands
mail	Send mail to specified names
mbox	Arrange to save a list of messages in *mbox*
next	Go to next message and show it
preserve	Arrange to leave list of messages in system mailbox
print	Show messages
quit	Leave *mail*; update system mailbox, *mbox* as appropriate
reply	Reply to a message
save	Append messages, headers included, on a file
set	Set binary or valued options
shell	Invoke an interactive Shell
top	Show first so many (5 by default) lines of list of messages
type	Show messages
undelete	Undelete list of messages
unset	Undo the operation of a *set* command
visual	Invoke visual editor on a list of messages
write	Append messages to a file, not including headers
z	Scroll to next/previous screenful of headers

3.12.2. 'set' Command Option Summary

The following table describes the options. Each option is shown as being either a binary or valued option.

Binary and Valued Options

Option	*Type*	Description
EDITOR	*valued*	Pathname of editor for '~e' and **edit**
SHELL	*valued*	Pathname of Shell for *shell*, '~!' and '!'
VISUAL	*valued*	Pathname of screen editor for '~v', *visual*
append	*binary*	Always append messages to end of *mbox*
ask	*binary*	Prompt user for 'Subject:' field when sending
askcc	*binary*	Prompt user for additional 'Cc's' at end of message
autoprint	*binary*	Print next message after *delete*
crt	*valued*	Set minimum number of lines before using *more*
debug	*binary*	Display debugging information
dot	*binary*	Accept '.' alone on line to terminate message input
escape	*valued*	Escape character to be used instead of '~'
folder	*valued*	Set directory to store folders in
hold	*binary*	Hold messages in system mailbox by default
ignore	*binary*	Ignore ^C's (RUBOUT) while sending mail
ignoreeof	*binary*	Don't terminate letters/command input with **D**
keep	*binary*	Don't unlink system mailbox when empty
keepsave	*binary*	Don't delete **s**aved messages by default
metoo	*binary*	Include sending user in aliases
nosave	*binary*	Don't save partial letter in *dead.letter*
quiet	*binary*	Suppress printing of *mail* version
record	*valued*	File to save all outgoing mail in
screen	*valued*	Size of window of message headers for **z**, etc.
sendmail	*valued*	Choose alternate mail delivery system
toplines	*valued*	Number of lines to print in **top**
verbose	*binary*	Invoke *sendmail* with the **–v** flag

3.12.3. Tilde Escape Summary

The following table summarizes the tilde escapes available while sending mail.

Tilde Escapes

Escape	*Arguments*	Description
~!	*command*	Execute Shell command
~c	*name ...*	Add names to 'Cc:' field
~d		Read *dead.letter* into message
~e		Invoke text editor on partial message
~f	*messages*	Read named messages
~h		Edit the header fields
~m	*messages*	Read named messages, right shift by tab
~p		Display message entered so far
~q		Abort entry of letter; like INTERRUPT (^C or RUBOUT)
~r	*filename*	Read file into message
~s	*string*	Set 'Subject:' field to *string*
~t	*name ...*	Add names to 'To:' field
~v		Invoke screen editor *vi* on message
~w	*filename*	Write message on file
~|	*command*	Pipe message through *command*
~~	*string*	Quote a '~' in front of *string*

3.12.4. 'mail' Command Line Flags

The following table shows the command line flags that *mail* accepts.

mail Command Line Flags

Flag	Description
-N	Suppress the initial printing of headers
-T *file*	Article-id's of read/deleted messages to *file*
-d	Turn on debugging
-f *file*	Show messages in *file* or ~/*mbox*
-i	Ignore tty interrupt signals
-n	Inhibit reading of /usr/lib/Mail.rc
-s *string*	Use *string* as subject in outgoing mail
-u *name*	Read *name's* mail instead of your own
-v	Invoke *sendmail* with the -v (verbose) flag.

Notes: Do not use **-T** and **-d** for normal operation.

4. NETWORK NEWS USER'S GUIDE

The network news, or simply *netnews*, is the set of programs that provide access to the User's Network called USENET. With *netnews*, you can post news articles for limited or very wide distribution on the USENET. You can post an article, which will be sent out to the network to be read by others interested in that topic. There are facilities for browsing through old news, posting follow-up articles, and sending direct electronic mail replies to the author of an article.

Whenever you read the news, you are presented with interesting articles that you have not yet read. These are divided into topics or *newsgroups*. You can specify which topics you are interested in with a *subscription list*. At the end of this guide, there is a list of newsgroups to help you determine which newsgroups you may want to subscribe to. *Netnews* keeps old articles around until they expire, which is usually about two weeks, so you can browse through old news from time to time.

USENET is a bulletin board shared among many computer systems in the computer science community, around the United States, Canada, Europe, and Australia. USENET is a logical network, sitting on top of several physical networks, including uucp, BLICN, and Berknet. Sites on USENET include many universities, private companies and research organizations.

USENET is useful in a number of ways:

- to share useful information,
- to report bugs and fixes without mass mailings, and
- to have discussions involving many people at different locations without having to get everyone together.

For additional options and details on the network news, refer to the user's manual on *readnews*, *checknews*, *postnews*, and *inews*.

4.1. Making the Connection with Your News Host System

To use *netnews*, a host system at your site (the news host) must connnect regularly by uucp to another site from which news can be forwarded. This must be arranged directly between your system administrator and the forwarding site. See the *System Manager's Manual for the Sun Workstation* for more information. If your system itself is the news host, you can run *readnews* directly. If the news host is another machine on your local Ethernet, you must do a remote login (*rlogin*). Let's assume your news host system is called 'mercury'. Log in to 'mercury' by typing:

```
tutorial% rlogin mercury
Last login: Tue Aug 5 13:41:36 on ttyp0
Sun UNIX 4.2 (Berkeley beta release) (GENERIC) #8: Oct 23 13:45:52 PDT 1983
mercury%
```

You are now ready to read the news.

For additional information on how to connect your system to the USENET, refer to the *System Manager's Manual for the Sun Workstation*.

4.2. How to Read the News with 'readnews'

The following assumes that your system 'tutorial%' is directly hooked up to USENET via uucp.

