MONDAY 15TH JUNE, 2009

UNIX AND COMMUNICATIONS HAND BOOK.

THIS WAS NOT REQUIRED BY ENGINERING AS SOFTWARE WAS DONE BY COMPUTING

Turn off

login as root (password Escargot)

shutdown -h now

-r (reboot)

AUSTRALIAN GOVERNMENT
DEPARTMENT OF ADMINISTRATIVE SERVICES

IPS RADIO AND SPACE SERVICES

HANDBOOK 3 OF 3

FOR

IONOSONDE TYPE 4B/D

UNIX CONTROL AND COMMUNICATIONS COMPUTER
FOR IPS IONOSONDE 4D

V2.1
ISSUED

AUGUST 1997

HANDBOOK: Richard Luckhurst
DESIGN: Craig Bevins

IPS RADIO AND SPACE SERVICES
Level 4, 15 HELP ST, CHATSWOOD NSW 2067
PO BOX 5606, WEST CHATSWOOD NSW 2067
Telephone: (02) 9414 8334 Fax: (02) 9414 8331

CONTENTS

1. INTRODUCTION

For a number of years IPS Radio and Space Services has operated 4B Ionosondes throughout Australia and Antarctica. Until 1993 the Ionograms, collected by these instruments, were recorded on 16mm film for later analysis. The use of film meant that there was a considerable delay between the ionogram being recorded and the data arriving at the IPS Head Office. Continued use of film would mean that a 'Real Time' data network could not be established.

In 1993 a program to install the Digion conversion on the network of 4B Ionosondes commenced. This program would see the end of film but did little to get the data back to IPS any quicker than if film were still being used. Throughout 1994 data files were transferred back electronically, on a regular basis, from the Antarctic sites but this was slow and meant considerable effort was required.

In mid 1994 a new system was proposed by Craig Bevins, at that time the IPS Radio and Space Services IT Manager, which would see the data coming back from Antarctica each day with no interaction required by any staff. The system featured a second computer, running the UNIX operating system, to collect the daily data files, from the digion, and send them to IPS Head Office. Three of these systems were installed in Antarctica in the summer of 1994 and IPS started receiving daily ionogram files.

Since the initial installation considerable effort has gone into improving the performance of these 'communication computers'. A new software package, for the DOS Digion PC, was written in 1995 which allows files to be retrieved faster than once each day and also for the configuration files on the DOS PC to be changed remotely.

2. THE COMMUNICATIONS COMPUTER

To keep the cost down and to allow for easy replacement of components a 486 PC was chosen as the communications computer. This computer was fitted with a Digital Audio Tape (DAT) drive to allow for local backups of ionosonde data. To allow for easy networking and good multi-tasking a good operating system was needed.

At this time a number of 'free' UNIX and UNIX-like operating systems were appearing for the i386 architecture, the NetBSD operating system was chosen. NetBSD is a Berkeley Networking Release 2 (Net/2) and 4.4 BSD-Lite derived system. It has been ported to a number of architectures and was a creation of members of the international networking community. The operating system feature support for the industry standard X-Windows environment.

The computers initially chosen were DEC 486 PC's with 8 Mbytes of RAM and a SuperVGA graphics card. A DAT drive and SCSI card were installed for local backups and an Ethernet card was installed to allow for a connection to the station Local Area Network (LAN).

The DOS Digion PC was connected to the UNIX Communications PC via a simple serial link. The ionogram files are transferred, once each day, using the 'kermit' communications program. After receiving the files they are processed, to allow for easy viewing, and then late at night transferred to IPS Head Office.

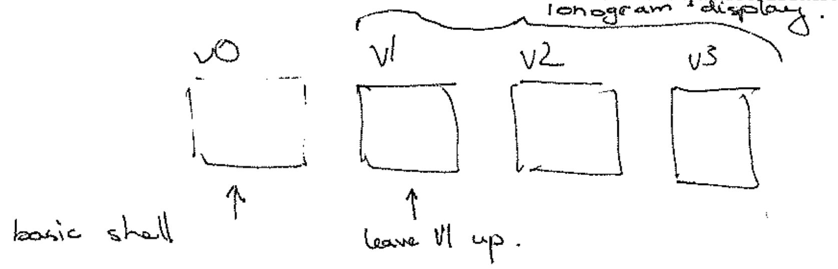
An account has been established on this computer, called **digion** with the password of **prediction**, for the day to day operation of the system.

2.1 MACQUARIE ISLAND

During 1996 it was decide that an attempt would be made to link the Macquarie Island Ionosonde hut to the system LAN with a higher speed data link. This installation required two 'Communications Computers' with a Spread Spectrum Radio System providing the data link.

The original 'Communications Computer' was replaced by two 75mhz Pentium PC's. The NetBSD operating system was replaced by FreeBSD, which offers better support for interface cards in the i386 architecture.

For further information refer to the 'IPS Macquarie Island Digion LAN' handbook.



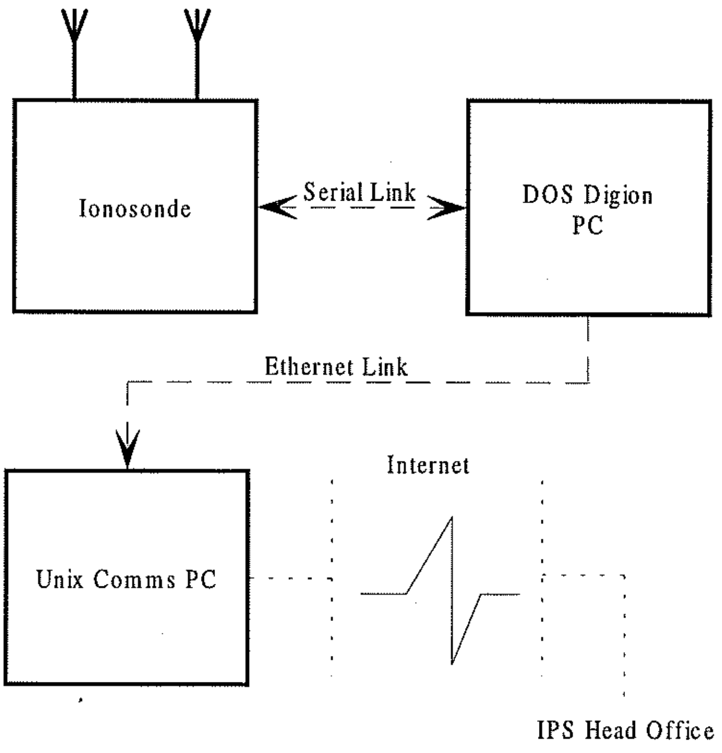
ALT - CTRL - F1/F4

>root.

ts carg[illegible]

3. HOW AN IONOGRAM GETS TO IPS HEAD OFFICE

Simplified connection from Antarctic 4D and IPS Head Office.



There are many steps involved in an ionogram file getting from the Ionosonde to IPS Head Office. There are a number of points along the way that the file may be examined or archived.

1. The ion.bat file calls getion.exe. Ionograms are recorded to a file on the DOS PC.

2. At the end of the day getion.exe exits.

3. The next step in ion.bat archives the day file to local storage. (DAT drive)

4. The file then gets sent, using kermit, to the UNIX PC.

5. The raw file ends up in a directory called raw in the digion account.

6. Some time late a process, called 4d2ips, runs and the file gets converted into the 'standard' IPS format. This allows the ionograms to be examined and scaled using the same software, called plotcln or scale, as IPS Head Office.