Use the *readnews* command to read the news:

```
tutorial% readnews
```

For each newsgroup to which you subscribe is displayed, one article at a time will be presented. You will be shown the article *header*, containing the name of the author, the subject, and the

---

The material in this guide is derived from *How to Read the Network News*, Mark R. Horton, Bell Telephone Laboratories.

length of the article. You are asked if you want to read more. The three most common responses are:

- Type **y** for 'yes' (or simply press RETURN) to display the rest of the message. (If the message is long, it may stop before it runs off the top of the screen. Type a space or press RETURN to see more of the message.)
- Type '**n**' for 'no' to indicate you are not interested in the message; it will not be offered to you again.
- Type '**q**' for 'quit' to make a record of which articles you read (or refused) and to exit *netnews*. When you have read all the news, this happens automatically. The quit command is useful if you are in a hurry and don't have time to read all the news right now.

To see a complete list of possible responses, type '**?**' for help.

4.3. Reading News for the First Time

If you are reading news for the first time, you may find yourself swamped by the volume of unread news, especially if the subscription list default is 'all'. Decide which *newsgroups* you want to read about. If you are getting newsgroups in which you have no interest, you can change your subscription list as we show below. Also, bear in mind that what you see is probably at least two weeks accumulation of news. If you want to just get rid of all old news and start anew, use the *readnews* command with the **–p** and **–n** options, which throw away all articles and any diagnostic output into */dev/null*. Note that this take a long time, so it's best to run the command in the background by typing an '&' at the end of the command line:

```
tutorial% readnews –p –n all >& /dev/null &
[1] 2345
tutorial%
```

This throws away all old news, recording that you have seen it.

Once you catch up or ignore all the old news, the news comes in daily at a more manageable rate. If the daily rate is still too much you may wish to change your subscription list to exclude some of the high volume newsgroups. Finally, note that while you are displaying an article, you can type a ^C (an INTERRUPT or DELETE) to throw away the rest of the article.

Other commands you can type after seeing the article header are:

u Unsubscribe from this newsgroup. Goes on to the next group. You will never see this newsgroup again unless you edit your *.newsrc* file (see below).

x Exit *readnews*. Unlike *quit*, does not update the record of which articles you have read and pretends you never started *readnews*.

N Go on to the next newsgroup. The remaining articles in the current newsgroup are considered 'unread' and are offered to you again the next time you read news.

s *filename*

Save the article in a disk file with the given *filename*. What usually happens is that an article is displayed, and then *readnews* goes on to display the header of the next article before you get a chance to type anything. To write out the *previous* message, that is, the last one you have read in full, use the form **s-** *filename*.

r Reply to the author of the message. Type in your message, and end it with ^D. This sends *mail* addressed to the author. You then get back to *readnews*. Use **r–** to reply to the previous message. Beware that this may sometimes generate an incorrect address. Check the address and correct it withh ~**h** if necessary.

f Post a follow-up message to the same newsgroup. Type in the body of your reply and type ^D. This posts an article on this newsgroup with the same title as the original article. Be sure your article is worthy of posting; many follow-up articles should be replies. Use '–' to follow up the previous message. If you type this by accident, type ^C (an INTERRUPT) to abort the follow-up. Note that the people who administer the machines on USENET are concerned about people sending news to inappropriate newsgroups. We suggest you read several week's worth of news and read up on network etiquette before posting any news.

+ Skip the article for now. The next time you read news, you are offered this article again.

- Go back to the previous article.

? Display a summary of valid commands. The '?' is also displayed if you type any unrecognized command.

4.4. Changing Your Subscription List

If you don't take any special action, you will subscribe to a default subscription list. This default varies locally. To find out your local default, type:

```
tutorial% readnews -s
Subscription list: general,all.general,general
```

Typically this list includes all newsgroups ending in 'general', such as 'general', and 'net.general'.

To change this, create a file in your home directory named *.newsrc* and type as its first line:

options -n *newsgroup newsgroup newsgroup ...*

Continue long lines on subsequent lines by beginning with a space. The *netnews* system updates this file to record which articles you have read. Ignore these lines unless you want to edit them. For example, if you are creating a subscription list for the first time, and have already read news, you will find some update text in your *.newsrc* file, recording which articles you have read. Put your *options* line before the first line of the file. For instance, a *.newsrc* file can look like:

```
options -n general net.ai fa.telecom
net.chess: 1-224
net.games.rogue: 1-45
net.games.trivia: 1-234
```

The update text shows three articles and the *options* lines that subscribes to the three newsgroups, 'general,' 'net.ai', and 'fa.telecom.'

Type:

!newsgroup

to exclude certain newsgroups and add the word '**all**' as a wild card to represent any newsgroup. You can also use '**all**' as a prefix or suffix to match a class of newsgroups. For example:

```
options -n all !fa.all !net.jokes !all.unix-all
```

subscribes to all newsgroups except for ARPANET news, jokes, and any UNIX information. The metacharacter '.' is like '/' to the Shell, and 'all' is like '*'.

4.5. Submitting Articles with 'postnews' and 'inews'

To submit a news article, use the *postnews* command, which then prompts you for an article title, newsgroup(s), and your news.

```
tutorial% postnews
Title:
Newsgroups (general):
Type news, end with control D
body of article
^D
EOT
tutorial%
```

Typing RETURN after the 'newsgroup' prompt uses the default newsgroup, 'general.' You can also use:

tutorial% **postnews** *filename*

in which case *postnews* uses the specified *filename* as the article.

If the environment variable EDITOR is set to the pathname of an editor, you can use that editor to type in your article. When creating a new article, specifying the newsgroup controls the level of distribution.

For something fancier than a short note, use the *inews* command, which has several options, such as specifying an article expiration date and the name of the sender.

```
tutorial% inews -t title -n newsgroup [-e expiration date] [-f sender]
body of article
^D
EOT
tutorial%
```

Inews does not prompt you like *postnews* does.