7. The raw file gets archived to a DAT tape and gets stored on the local hard drive in an Archive directory under the raw directory.

8. The cleaned file gets stored on the hard disk in a directory tree under cln under site4d where site is mac, maw or davis.

9. At a predetermined time the raw file gets transferred to IPS Head Office. This is usually late at night when the link to Australia in quiet.

10. The next morning a process runs in Sydney and the raw file is archived and converted to the 'standard' IPS format for scaling.

    Note: All storage after format conversion is done in a compressed format.

11. Another process runs each day cleaning up the number of raw files on the system. Only the last 90 days are kept on the system.

The ionogram data has been archived at a number of points along the way and it can be viewed, if desired, at a number of points.

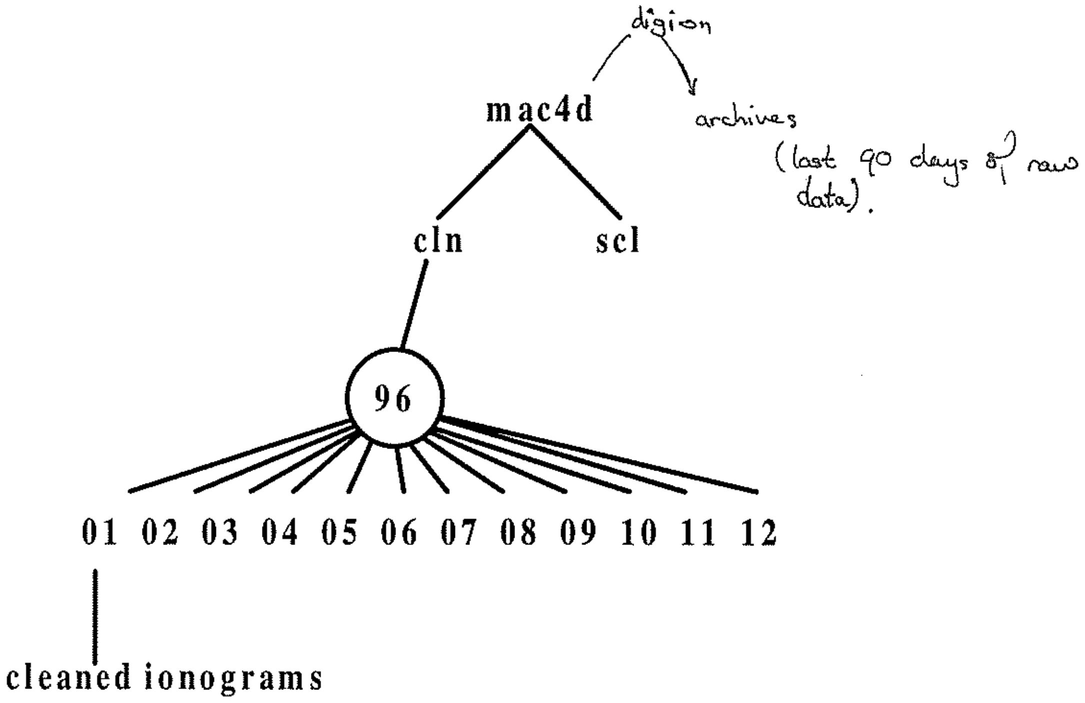
1. The first place the data gets stored is on the DOS PC. There is an archive directory where the data is stored in zip format. The raw data is also written to a local DAT tape. The digion.exe program can be used to view the raw data files and if needed the pkunzip program is installed in the c:\bin directory.

   NOTE: At some sites the DAT drive has been commented out of ion.bat. Strictly speaking it is not really needed as the data should get transferred. The DAT drive has been left in place in case the UNIX PC fails.

2. The raw file is only held briefly in the raw directory in the digion account. It is not possible to view the raw files on the UNIX PC.

3. The raw file, now compressed is stored in the Archive directory under the raw directory.

4. In the site4d directory there is the following directory tree:



These cleaned ionograms can be viewed using the scale program in the X-Windows environment, in the digion account.

4. BACKUPS

There are no routine backups to be done on the DOS Digion PC. Each installation was supplied with a floppy disk containing the system files and this can be used to restore the installation if needed.

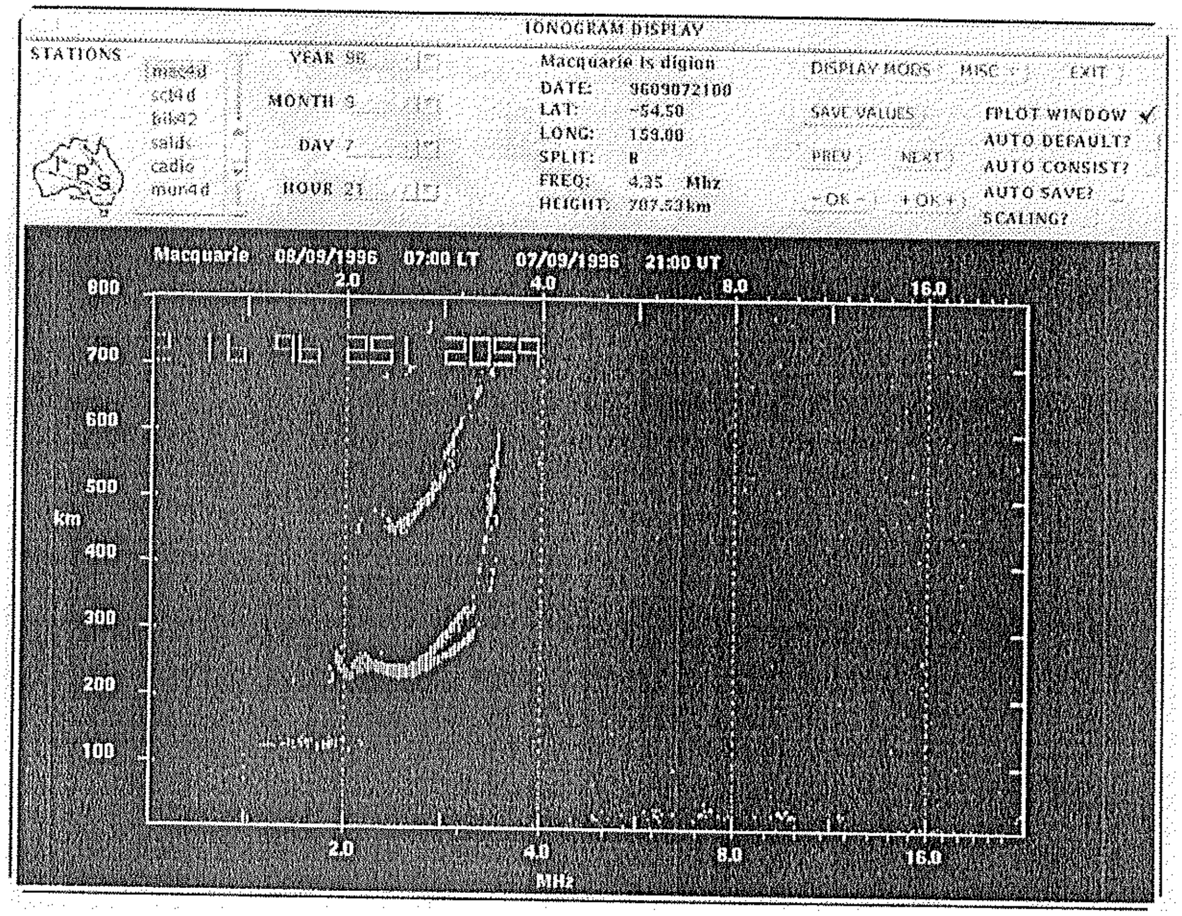
Do ~ 4 weeks.