You can also use an editor with *inews* to prepare the body of your article. Edit your file, then type:

tutorial% **inews** **-t** *title* [**-n** *newsgroup*] **<** *file*

to use sent the edited file as your news article. A sample command line looks like:

```
tutorial% inews -t seminar -n net.micro -e next friday < seminfo
```

This article announces a seminar to the 'net.micro' newsgroup readers. This expiration date option is useful if you are announcing a meeting for a particular date after which the announcement will no longer be valid.

On some systems, it is possible to post news articles by sending mail. See the section on *mail* in *User Interface* in this guide or refer to *mail* in the user's manual.

The network news provides a unique method of communication for many people. Please consider the following guidelines when using USENET and the network news:

- Put all items in an appropriate group.
- Use *mail* instead of a followup news item.
- Be careful when preparing articles for submission.
- Read followups before responding.
- Use an editor to prepare items for submission.
- Don't be rude or abusive.
- Avoid sarcasm and facetious remarks.
- Use descriptive titles.
- Whenever possible, cite references.
- Make a summary of the original item in followups.
- In posting summaries of replies, do make a summary.
- Don't force people to read the same thing more than once.
- Be as brief as possible.

4.6. Browsing Through Old News

Readnews command line options that help you find an old article again are:

-n *newsgroups*
: Restricts your search to certain newsgroups.

-x
: Ignores the record of articles read kept in your *.newsrc* file. This displays all articles in all newsgroups to which you subscribe, even those which you have already seen. It also prevents *readnews* from updating the *.newsrc* file.

-a *date*
: Asks for news received since the given *date*. Note that even with the **-a** option, only articles you have not already seen are displayed, unless you combine it with the **-x** option. Articles are kept on file until they expire, typically after two weeks.

-t *keywords*
: Restricts the query to articles mentioning one of the *keywords* in the title of the article. If you try:

tutorial% **readnews -n** net.unix-wizards **-x -a** last friday**-t** setuid

you see all articles in newsgroup net.unix-wizards since last Friday about the setuid feature. The -t option finds articles about the specific keyword; it does not find articles about 'suid', 'Setuid', nor articles with no title or whose author did not use the word 'setuid' in the title.

-l
: Lists only the headers of articles — a useful form for browsing through lots of messages.

-p
: Prints the messages without asking for any input.

-r
: Produces articles in reverse order, from newest to oldest.

4.7. Getting News When You Log In — Your Morning Newspaper

To be told if there is any news when you first log in, put *checknews* or *readnews* in your *.login* or *.profile* file. This way you are reminded of news, but are not interrupted by it during the day.

The *readnews* command tries to find all unread news (assuming you are going to read it), and takes a lot of time to do it, so it's better to use the smaller, faster *checknews*, which tells you if there is any news. It was designed especially for a login file. There are also options to be silent if there is (or is not) news, and to start up *readnews* automatically if there is news.

The *checknews* options are:

-y Print 'There is news' if there is any news arrives during a login session.

-v If **-y** is also given, instead of printing 'There is news', print 'News: *newsgroup* ...' giving the name of the first newsgroup containing unread news.

-n Print 'No news' if there is no unread news.

-e Start up *readnews* if there is any unread news. Any additional arguments after the **-e** are passed to *readnews*.

Thus, *checknews -yn* tells you whether there is any unread news. *Checknews -y* tells you if there is news, and is silent if there is no news.

4.8. Creating New Newsgroups

To create a newsgroup, post an article to an appropriate 'general' newsgroup suggesting the new newsgroup, (for example, for a new 'net' or 'fa' newsgroup, post to 'net.general', for a new local newsgroup, post to 'general') with another copy to 'net.news.group', for example:

tutorial% **inews -t** suggested new newsgroup **-n net.general,net.news.group**

Other users will follow-up to 'net.news.group', giving opinions about whether the suggested newsgroup makes sense, should have a different name, etc.

When agreement is reached and it is established that there is interest in the topic, ask your local netnews administrator to create the newsgroup. It can actually be created by any netnews administrator anywhere on the net, within the scope of the newsgroup. Once the newsgroup is created and the first article has been posted, the newsgroup is available for all interested persons to post to.

4.9. User Interfaces

The *user interface* of a program is the face it presents to the user, that is, what it displays and what it allows you to type. *Readnews* has options allowing you to use different user interfaces. These are:

- The **-c** option displays the the entire message, header and body, and prompts you at the end of the message. The command options are the same as the msgs interface, but it is usually not necessary to use the '-' suffix on the reply, save, or followup commands. This interface is called the '/bin/mail' interface, because it mimics the UNIX program of that name.
- The *mail* interface, available with the **-M** option, invokes the *mail* program directly, and allows you to read news with the same commands as you read mail. This interface may not work on your system - it requires a special version of mail with a **-T** option.)
- Use your favorite mail system as an interface, including /bin/mail and mail. Any mail system with an option to specify an alternative mailbox can be used to read news. For example, to use mail without the **-M** option, type:

tutorial% **readnews -c "mail -f %"**

The Shell command in quotes is invoked as a child of *readnews*. The **-f** option to mail names the alternative mailbox. Readnews puts the news in a temporary file, and gives the name of this file to the mailer in place of the '%'. There is an important difference when using this kind of interface. The mailers do not give any indication of which articles you read and which ones you skipped. Readnews assumes you read *all* the articles, even if you didn't, and marks them all read. By contrast, the **-M** option uses the **-T** option to mail, asking mail to tell readnews which articles you read.

4.10. From the ARPANET

The 'FA' (from the ARPANET) newsgroups have a different convention for posting news. Rather than using *inews*, *postnews*, or the *followup* command, send mail to a particular electronic mailing address along the return address found in the From line of the article. Although it is possible to post news directly to the newsgroup, don't do it.

The correct way to post news to an 'FA' newsgroup is to send electronic mail to **ucbvax!C70:***newsgroup*. (You will have to route your mail to ucbvax - inquire locally how to do this or check the return address on any 'FA' article.) For this reason, all articles in 'FA' newsgroups have a return address of the form **...!ucbvax!C70:***newsgroup*.

'Fa' newsgroups are electronic mailing lists on the ARPANET. A number of people on the ARPANET get the mailings directly from the mailing lists. One entry on each mailing list is of the form **post-***newsgroup***@Berkeley** which is fed into a program that posts the article on newsgroup **fa.***newsgroup*. From there it is distributed to the other sites on USENET. If you post an article directly to the newsgroup, you will reach all the readers of that newsgroup on USENET, but you will miss all the people getting the direct mailing on the ARPANET.

To follow up an 'FA' article, use the *reply* command of *readnews*, not the *follow-up* command. This insures that ARPANET members also see the reply.

If you are a USENET site on the ARPANET, the corresponding mailing address is *newsgroup***@Berkeley**, although it also works to send mail directly to the ARPANET list, if you know the proper address. The address at Berkeley only forwards to the real ARPANET mailing list. Thus, for example, sending mail from the ARPANET to either **CSVAX.unix-wizards@Berkeley** or **unix-wizards@SRI-WARF** is correct, but sending mail to **CSVAX.post-unix-wizards@Berkeley** is wrong, since only USENET readers will see it.

4.11. List of Newsgroups

This lists the active newsgroups to help you decide which you want to subscribe to. Note that the list is constantly changing. Note also that this list is specially tailored for the Berkeley sites. Check with your netnews administrator for a local list.

There are two basic subcategories of netwide newsgroups:

1. The 'net.all' group consists of USENET bulletin board newsgroups that are circulated around the entire net.
2. The 'fa.all' group is a set of groups that are gatewayed to USENET from the ARPANET. These groups consist mainly of digests, although there are some bulletin boards.

Some of the 'net.all' and 'fa.all' groups are gatewayed between the networks, that is, items submitted from the ARPA side to the digest are split up and submitted to the USENET group, while articles submitted on the USENET side are bundled up and submitted to the digest.

4.11.1. Local Newsgroups

Local groups are kept on the current machine only. Local names can be identified by the lack of a prefix, that is, there are no periods in local newsgroup names.

general — News and important announcements to be read by everyone on the local machine. This newsgroup is usually mandatory. The list of mandatory newsgroups varies locally. *csmsgs*.)

4.11.2. FA Newsgroups

FA groups are 'from the ARPANET' and are mostly copies of mailing lists or 'digests' distributed on that network. A digest is a collection of mail, much like a newsletter, that is put together by an editor and sent out every so often. A special convention applies to submissions to FA newsgroups. As previously described, you should not post directly to the newsgroup, since this will be seen by people on USENET but not by the people on the ARPANET who get the list directly mailed to them. Instead, send mail to the return address on any article, by using the reply

command to *readnews*. For example, to post to **fa.human-nets**, the reply command might mail to **ucbvax!C70:human-nets**

FA groups and their corresponding mailing lists can reach a very large user community, including USENET sites on UUCP, Berknet, BLN, and the ARPANET, as well as sites on the ARPANET which are not on USENET, who get the news via direct electronic mailing.

fa.arms-d	People worried about nukes.
fa.arpa-bboard	Announcements that are posted to all arpanet boards are also fed into this newsgroup.
fa.digest-p	People who deal with digests. Mostly the people who moderate them.
fa.editor-p	Interest group in computer editors, both text and program.
fa.energy	Topics relating to alternate energy production, conservation, etc.
fa.human-nets	A daily moderated digest with discussions of computer-aided human-to-human communications. Probably the most widely read ARPANET publication.
fa.info-cpm	CP/M and other operating systems for micro computers.
fa.info-micro	Microprocessor discussions.
fa.info-terms	Opinions/queries about what's a good/bad computer terminal.
fa.info-vax	VAX interest group. Seems to be mostly VMS issues, but some hardware discussions too.
fa.poli-sci	Political Science discussions digest.
fa.sf-lovers	Science Fiction book/movie reviews, etc.
fa.space	Digest containing comments on the space program and outer space in general.
fa.tcp-ip	Digest relating to the TCP and IP network protocols.
fa.telecom	Technical topics relating to telecommunications, especially the telephone system. A digest recently spun off from fa.human-nets.
fa.teletext	Teletext discusses all aspects of "esoteric" data systems. This includes teletext, viewdata, closed-captioning, and digicasting.
fa.unix-cpm	CPM/UNIX discussions.
fa.works	Interest group on personal workstations (e.g. Apollo, Perq, Sun, Xerox Star, etc).

4.11.3. Net Newsgroups

Net groups are intended to be available to all people on the entire network who read netnews. This does not mean they go to every machine, since some machines restrict the type of news that comes in. Net groups reach all of USENET (including USENET sites on the ARPANET) but do not reach any sites that are not on USENET. That is, USENET is defined as all sites that receive 'net.general'.

net.general	Articles to be read by everyone on the whole net.
net.applic	Functional programming (applicative) languages.
net.auto	Notes of interest to owners of particular cars. Main subgroup is
net.auto.vw	for owners of Volkswagen Rabbits.
net.aviation	Private pilots.
net.bugs	Bug reports and fixes. Subscribing to 'net.bugs' gets all bug reports, but bugs are normally posted to one of 'net.bugs.2bsd', 'net.bugs.4bsd', 'net.bugs.v7', or 'net.bugs.u3', for the 2nd and 4th Berkeley Software Distribution, Version 7, or UNIX system III, as appropriate.