There is a need to perform regular backups on the UNIX PC. To make this easier a special account has been created called **dumps**. If you log into the UNIX computer with the username of **dumps** no password will be asked for . A program will start and you will be guided through the backup procedure. This process will require about four DAT tapes. This should be done on a regular basis and if it is forgotten for a long period IPS IT Section will get a piece of email warning that the backup has not been done.

As there are a lot of customised configuration files in the UNIX system these backup tapes are invaluable in restoring the system if there is a problem.

5. IPS SCALING PROGRAM

Description of items on the scaling control panel.


IONOGRAM DISPLAY

STATIONS: mac4d, scl4d, bdk42, sal4c, cad4e, mun4d

YEAR 96
MONTH 9
DAY 7
HOUR 21

Macquarie Is digion
DATE: 9609072100
LAT: -54.50
LONG: 159.00
SPLIT: B
FREQ: 4.35 Mhz
HEIGHT: 707.53km

DISPLAY MODS · MISC · EXIT
SAVE VALUES
PREV · NEXT
-OK- · +OK+

FPLOT WINDOW ✓
AUTO DEFAULT?
AUTO CONSIST?
AUTO SAVE?
SCALING?

Macquarie 08/09/1996 07:00 LT 07/09/1996 21:00 UT

The leftmost item (labelled **STATIONS**) on the scaling display package is a scrolling list of the available stations. The current station is selected by clicking with the left mouse button on the station abbreviation. Stations are abbreviated to 5 letters. The first 3 usually denote the station and the last 2 the type of ionosonde.

To the right of this is the date. It defaults to the current time on startup. The date can be changed by clicking with left mouse button the up and down arrows associated with each field. The date can also be modified by clicking on or after a number, backspacing, typing in a new number and hitting enter.

The area on the control panel displays information about the current mode. The **FREQ** and **HEIGHT** fields display the frequency and height at the point where the left mouse button was clicked in the display.

Now proceeding from left to right and top to bottom.

DISPLAY MODS brings up the display mods window. This window will be discussed later.

MISC is a menu button. The user opens the menu by clicking on MISC. The desired item is selected by clicking on it. If there is an arrow to the right of a menu item then there is a pullright menu associated with the item. Move the mouse right to bring up the menu.

EXIT - exits the application.

SAVE VALUES saves the scaled values for this day. Values are saved automatically when the day or station changes.

FPLOT shows a graph of scaled parameters for the current day. The fplot window is discussed below.

PREVIOUS will search for the previous ionogram in the current month.

NEXT will search for the next ionogram in the current month.

DEFAULTS selects whether to use parameter defaults when scaling. Defaults are supplied when scaling of the current ionogram is finished, denoted by changing ionograms. When selected this box will contain a tick.

-OK- +OK+ are used when the scaler is checking autoscaled data. These buttons will only work when the scale option is enabled. -OK- finds the previous ionogram, and marks the current one as validated. +OK+ goes to the next ionogram and marks the current one as validated.

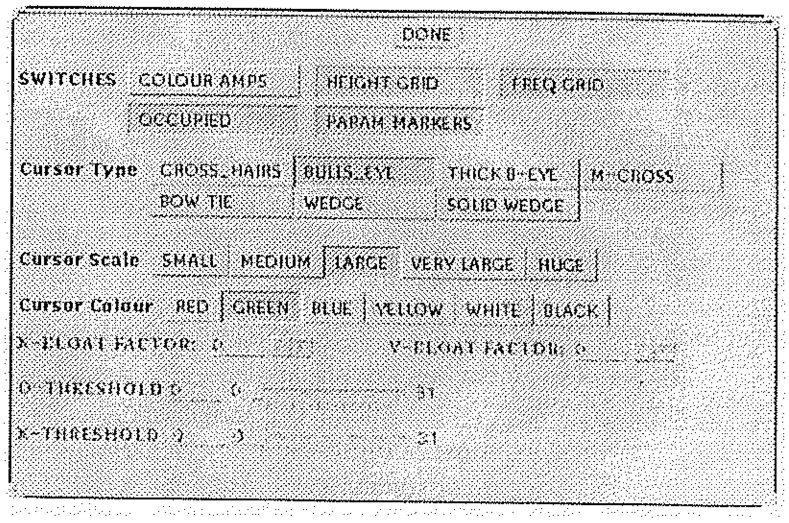
SCALE selects whether you are scaling or just reviewing. Autoscaling will only work if **SCALE** is set. When selected the box will contain a tick.

The following keys also perform the following actions:

'a' is the same as -OK-
'b' is the same as +OK+
'w' is minus 1 hour
'e' is plus 1 hour

Description of Windows

Display Mods Window

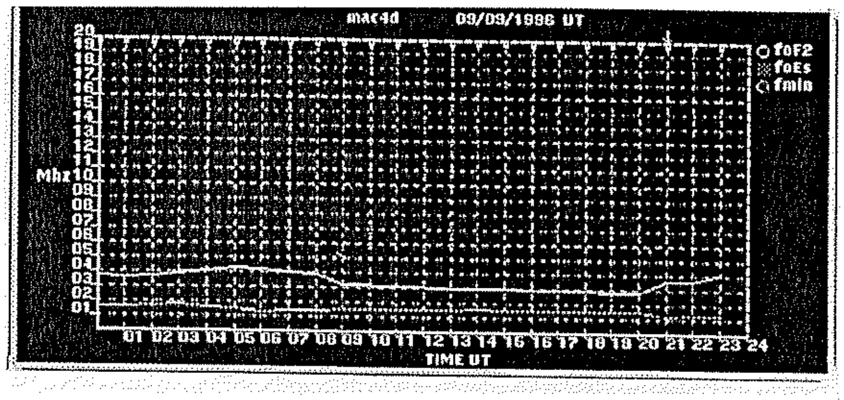


X-BLOAT FACTOR: 0
Y-BLOAT FACTOR: 0
O-THRESHOLD 0 0 31
X-THRESHOLD 0 0 31

The **DISPLAY MODS** window allows the user to modify certain aspects of the display **X-BLOAT** and **Y-BLOAT** simply bloat the drawing pixel in the X and Y axis by the amount selected using the increment/decrement arrows. Bloating only occurs when **COLOUR AMPS** has been selected. **O-THRESHOLD** and **X-THRESHOLD** are only useful when the ionogram contains amplitude information.

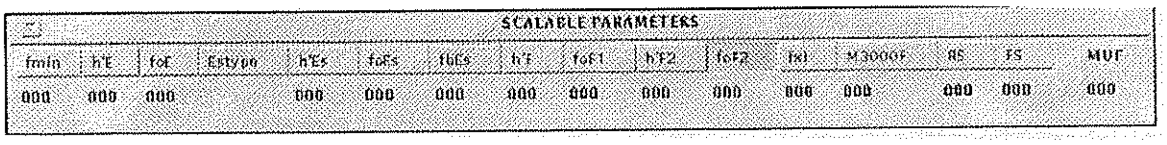
It is recommended the **COLOUR AMPS** is left off as the redraw of the screen is quite slow.