net.chess	Interest group for computer chess. This newsgroup is connected into an ARPANET mailing list but appears as a normal newsgroup to USENET, so it is called 'net.chess' instead of 'fa.chess'.
net.columbia	Newswire items and comments on the Space Shuttle, and on the space program in general.
net.cooks	Food, cooking, cookbooks, and recipes.
net.cycle	Motorcycle interest group.
net.eunice	Topics of interest to sites running SRI's Eunice system, which simulates UNIX on VMS.
net.games	Discussion of computer games (of the /usr/games/variety). Subgroups include 'net.games.rogue', 'net.games.frp' (for fantasy role playing games,) and 'net.games.trivia'.
net.ham-radio	Topics of interest to amateur radio operators.
net.jokes	The latest good joke you've heard.
net.lan	Local area network interest group.
net.lsi	Large Scale Integrated Circuit discussions.
net.misc	Miscellaneous discussions that start in net.general but are not permanent enough for their own newsgroup.
net.movies	Movie reviews by members of USENET.
net.music	Computer generated music.
net.news	Discussion of *netnews* itself. Subgroups discuss or post various aspects of netnews, including 'net.news.b' for the B version of *netnews*, 'net.news.directory' to post all or part of the USENET directory, 'net.news.group' for discussions about proposed new newsgroups, 'net.news.map' to post maps of USENET or additions/corrections to previously posted maps, 'net.news.newsite' to announce a new site. 'net.news' itself is used for discussions relating to USENET policies and the like, rather than any specific software.
net.oa	Office Automation/Word Processing interest group.
net.periphs	Queries and discussions about particular peripherals. ("Does anyone have a driver for a frammis-11?")
net.rec	Recreational games. This differs from 'net.sport' in that 'net.rec' discusses games where one generally participates, but 'net.sport' is for spectator sports. 'net.games' is for computer type games. Subgroups of 'net.rec' include 'net.rec.bridge' for contract bridge discussions, 'net.rec.scuba' for scuba divers, and 'net.rec.ski' for skiers.
net.records	Discussions of phonograph records, albums, record stores, etc.
net.rumor	For posting of rumors.
net.sf-lovers	For science fiction lovers.
net.sources	For posting source code for software distribution.
net.space	Undigested, immediate distribution version of fa.space.
net.sport	Spectator sports. Subgroups include 'net.sport.baseball', 'net.sport.football', and 'net.sport.hockey'.
net.taxes	Tax advice and queries.
net.test	Test messages are posted here. Generally this is not interesting to ordinary readers.

net.travel	Requests, suggestions, and opinions about traveling.
net.ucds	Circuit drawing system.
net.unix-wizards	ARPANET mailing list for UNIX Wizards. Anything and everything relating to UNIX is discussed here. This list is connected to the ARPANET mailing list but appears like a regular 'net' newsgroup to USENET.
net.wines	Information and recommendations about wines and alcoholic beverages.

APPENDIX A: GLOSSARY

This glossary lists the most important terms in this introduction to the Sun system.

. The name of your current directory displayed by the command *pwd*; also see *dirs*. Usually the first *component* of the search path contained in the variable *path*, so commands in '.' are found first. At the beginning of a component of a pathname, '.' is treated specially and not matched by the filename expansion metacharacters '?', '*', and '[' ']' pairs. The character '.' is also used in separating filename components.

.. Each directory has a file '..' in which is a reference to its parent directory. After changing into a directory with *cd*, you can return to the parent directory by typing **cd ..**. Then check the current directory with *pwd*.

a.out The default file that contains the executable images that compilers create.

absolute pathname

A *pathname* which begins with a '/' is *absolute* since it specifies the *path* of directories from the beginning of the entire directory system – called the *root* directory. *Pathnames* which are not *absolute* are called *relative* (see *relative pathname*).

alias An *alias* specifies a shorter or different name for a Sun system command, or a command transformation to be performed in the Shell. The Shell has an *alias* command that establishes *aliases* and can show their current values. The command *unalias* removes *aliases*.

argument

Commands in the Sun system receive a list of *argument* words. Thus the command *echo a b c* consists of the command name *echo* and three argument words 'a', 'b' and 'c'. The set of arguments after the command name is the *argument list* of the command.

background

Commands started without waiting for them to complete.

base That part of a filename before any '.' character. See also *filename* and *extension*.

bg The *bg* command causes a suspended job to continue execution in the background.

bin A directory containing binaries of programs and Shell scripts to be executed. The standard system *bin* directories are */bin*, containing the most heavily used commands and */usr/bin*, which contains most other user programs. Programs developed at UC Berkeley live in */usr/ucb*, while locally written programs live in */usr/local*. Games are kept in the directory */usr/games*. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a *component* of the variable *path*.

built-in A command that the Shell executes directly. Most commands in the Sun system are not built into the Shell, but rather exist as files in *bin* directories.

cat The *cat* program concatenates a list of specified files on the standard output.

cd The *cd* command changes the *working directory*. With no arguments, *cd* changes your *working directory* to be your *home* directory.

chsh The *chsh* command changes the Shell which you use on the Sun system. By default, you use the C-Shell which resides in */bin/csh*.

command

A function performed by the system, either by the Shell (a built-in *command*) or by a program residing in a file in a directory within the Sun system.

command name

When a command is issued, it consists of a *command name*, which is the first word of the command, followed by arguments. The convention is that the first word of a command names the function to be performed.

component
The part of a pathname between '/' characters. A variable which has multiple strings as a value is said to have several *components*; each string is a component of the variable.

continue The built-in command that causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C.

control- Certain special characters, called *control* characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper-case characters. To produce *control-c* (or ^C), hold down the CONTROL key while pressing the C key. Usually the Sun system shows a caret (^) followed by the corresponding letter when you type a control character.

core dump
When a program terminates abnormally, the system places an image of its current state in a file named *core*. This *core dump* can be examined with the system debugger *adb* or *dbx* to determine what went wrong with the program. The Shell may produce a message of the form 'Illegal instruction (core dumped)' where 'Illegal instruction' is only one of several possible messages.

cp The *cp* (copy) program copies the contents of one file into another file.

csh The name of the Shell program for the C-Shell.

.cshrc The file *.cshrc* in your home directory that each Shell reads as it begins execution. It is usually used to change the setting of the variable *path* and to set alias parameters which are to take effect globally.

cwd The *cwd* variable in the Shell that holds the absolute pathname of the current working directory. The Shell changes it whenever your current working directory changes, and it should not be changed otherwise.

date The *date* command prints the current date and time.

debugging
The process of correcting mistakes in programs and Shell scripts. The Shell has several options and variables which may be used to aid in Shell debugging.