Fplot Window



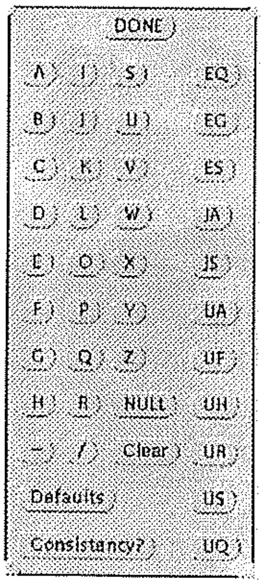
The **FPLOT** window displays frequency versus time plot of the scaled foF2, foEs and fmin values for the current ionogram date. A line joins the values which have an associated cleaned ionogram. Because of its nature foEs values are not joined by lines. Values with no cleaned ionogram appear as circles. At the top of the graph will be an arrow showing the time on the ionogram display. Small squares at the top of the graph indicate the existence of a cleaned ionogram. Clicking with the left mouse button on or near a square will change the currently displayed ionogram to the one at this time. The **FPLOT** window can be removed by clicking on the FPLOT box, to remove the tick, on the main control panel.

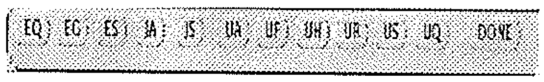
Parameters Window



This window is the long rectangular window which appears when the scaling application is started. It (as will all the windows) can be repositioned by clicking and holding down the left mouse button while the pointer is on the border of the window. It can then be dragged to the desired location. The window consists of a series of selectable buttons, each labelled with a scalable parameter. If values exist for the current ionogram they will be displayed below the appropriate button. When scaling the depressed button is the current parameter being scaled. The parameter can be changed by clicking the left mouse button on the appropriate parameter button, or the scaler can click the right mouse button while the pointer is in the display window.

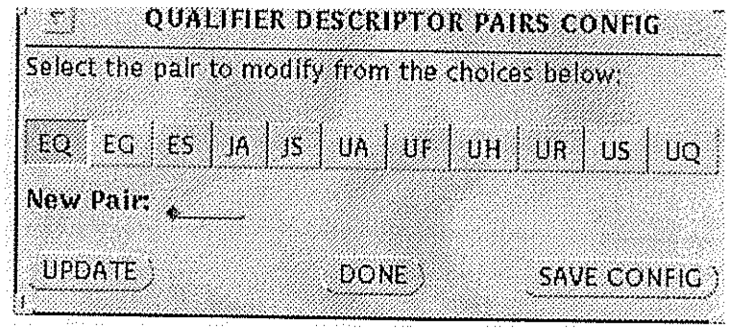
Left, Right and Pairs Qualifier/Descriptor Windows





These windows are selected from the **MISC** menu and are used when scaling to add qualifiers and descriptors to the parameters. Left and Right orientation are provided to minimise cursor movement. The pairs window displays commonly used qualifier/descriptor pairs.

Configure Qual/Desc Pairs Window


QUALIFIER DESCRIPTOR PAIRS CONFIG
Select the pair to modify from the choices below:
EQ
EG
ES
JA
JS
UA
UF
UH
UR
US
UQ
New Pair:
UPDATE
DONE
SAVE CONFIG

This window is used to change the qualifier/descriptor pairs. The pair to be changed is clicked on using the left mouse button. The user then types in the new pair. Clicking on **UPDATE** will add this change to all qualifier/descriptor windows. Clicking on **SAVE** will save this new configuration to the users configuration file. Clicking on **DONE** will quit this window.

How To SCALE

First select the ionogram to be scaled using the increment/decrement arrows on the date fields, or the next/previous buttons. Then make sure that the **SCALE** tick box has been ticked. The **DEFAULTS** tick box may or may not be enabled. If enabled then default values will be assigned to parameters when the user changes to another ionogram. These defaults are based on the scaled parameters. Select the parameter to be scaled (see **PARAMETER WINDOW**). Move the cursor to the desired spot on the ionogram and click the left mouse button. Parameter values can be removed by clicking outside the ionogram display box. Qualifiers and descriptors can be added and removed by using the Qualifier/Descriptor windows.

After all scaling has been completed the user should click on the **SAVE VALUES** button on the control panel and click on the **SCALE** box to remove the tick.

Introduction to Unix

Frank G. Fiamingo

Linda DeBula

Linda Condron

University Technology Services

The Ohio State University

August 14, 1996

The authors' email addresses are:

Frank Fiamingo	fiamingo.1@osu.edu
Linda DeBula	debula.2@osu.edu
Linda Condron	condron.1@osu.edu

This document can be obtained via:

http://www-wks.uts.ohio-state.edu/unix_course/unix.html

or

ftp://www-wks.uts.ohio-state.edu/unix_course/unix_book.ps

Table of Contents

CHAPTER 1 History of Unix

1965 Bell Laboratories joins with MIT and General Electric in the development effort for the new operating system, Multics, which would provide multi-user, multi-processor, and multi-level (hierarchical) file system, among its many forward-looking features.

1969 AT&T was unhappy with the progress and drops out of the Multics project. Some of the Bell Labs programmers who had worked on this project, Ken Thompson, Dennis Ritchie, Rudd Canaday, and Doug McIlroy designed and implemented the first version of the Unix File System on a PDP-7 along with a few utilities. It was given the name UNIX by Brian Kernighan as a pun on Multics.

1970, Jan 1 time zero for UNIX

1971 The system now runs on a PDP-11, with 16Kbytes of memory, including 8Kbytes for user programs and a 512Kbyte disk.

Its first real use is as a text processing tool for the patent department at Bell Labs. That utilization justified further research and development by the programming group. UNIX caught on among programmers because it was designed with these features:

- programmers environment
- simple user interface
- simple utilities that can be combined to perform powerful functions
- hierarchical file system
- simple interface to devices consistent with file format
- multi-user, multi-process system
- architecture independent and transparent to the user.

1973 Unix is re-written mostly in C, a new language developed by Dennis Ritchie. Being written in this high-level language greatly decreased the effort needed to port it to new machines.

1974 Thompson and Ritchie publish a paper in the Communications of the ACM describing the new Unix OS. This generates enthusiasm in the Academic community which sees a potentially great teaching tool for studying programming systems development. Since AT&T is prevented from marketing the product due to the 1956 Consent Decree they license it to Universities for educational purposes and to commercial entities.

1977 There are now about 500 Unix sites world-wide.

1980 BSD 4.1 (Berkeley Software Development)

1983 SunOS, BSD 4.2, SysV

1984 There are now about 100,000 Unix sites running on many different hardware platforms, of vastly different capabilities.

1988 AT&T and Sun Microsystems jointly develop System V Release 4 (SVR4). This would later be developed into UnixWare and Solaris 2.

1993 Novell buys UNIX from AT&T

1994 Novell gives the name "**UNIX**" to X/OPEN

1995 Santa Cruz Operations buys UnixWare from Novell. Santa Cruz Operations and Hewlett-Packard announce that they will jointly develop a 64-bit version of Unix.

1996 International Data Corporation forecasts that in 1997 there will be 3 million Unix systems shipped world-wide.

CHAPTER 2 Unix Structure

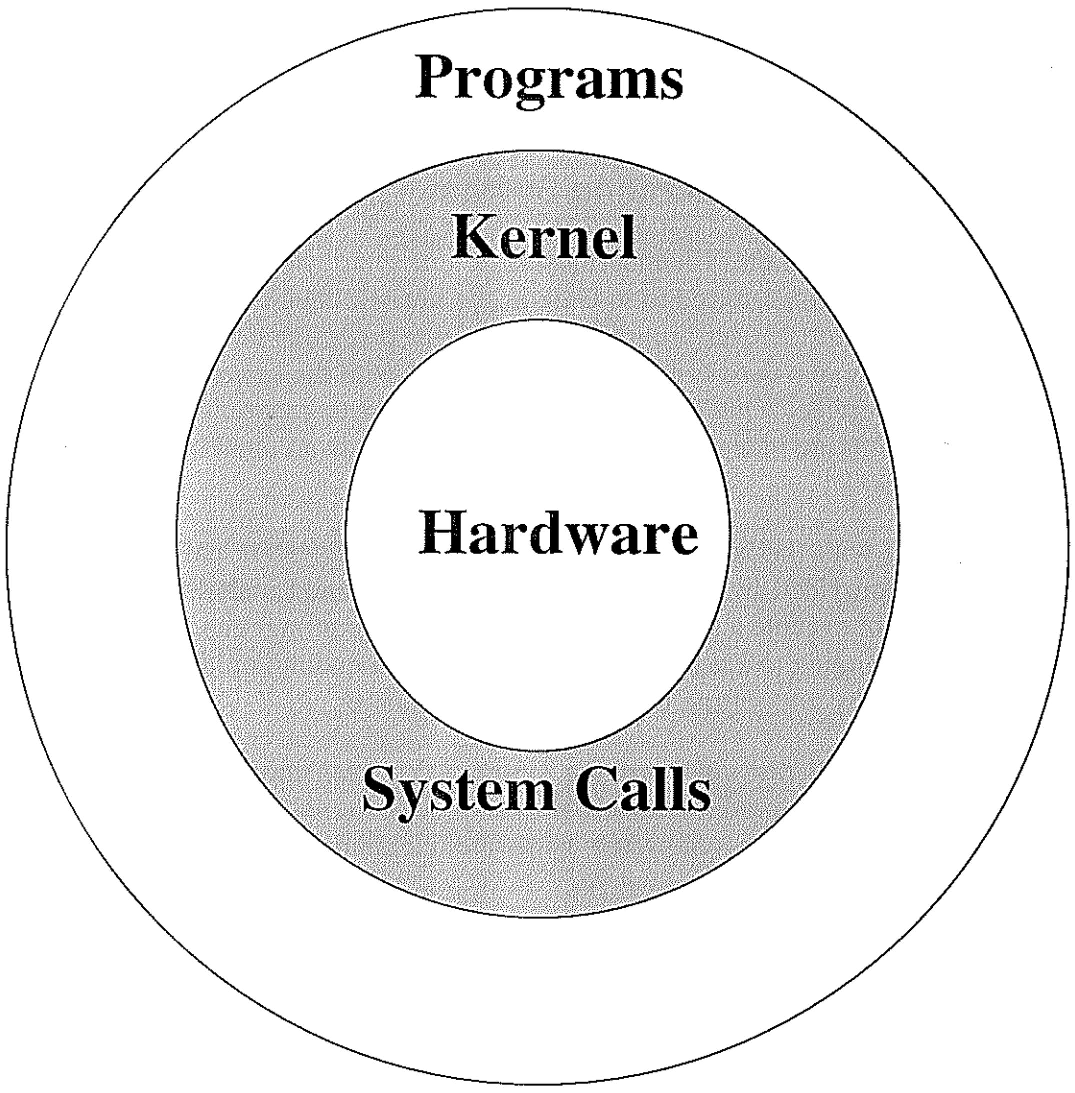
2.1 The Operating System

Unix is a layered operating system. The innermost layer is the hardware that provides the services for the OS. The operating system, referred to in Unix as the **kernel**, interacts directly with the hardware and provides the services to the user programs. These user programs don't need to know anything about the hardware. They just need to know how to interact with the kernel and it's up to the kernel to provide the desired service. One of the big appeals of Unix to programmers has been that most well written user programs are independent of the underlying hardware, making them readily portable to new systems.

User programs interact with the kernel through a set of standard **system calls**. These system calls request services to be provided by the kernel. Such services would include accessing a file: open close, read, write, link, or execute a file; starting or updating accounting records; changing ownership of a file or directory; changing to a new directory; creating, suspending, or killing a process; enabling access to hardware devices; and setting limits on system resources.

Unix is a **multi-user, multi-tasking** operating system. You can have many users logged into a system simultaneously, each running many programs. It's the kernel's job to keep each process and user separate and to regulate access to system hardware, including cpu, memory, disk and other I/O devices.