DELETE The DELETE or RUBOUT key on a terminal normally sends an INTERRUPT to the current job. The workstation default is ^C.

detached
A process that continues running in the background after you logout.

diagnostic
An error message produced by a program. Most error messages are written to the diagnostic output, which may be directed away from the workstation, but usually is not, so diagnostics usually appear on the workstation.

directory
A structure which contains files. At any time you are in one particular *directory* whose name can be shown by the command *pwd*. The *cd* command changes you to another *directory*, and makes the files in that directory visible. The directory which you are in when you first log in is your *home* directory.

directory stack
The C-Shell saves the names of previous working directories in the *directory stack* when you change your current working directory with the *pushd* command. To display the *directory stack* use the *dirs* command, which includes your current working directory as the first directory name on the left.

dirs The *dirs* command displays the C-Shell's directory stack.

du Short for 'disk usage,' the *du* program shows the number of disk blocks in all directories below and including your current working directory.

echo The *echo* command displays its arguments.

EOF An *end-of-file* is generated by a ^D, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a pipe receive an end-of-file when the command sending them input completes. Most commands terminate when they receive an EOF.

escape A character ' \' that prevents the special meaning of a metacharacter; it *escapes* the metacharacter from its special meaning.

/etc/passwd
The file containing information about the accounts currently on the system. The */etc/passwd* file consists of a line for each account with fields separated by ':' characters. See *passwd*.

expansion
The replacement of strings in the Shell input which contain metacharacters by other strings. The replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion.' Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion.'

extension
Filenames often consist of a *base* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same *root* name. Thus if *prog.c* were a C program, then the object file for this program would be stored in *prog.o*. Similarly a paper written with the *-ms* nroff macro package might be stored in *paper.ms*, while a formatted version of this paper might be kept in *paper.out* and a list of spelling errors in *paper.errs*.

fg The job control command that runs a background or suspended job in the *foreground*.

filename
Each file in the Sun system has a name consisting of characters, not including the character '/', which is used in pathname building. Most *filenames* do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the base portion of the filename from an extension.

filename expansion
Filename expansion uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. You can name all the files in the current directory, or all files which have a common root name. Other filename expansion mechanisms use the metacharacter '~' and allow files in other users' directories to be easily named.

flag Many Sun system commands accept arguments which are not the names of files or other users, but modify the action of the commands. These are referred to as *flag* options, and by convention consist of one or more letters usually preceded by the character '–'. Thus the *ls* (list files) command has an option –s to list the sizes of files.

foreground
When commands are executing in the normal way such that the Shell is waiting for them to finish before prompting for another command they are said to be *foreground jobs* or *running in the foreground*, as opposed to background. Typing different control characters at the keyboard stops foreground jobs.

grep The *grep* command searches through a list of argument files for a specified string. *Grep* scans for *regular expressions* in the sense of the editors *ed* and *ex*. *Grep* stands for 'global regular expression print.'

head The *head* command shows the first few lines of one or more files. *Head* also describes the part of a pathname before and including the last '/' character.

header field
At the beginning of a message, a line that contains information that is part of the structure of the message. Header fields include *to, cc,* and *subject.*

history The *history* mechanism of the Shell repeats previous commands, possibly after modification to correct typing mistakes or to change the meaning of the command. The Shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is.

home directory
Each user has a *home directory,* which is defined in his entry in the password file, */etc/passwd.* This is the directory which you are placed in when you first log in. The *cd* command with no arguments takes you back to this directory, whose name is recorded in the Shell variable *home.* You can also access the home directories of other users by forming filenames using a filename expansion notation and the character '~'.

ignoreeof
Normally, your Shell will exit, displaying 'logout' if you type a ^D at a prompt of 'tutorial%'. This is the way you usually log off the system. You can set the *ignoreeof* variable if you wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many ^D's characters, logging yourself off.

input The information that many Sun system commands take from the workstation or from files and act on. Commands normally read for input from their *standard input* which is, by default, the workstation or terminal. This standard input can be redirected from a file using the Shell metanotation character '<'. Many commands also read from a file specified as argument. Commands placed in pipelines read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a filename to use as standard input. Special mechanisms exist for supplying input to commands in Shell scripts.

interrupt
A signal to a program that is generated by typing ^C (or the RUBOUT or DELETE key on some terminals), which causes most programs to stop execution. Certain programs, such as the Shell and the editors, handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While executing another command and waiting for it to finish, the Shell does not listen to interrupts. Typing an interrupt often wakes up the Shell as many commands die when interrupted.

job One or more commands typed on the same input line separated by '|' or ';' characters and run together. Simple commands run by themselves without any '|' or ';' characters are the simplest jobs. Jobs are classified as foreground, background, or suspended.

job control
The built-in functions that control the execution of jobs. These functions are *bg, fg, stop, kill.*

job number
When each job is started it is assigned a small *job number,* which is displayed next to the job in the output of the *jobs* command. Use this number, preceded by a '%' character, as an argument to job control commands to indicate a specific job.

jobs The *jobs* command displays a table showing jobs that are either running in the background or are suspended.

kill A command which sends a signal to a job causing it to terminate.

.login The file *.login* in your *home* directory is read by the Shell each time you log in to the Sun system, and the commands contained there are executed. A number of commands are usually placed in *.login*, especially *set* commands to the Shell itself.

login Shell
The Shell that is started on your terminal when you log in. It is different from other Shells which you may run (such as on Shell scripts) in that it reads the *.login* file before reading commands from the workstation or terminal, and it reads the *.logout* file after you logout.

logout The *logout* command causes a login Shell to exit. Normally, a login shell exits when you type ^D generating an end-of-file, but if you have set *ignoreeof* in your *.login* file, this will not work and you must use *logout* to log off the Sun system.