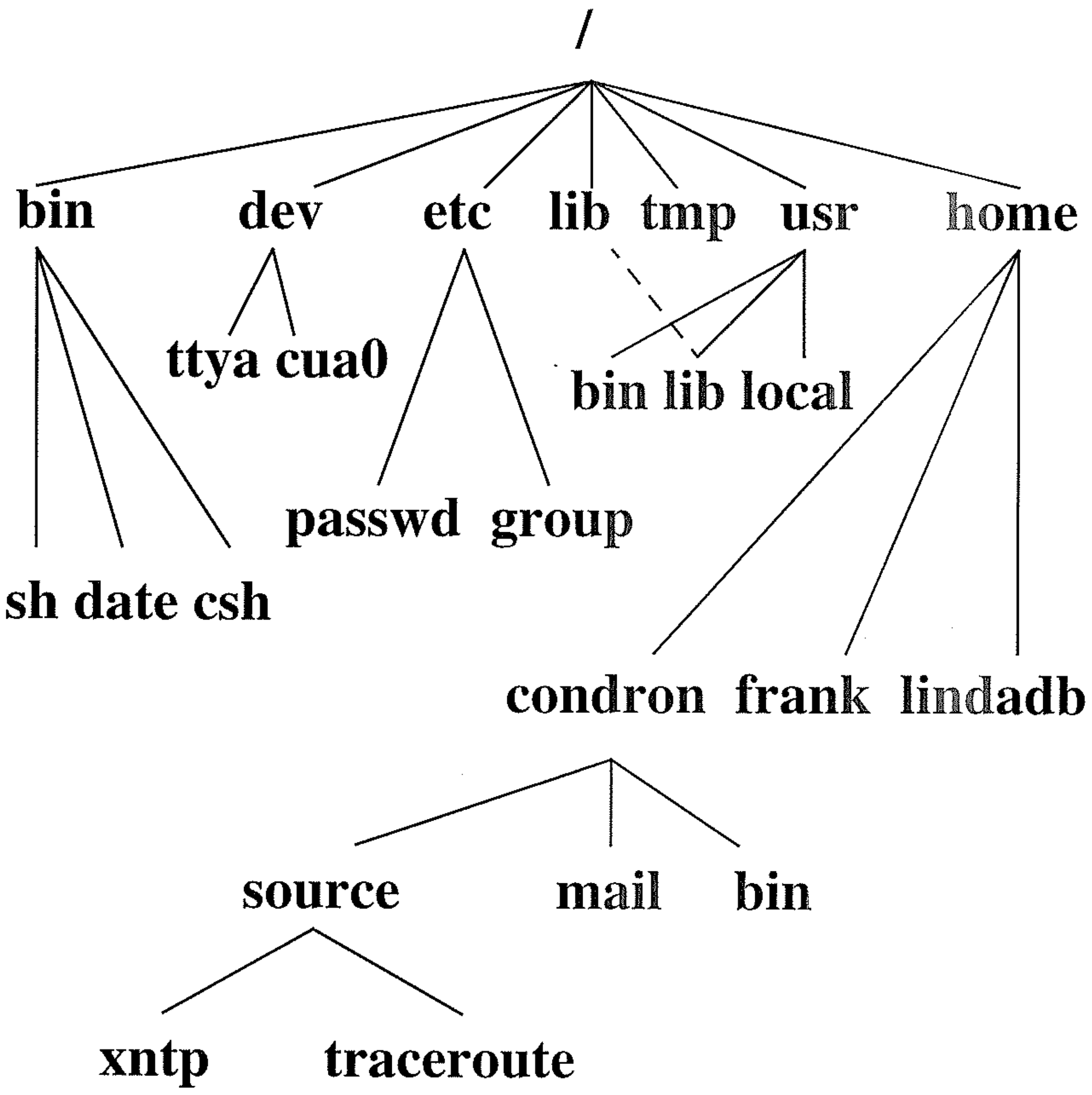
FIGURE 2.1 **Unix System Structure**



2.2 The File System

The Unix file system looks like an inverted tree structure. You start with the **root** directory, denoted by /, at the top and work down through sub-directories underneath it.

FIGURE 2.2 Unix File Structure



Each node is either a **file** or a **directory** of files, where the latter can contain other files and directories. You specify a file or directory by its **path name**, either the full, or absolute, path name or the one relative to a location. The full path name starts with the root, /, and follows the branches of the file system, each separated by /, until you reach the desired file, e.g.:

```
/home/condron/source/xntp
```

A relative path name specifies the path relative to another, usually the current working directory that you are at. Two special directory entries should be introduced now:

.	the current directory
..	the parent of the current directory

So if I'm at /home/frank and wish to specify the path above in a relative fashion I could use:

```
../condron/source/xntp
```

This indicates that I should first go up one directory level, then come down through the **condron** directory, followed by the **source** directory and then to **xntp**.

2.3 Unix Directories, Files and Inodes

Every **directory** and **file** is listed in its parent directory. In the case of the root directory, that parent is itself. A directory is a file that contains a table listing the files contained within it, giving file names to the **inode** numbers in the list. An inode is a special file designed to be read by the kernel to learn the information about each file. It specifies the permissions on the file, ownership, date of creation and of last access and change, and the physical location of the data blocks on the disk containing the file.

The system does not require any particular structure for the data in the file itself. The file can be ASCII or binary or a combination, and may represent text data, a shell script, compiled object code for a program, directory table, junk, or anything you would like.

There's no header, trailer, label information or **EOF** character as part of the file.

2.4 Unix Programs

A **program**, or **command**, interacts with the kernel to provide the environment and perform the functions called for by the user. A program can be: an executable shell file, known as a shell script; a built-in shell command; or a source compiled, object code file.

The **shell** is a command line interpreter. The user interacts with the kernel through the shell. You can write ASCII (text) scripts to be acted upon by a shell.

System programs are usually binary, having been compiled from C source code. These are located in places like **/bin, /usr/bin, /usr/local/bin, /usr/ucb**, etc. They provide the functions that you normally think of when you think of Unix. Some of these are ***sh***, ***csh***, ***date***, ***who***, ***more***, and there are many others.

CHAPTER 3 Getting Started

3.1 Logging in

After connecting with a Unix system, a user is prompted for a **login** username, then a **password**. The login username is the user's unique name on the system. The password is a changeable code known only to the user. At the **login** prompt, the user should enter the username; at the **password** prompt, the current password should be typed.

Note: **Unix is case sensitive.** Therefore, the **login** and **password** should be typed exactly as issued; the login, at least, will normally be in lower case.

3.1.1 Terminal Type

Most systems are set up so the user is by default prompted for a terminal type, which should be set to match the terminal in use before proceeding. Most computers work if you choose "**vt100**". Users connecting using a Sun workstation may want to use "**sun**"; those using an X-Terminal may want to use "**xterms**" or "**xterm**".

The terminal type indicates to the Unix system how to interact with the session just opened.

Should you need to reset the terminal type, enter the command:

setenv TERM *<term type>* - if using the C-shell (see Chapter 4.)

(On some systems, e.g. MAGNUS, it's also necessary to type "***unsetenv TERMCAP***".)

-or-

TERM=*<term type>*; export TERM - if using the Bourne shell (see Chapter 4.)

where *<term type>* is the terminal type, such as **vt100**, that you would like set.

3.1.2 Passwords

When your account is issued, you will be given an initial password. It is important for system and personal security that the password for your account be changed to something of your choosing. The command for changing a password is "***passwd***". You will be asked both for your old password and to type your new selected password twice. If you mistype your old password or do not type your new password the same way twice, the system will indicate that the password has not been changed.

Some system administrators have installed programs that check for appropriateness of password (is it cryptic enough for reasonable system security). A password change may be rejected by this program.

When choosing a password, it is important that it be something that could not be guessed -- either by somebody unknown to you trying to break in, or by an acquaintance who knows you. Suggestions for choosing and using a password follow:

Don't	use a word (or words) in any language
	use a proper name
	use information that can be found in your wallet
	use information commonly known about you (car license, pet name, etc)
	use control characters. Some systems can't handle them
	write your password anywhere
	ever give your password to *anybody*
Do	use a mixture of character types (alphabetic, numeric, special)
	use a mixture of upper case and lower case
	use at least 6 characters
	choose a password you can remember
	change your password often
	make sure nobody is looking over your shoulder when you are entering your password

3.1.3 Exiting

^D - indicates end of data stream; can log a user off. The latter is disabled on many systems

^C - interrupt

logout - leave the system

exit - leave the shell

3.1.4 Identity

The system identifies you by the user and group numbers (**userid** and **groupid**, respectively) assigned to you by your system administrator. You don't normally need to know your userid or groupid as the system translates username ↔ userid, and groupname ↔ groupid automatically. You probably already know your username; it's the name you logon with. The groupname is not as obvious, and indeed, you may belong to more than one group. Your primary group is the one associated with your username in the password database file, as set up by your system administrator. Similarly, there is a group database file where the system administrator can assign you rights to additional groups on the system.

In the examples below % is your shell prompt; you don't type this in.

You can determine your userid and the list of groups you belong to with the ***id*** and ***groups*** commands. On some systems ***id*** displays your user and primary group information, e.g.:

```
% id
    uid=1101(frank) gid=10(staff)
```

on other systems it also displays information for any additional groups you belong to:

```
% id
    uid=1101(frank) gid=10(staff) groups=10(staff),5(operator),14(sysadmin),110(uts)
```

The ***groups*** command displays the group information for all the groups you belong to, e.g.:

```
% groups
    staff sysadmin uts operator
```

3.2 Unix Command Line Structure

A **command** is a program that tells the Unix system to do something. It has the form:

command [***options***] [***arguments***]

where an **argument** indicates on what the command is to perform its action, usually a file or series of files. An option modifies the command, changing the way it performs.

Commands are case sensitive. ***command*** and ***Command*** are not the same.

Options are generally preceded by a hyphen (-), and for most commands, more than one option can be strung together, in the form:

command -[option][option][option]

e.g.:

ls -alR

will perform a long list on all files in the current directory and recursively perform the list through all sub-directories.

For most commands you can separate the options, preceding each with a hyphen, e.g.:

command -option1 -option2 -option3

as in:

ls -a -l -R

Some commands have options that require parameters. Options requiring parameters are usually specified separately, e.g.:

lpr -Pprinter3 -# 2 file

will send 2 copies of file to printer3.

These are the standard conventions for commands. However, not all Unix commands will follow the standard. Some don't require the hyphen before options and some won't let you group options together, i.e. they may require that each option be preceded by a hyphen and separated by whitespace from other options and arguments.

Options and syntax for a command are listed in the **man page** for the command.

3.3 Control Keys

Control keys are used to perform special functions on the command line or within an editor. You type these by holding down the **Control** key and some other **key** simultaneously. This is usually represented as **^Key**. **Control-S** would be written as ^S. With control keys upper and lower case are the same, so **^S** is the same as **^s**. This particular example is a **stop** signal and tells the terminal to stop accepting input. It will remain that way until you type a **start** signal, **^Q**.

Control-U is normally the "**line-kill**" signal for your terminal. When typed it erases the entire input line.

In the ***vi*** editor you can type a control key into your text file by first typing ^V followed by the control character desired, so to type **^H** into a document type ^V^**H**.

3.4 stty - terminal control

stty reports or sets terminal control options. The "**tty**" is an abbreviation that harks back to the days of teletypewriters, which were associated with transmission of telegraph messages, and which were models for early computer terminals.

For new users, the most important use of the ***stty*** command is setting the erase function to the appropriate key on their terminal. For systems programmers or shell script writers, the ***stty*** command provides an invaluable tool for configuring many aspects of I/O control for a given device, including the following:

- erase and line-kill characters
- data transmission speed
- parity checking on data transmission
- hardware flow control
- newline (NL) versus carriage return plus linefeed (CR-LF)

- interpreting tab characters
- edited versus raw input
- mapping of upper case to lower case

This command is very system specific, so consult the **man pages** for the details of the ***stty*** command on your system.

Syntax

stty [options]

Options

(none)	report the terminal settings
all (or -a)	report on all options
echoe	echo ERASE as BS-space-BS
dec	set modes suitable for Digital Equipment Corporation operating systems (which distinguishes between ERASE and BACKSPACE) (Not available on all systems)
kill	set the LINE-KILL character
erase	set the ERASE character
intr	set the INTERRUPT character

Examples

You can display and change your terminal control settings with the ***stty*** command. To display all (**-a**) of the current line settings:

```
% stty -a
    speed 38400 baud, 24 rows, 80 columns
    parenb -parodd cs7 -cstopb -hupcl cread -clocal -crtscts
    -ignbrk brkint ignpar -parmrk -inpck istrip -inlcr -igncr icrnl -iuclc
    ixon -ixany -ixoff imaxbel
    isig iexten icanon -xcase echo echoe echok -echonl -noflsh -tostop
    echoctl -echoprt echoke
    opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel
    erase  kill  werase rprnt  flush  lnext  susp   intr  quit  stop  eof
    ^H     ^U    ^W     ^R     ^O     ^V     ^Z/^Y  ^C    ^\    ^S/^Q ^D
```

You can change settings using *stty*, e.g., to change the erase character from **^?** (the delete key) to **^H**:

```
% stty erase ^H
```

This will set the terminal options for the current session only. To have this done for you automatically each time you login, it can be inserted into the **.login** or **.profile** file that we'll look at later.

3.5 Getting Help

The Unix manual, usually called **man pages**, is available on-line to explain the usage of the Unix system and commands. To use a man page, type the command "*man*" at the system prompt followed by the command for which you need information.

Syntax

man [options] command_name

Common Options

-k keyword	list command synopsis line for all keyword matches
-M path	path to man pages
-a	show all matching man pages (SVR4)

Examples

You can use ***man*** to provide a one line synopsis of any commands that contain the keyword that you want to search on with the "**-k**" option, e.g. to search on the keyword **password**, type:

```
% man -k password
    passwd (5)        - password file
    passwd (1)        - change password information
```

The number in parentheses indicates the section of the man pages where these references were found. You can then access the man page (by default it will give you the lower numbered entry, but you can use a command line option to specify a different one) with:

```
% man passwd
PASSWD(1)           USER COMMANDS           PASSWD(1)
NAME
   passwd    - change password  information
SYNOPSIS
   passwd [ -e login_shell ] [ username ]
DESCRIPTION
   passwd changes (or sets) a user's password.
   passwd prompts twice for the new password, without displaying
   it.  This is to allow for the possibility of typing mistakes.
   Only the user and the super-user can change the user's password.
OPTIONS
   -e  Change the user's login shell.
```

Here we've paraphrased and truncated the output for space and copyright concerns.

3.6 Directory Navigation and Control

The Unix file system is set up like a tree branching out from the root. The the **root** directory of the system is symbolized by the forward slash (/). System and user directories are organized under the **root**. The user does not have a root directory in Unix; users generally log into their own **home** directory. Users can then create other directories under their **home**. The following table summarizes some directory navigation commands.

TABLE 3.1 **Navigation and Directory Control Commands**

Command/Syntax	What it will do
cd [*directory*]	change directory
ls [options] [*directory* or *file*]	list *directory* contents or *file* permissions
mkdir [options] *directory*	make a *directory*
pwd	print working (current) directory
rmdir [options] *directory*	remove a *directory*

If you're familiar with DOS the following table comparing similar commands might help to provide the proper reference frame.

TABLE 3.2 **Unix vs DOS Navigation and Directory Control Commands**

Command	Unix	DOS
list directory contents	ls	dir
make directory	mkdir	md & mkdir
change directory	cd	cd & chdir
delete (remove) directory	rmdir	rd & rmdir
return to user's home directory	cd	cd\
location in path (present working directory)	pwd	cd

3.6.1 pwd - print working directory

At any time you can determine where you are in the file system hierarchy with the *pwd*, print working directory, command, e.g.:

```
% pwd
    /home/frank/src
```

3.6.2 cd - change directory

You can change to a new directory with the *cd*, change directory, command. *cd* will accept both absolute and relative path names.

Syntax

cd [directory]

Examples

cd (also *chdir* in some shells)	change directory
cd	changes to user's home directory
cd /	changes directory to the system's root
cd ..	goes up one directory level
cd ../..	goes up two directory levels
***cd* /full/path/name/from/root**	changes directory to absolute path named (note the leading slash)
***cd* path/from/current/location**	changes directory to path relative to current location (no leading slash)
***cd* ~username/directory**	changes directory to the named username's indicated directory (Note: the ~ is not valid in the Bourne shell; see Chapter 5.)