.logout When you log off the Sun system, the Shell executes commands from the file *.logout* in your home directory after it displays 'logout.'

lpr The line printer daemon command. The standard input of *lpr* is spooled and printed on the line printer. You can also give *lpr* a list of filenames as arguments to be printed.

ls With no argument filenames, *ls* shows the names of the files in the current directory. It has a number of useful flag arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories.

mail The *mail* program sends and receives messages from other system users.

mailbox The place where your mail is stored, typically in the directory */usr/spool/mail.*

message A single letter from someone, initially stored in your *mailbox.*

message list
A string used in *mail* command mode to describe a sequence of messages.

metacharacter
The characters that are neither letters nor digits that have special meaning either to the Shell or to the Sun system. Enclose them in quotes if it is necessary to place these characters in arguments to commands without them having their special meaning. An example of a metacharacter is the character '>' which indicates placement of output into a file. For the purposes of the *history* mechanism, most unquoted metacharacters form separate words.

mkdir The *mkdir* command creates a new directory.

modifier
Substitutions with the *history* mechanism, keyed by the character '!' or of variables using the metacharacter '$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the *modifier* itself.

more The program *more* shows a file on your workstation and allows you to control how much text is displayed at a time. *More* can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file.

noclobber
The Shell variable that prevents accidental destruction of files by the '>' output redirection metasyntax of the Shell if set in the file *.login*

noglob The Shell variable that suppresses the filename expansion of arguments containing the metacharacters '~', '*', '?', '[' and ']'.

notify The *notify* variable tells the Shell to report on the termination of a specific background job at the exact time it occurs as opposed to waiting until just before the next prompt to report the termination. If set, the *notifyfP variable causes the Shell to always report the termination of background jobs exactly when they occur.*

output The lines of text resulting from many Sun system commands. This output is usually placed on what is known as the standard output, which is normally connected to the user's workstation. The Shell has a syntax using the metacharacter '>' for redirecting the *standard output* of a command to a file. Using the pipe mechanism and the metacharacter '|' it is also possible for the standard output of one command to become the standard input of another command. Certain commands such as the line printer daemon *lpr* do not place their results on the standard output but rather on the line printer. Similarly the *write* command places its output on another user's workstation rather than its own standard output. Commands also have a diagnostic output where they write their error messages. Normally these go to the workstation even if the standard output has been sent to a file or another command, but it is possible to direct error diagnostics along with standard output using a special notation.

pushd The *pushd* command, which means 'push directory', changes the Shell's working directory and also remembers the current working directory before the change is made, so you can return to the same directory with the *popd* command later without retyping its name.

path The Shell variable that gives the names of the directories in which it searches for the commands which it is given. *Path* always checks first to see if the command it is given is built into the Shell. If it is, then it need not search for the command as it can do it internally. If the command is not built in, the Shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is . */usr/ucb /bin /usr/bin*, the Shell normally looks in the current directory, and then in the standard system directories */usr/ucb*, */bin* and */usr/bin* for the named command. If the command cannot be found the Shell displays an error diagnostic. Scripts of Shell commands are executed using another Shell to interpret them if they have 'execute' permission set. If you add new commands to a directory in the *path*, use the command *rehash*.

pathname A list of names, separated by '/' characters. Each component of a pathname, between successive '/' characters, names a directory in which the next component file resides. Pathnames which begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other pathnames are interpreted relative to the current directory as reported by *pwd*. The last component of a pathname may name a directory, but usually names a file.

pipeline A group of commands connected together, the standard output of each being connected to the standard input of the next. The Shell metacharacter '|' indicates the pipe mechanism.

popd The *popd* command changes the Shell's working directory to the directory you most recently left using the *pushd* command. It returns to the directory without having to type its name, forgetting the name of the current working directory before doing so.

port The part of a computer system to which each terminal is connected is called a *port*. Usually the system has a fixed number of ports, some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.

pr The *pr* command prepares listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified.

printenv The *printenv* command prints the current setting of variables in the environment.

process An instance of a running program. The system assigns each process a unique *process id number* when it is started. Use process id numbers to stop individual processes using the *kill* or *stop* commands when the processes are part of a detached background job.

program Usually synonymous with command; a binary file or Shell command script that performs a useful function.

prompt The indication by a program on the screen that it is ready to accept input. The Shell prompts for input with 'hostname% ' and occasionally with '?' when reading commands from the workstation. The Shell has a variable *prompt* which may be set to a different value to change the Shell's main prompt. This is mostly used when debugging the Shell.

ps The *ps* command shows processes you are currently running, each process being shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), or whether it is swapped out), and the amount of CPU time it has used so far. A command is identified by printing some of the words used when it was invoked. Shells, such as the *csh* you use to run the *ps* command, are not normally shown in the output.

pwd The *pwd* command displays the full pathname of the current working directory, similar to the *dirs* built-in command.

quit The signal generated by a control-\ (^\), that terminates programs which are behaving unreasonably. *Quit* normally produces a core image file.

quoting The process by which metacharacters are prevented their special meaning, usually by using the character '' ' in pairs, or by using the character '\'.

redirection
The routing of input or output from or to a file.

rehash The *rehash* command tells the Shell to rebuild its internal table of which commands are found in which directories in your path. This is necessary when a new program is installed in one of these directories.

relative pathname
A pathname that does not begin with a '/' since it is interpreted relative to the current working directory. The first component of such a pathname refers to some file or directory in the working directory, and subsequent components between '/' characters refer to directories below the working directory. See also *absolute pathnames.*

repeat The *repeat* command iterates another command a specified number of times.

root The directory that is at the top of the entire directory structure; it is the 'root' of the entire tree structure of directories. The '/' indicates the *root* name in pathnames. Pathnames starting with '/' are absolute since they start at the *root* directory. Root is also used as the part of a pathname that is left after removing the extension. See *filename* for a further explanation.

scratch file
Files whose names begin with a '#' and are automatically removed by the system after a couple of days of non-use, or more frequently if disk space becomes tight.

set The built-in command that assigns new values to Shell variables and shows the values of the current variables. Many Shell variables have special meaning to the Shell itself, so using the *set* command can affect the behavior of the Shell.

setenv The built-in command that changes variables in the environment 'environ'. The *printenv* command displays the value of the variables in the environment.

shell A command language interpreter. It is possible to write and run your own Shell, as Shells are no different than any other programs as far as the system is concerned.

signal A short message that is sent to a running program which affects that process. Signals are sent either by typing special control characters on the keyboard or by using the *kill* or *stop* commands.

sort The *sort* program sorts a sequence of lines in ways that you can control with argument flags.

source The *source* command reads commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them.

special character
See *metacharacters*.

standard
Used as in the *standard input* and *standard output* of commands. See *input* and *output*.

status A command normally returns a *status* when it finishes. By convention a status of zero indicates that the command succeeded. Commands may return non-zero status to indicate that some abnormal event has occurred. The Shell variable *status* is set to the status returned by the last command. It is most useful in Shell command scripts.

stop The *stop* command suspends a background job.

string A sequential group of characters taken together. Strings can contain any printable characters.

stty The *stty* program changes certain parameters inside the Sun system to determine how your terminal is handled. See *stty* in the user's manual for a complete description.

substitution
The Shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history substitution keyed by the metacharacter '!' and variable substitution indicated by '$'. We also refer to substitutions as *expansions*.

suspended
A job becomes *suspended* after a STOP signal is sent to it, either by typing a ^Z (for foreground jobs) or by using the *stop* command (for background jobs). When suspended, a job temporarily stops running until it is restarted by either the *fg* or *bg* command.

termination
Occurs when a command which is being executed finishes. Commands normally terminate when they read an end-of-file from their standard input. It is also possible to terminate commands by sending them an interrupt or quit signal. The *kill* program terminates specified jobs.

time Measures the amount of CPU and real time used by a specified command as well as the amount of disk I/O, memory utilized, and number of page faults and swaps taken by the command.

tset The *tset* program sets standard erase and kill characters and tells the system what kind of terminal you are using. It is often invoked in a *.login* file.

tty The historical abbreviation for 'teletype' which is frequently used in the Sun system to indicate the port to which a given workstation is connected. The *tty* command displays the name of the tty or port to which your workstation or terminal is presently connected.

unalias The *unalias* command removes aliases.

UNIX The operating system on which the Sun system is based.

unset The *unset* command removes the definitions of Shell variables.

variable expansion
See *variables* and *expansion.*

variables
Contain one or more strings as value and control the behavior of the Shell.

verbose A Shell variable that echoes commands after they are history expanded. This is often useful in debugging Shell scripts. The *verbose* variable is set by the Shell's –v command line option.

wc The *wc* program counts the number of characters, words, and lines in the files whose names are given as arguments.

word A sequence of characters which forms an argument to a command. Many characters which are neither letters, digits, '–', '.' nor '/' form words all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a word by surrounding it with '' characters except for the characters '' and '!' which require special treatment. This process of placing special characters in words without their special meaning is called *quoting.*

working directory
The particular directory you are in at any given time. *Pwd* displays this directory's name and *ls* lists its files. You can change working directories using *cd.*

write The *write* command communicates with other users who are logged in to the system.

APPENDIX B: BIBLIOGRAPHY

General Sun System Reference

User's Manual for the Sun Workstation, Sun Microsystems Inc. The Sun system command reference manual.

The UNIX System, S. R. Bourne, Addison-Wesley Publishing Co., 1982. A comprehensive practical introduction for users from novices to experts.

The UNIX Operating System, Kaare Christian, John Wiley and Sons, 1983. Geared to the beginning user in Part One and the advanced user in Part Two, provides system basics as well as extensive coverage of the Version 7 Shell, internal system organization, information for programmers and managers, and introductions to several utility programs.

Introducing the UNIX System, Henry McGilton and Rachel Morgan, McGraw-Hill Book Company, 1983. An introduction to UNIX for beginners and more sophisticated users. Covers the usual, plus communication facilities, editors, document formatting, software tool development, Berkeley UNIX, and system management. Packed with helpful examples.

UNIX Primer Plus, Mitchell Waite, Donald Martin, Stephen Prata, Howard W. Sams and Co., Inc., 1983. Provides hands-on examples for both Berkeley and Bell Labs UNIX.

A User Guide to the UNIX System, Jean Yates and Rebecca Thomas, Osborne/McGraw-Hill, 1982. A tutorial introduction to the 40 most used system commands.

The UNIX Time-sharing System, D. M. Ritchie and K. L. Thompson, CACM, July 1974. An overview of the UNIX system for people interested in operating systems and worth reading by anyone who programs. Contains a remarkable number of one-sentence observations on how to do things right.

The Bell System Technical Journal (BSTJ) Special Issue on UNIX, July/August, 1978. Contains many papers describing recent developments, and some retrospective material.

The 2nd International Conference on Software Engineering (October, 1976). Contains several papers describing the use of the Programmer's Workbench (PWB) version of UNIX.

Document Preparation

Editing and Text Processing on the Sun Workstation, Sun Microsystems Inc. Tutorial and reference material on the editors and text processors.

Software Development Tools

System Interface Manual for the Sun Workstation, Sun Microsystems Inc. Contains system calls, library functions, and file formats and is of particular interest to programmers.

Programming Tools for the Sun Workstation, Sun Microsystems Inc. Contains information of general interest to anyone using the Sun system to write programs.

Fortran and Pascal for the Sun Workstation, Sun Microsystems Inc. Information specific to the Fortran and Pascal programming languages.

SunCore for the Sun Workstation, Sun Microsystems Inc. Describes the SunCore graphics package.

The C Programming Language, B. W. Kernighan and D. M. Ritchie, Prentice-Hall, 1978. Contains a tutorial introduction, complete discussions of all language features, and a reference manual.

Shell Reference

The UNIX Shell, S. R. Bourne, Bell System Technical Journal, July-August 1978, Volume 57, Number 6, Part 2. An introduction to the Bourne Shell and how to program it.

READER COMMENT SHEET

Dear Customer,
We who work here at Sun Microsystems wish to provide the best possible documentation for our products. To this end, we solicit your comments on this manual. We would appreciate your telling us about errors in the content of the manual, and about any material which you feel should be there but isn't.

Typographical Errors:

Please list typographical Errors by page number and actual text of the error.

Technical Errors:

Please list errors of fact by page number and actual text of the error.

Content:

Did this guide meet your needs? If not, please indicate what you think should be added or deleted in order to do so. Please comment on any material which you feel should be present but is not. Is there material which is in other manuals, but would be more convenient if it were in this manual?

Layout and Style:

Did you find the organization of this guide useful? If not, how would you rearrange things? Do you find the style of this manual pleasing or irritating? What would you like to see different?