3.6.3 mkdir - make a directory

You extend your home hierarchy by making sub-directories underneath it. This is done with the ***mkdir***, make directory, command. Again, you specify either the full or relative path of the directory:

Syntax

mkdir [options] directory

Common Options

-p	create the intermediate (parent) directories, as needed
-m mode	access permissions (SVR4). (We'll look at modes later in this Chapter).

Examples

```
% mkdir /home/frank/data
```

or, if your present working directory is /home/frank the following would be equivalent:

```
% mkdir data
```

3.6.4 rmdir - remove directory

A directory needs to be empty before you can remove it. If it's not, you need to remove the files first. Also, you can't remove a directory if it is your present working directory; you must first change out of it.

Syntax

rmdir directory

Examples

To remove the empty directory /home/frank/data while in /home/frank use:

```
% rmdir data
```

or

```
% rmdir /home/frank/data
```

3.6.5 ls - list directory contents

The command to list your directories and files is *ls*. With options it can provide information about the size, type of file, permissions, dates of file creation, change and access.

Syntax

ls [options] [argument]

Common Options

When no argument is used, the listing will be of the current directory. There are many very useful options for the ls command. A listing of many of them follows. When using the command, string the desired options together preceded by "-".

-a	lists all files, including those beginning with a dot (.).
-d	lists only names of directories, not the files in the directory
-F	indicates type of entry with a trailing symbol:
	directories /
	sockets =
	symbolic links @
	executables *
-g	displays Unix group assigned to the file, requires the -l option (BSD only)
	-or- on an SVR4 machine, e.g. Solaris, this option has the opposite effect
-L	if the file is a symbolic link, lists the information for the file or directory the link references, not the information for the link itself
-l	long listing: lists the mode, link information, owner, size, last modification (time). If the file is a symbolic link, an arrow (-->) precedes the pathname of the linked-to file.

The **mode field** is given by the **-l** option and consists of 10 characters. The first character is one of the following:

CHARACTER	IF ENTRY IS A
d	directory
-	plain file
b	block-type special file
c	character-type special file
l	symbolic link
s	socket

The next 9 characters are in 3 sets of 3 characters each. They indicate the **file access permissions**: the first 3 characters refer to the permissions for the **user**, the next three for the users in the Unix **group** assigned to the file, and the last 3 to the permissions for **other** users on the system. Designations are as follows:

r	read permission
w	write permission
x	execute permission
-	no permission

There are a few less commonly used permission designations for special circumstances. These are explained in the man page for ***ls***.

Examples

To list the files in a directory:

```
% ls
    demofiles     frank        linda
```

To list all files in a directory, including the hidden (dot) files try:

```
% ls -a
    .       .cshrc    .history   .plan      .rhosts       frank
    ..      .emacs    .login     .profile   demofiles     linda
```

To get a long listing:

```
% ls -al
    total 24
    drwxr-sr-x   5 workshop acs          512 Jun  7 11:12 .
    drwxr-xr-x   6 root     sys          512 May 29 09:59 ..
    -rwxr-xr-x   1 workshop acs          532 May 20 15:31 .cshrc
    -rw-------   1 workshop acs          525 May 20 21:29 .emacs
    -rw-------   1 workshop acs          622 May 24 12:13 .history
    -rwxr-xr-x   1 workshop acs          238 May 14 09:44 .login
    -rw-r--r--   1 workshop acs          273 May 22 23:53 .plan
    -rwxr-xr-x   1 workshop acs          413 May 14 09:36 .profile
    -rw-------   1 workshop acs           49 May 20 20:23 .rhosts
    drwx------   3 workshop acs          512 May 24 11:18 demofiles
    drwx------   2 workshop acs          512 May 21 10:48 frank
    drwx------   3 workshop acs          512 May 24 10:59 linda
```

3.7 File Maintenance Commands

To create, copy, remove and change permissions on files you can use the following commands.

TABLE 3.3 **File Maintenance Commands**

Command/Syntax	What it will do
chgrp [options] *group file*	change the group of the file
chmod [options] *file*	change file or directory access permissions
chown [options] *owner file*	change the ownership of a file; can only be done by the superuser
cp [options] *file1 file2*	copy *file1* into *file2*; *file2* shouldn't already exist. This command creates or overwrites *file2*.
mv [options] *file1 file2*	move *file1* into *file2*
rm [options] *file*	remove (delete) a file or directory (**-r** recursively deletes the directory and its contents) (**-i** prompts before removing files)

If you're familiar with DOS the following table comparing similar commands might help to provide the proper reference frame.

TABLE 3.4 **Unix vs DOS File Maintenance Commands**

Command	Unix	DOS
copy file	cp	copy
move file	mv	move (not supported on all versions of DOS)
rename file	mv	rename & ren
delete (remove) file	rm	erase & del
display file to screen		
entire file	cat	type
one page at a time	more, less, pg	type/p (not supported on all versions of DOS)

3.7.1 cp - copy a file

Copy the contents of one file to another with the *cp* command.

Syntax

cp [options] old_filename new_filename

Common Options

-i	interactive (prompt and wait for confirmation before proceeding)
-r	recursively copy a directory

Examples

```
% cp old_filename new_filename
```

You now have two copies of the file, each with identical contents. They are completely independent of each other and you can edit and modify either as needed. They each have their own inode, data blocks, and directory table entries.

3.7.2 mv - move a file

Rename a file with the move command, *mv*.

Syntax

mv [options] old_filename new_filename

Common Options

-i	interactive (prompt and wait for confirmation before proceeding)
-f	don't prompt, even when copying over an existing target file (overrides **-i**)

Examples

```
% mv old_filename new_filename
```

You now have a file called **new_filename** and the file **old_filename** is gone. Actually all you've done is to update the directory table entry to give the file a new name. The contents of the file remain where they were.

3.7.3 rm - remove a file

Remove a file with the ***rm***, remove, command.

Syntax

rm [options] filename

Common Options

-i	interactive (prompt and wait for confirmation before proceeding)
-r	recursively remove a directory, first removing the files and subdirectories beneath it
-f	don't prompt for confirmation (overrides -i)

Examples

```
% rm old_filename
```

A listing of the directory will now show that the file no longer exists. Actually, all you've done is to remove the directory table entry and mark the inode as unused. The file contents are still on the disk, but the system now has no way of identifying those data blocks with a file name. There is no command to "**unremove**" a file that has been removed in this way. For this reason many novice users alias their remove command to be "***rm -i***", where the **-i** option prompts them to answer yes or no before the file is removed. Such aliases are normally placed in the **.cshrc** file for the C shell; see Chapter 5)

3.7.4 File Permissions

Each file, directory, and executable has permissions set for who can **read**, write, and/or **execute** it. To find the permissions assigned to a file, the ***ls*** command with the **-l** option should be used. Also, using the **-g** option with "***ls -l***" will help when it is necessary to know the group for which the permissions are set (BSD only).

When using the "***ls -lg***" command on a file (***ls -l*** on SysV), the output will appear as follows:

```
-rwxr-x---    user  unixgroup       size Month nn hh:mm filename
```

The area above designated by letters and dashes (**-rwxr-x---**) is the area showing the file type and permissions as defined in the previous Section. Therefore, a permission string, for example, of **-rwxr-x---** allows the **user** (owner) of the file to read, write, and execute it; those in the **unixgroup** of the file can read and execute it; **others** cannot access it at all.

3.7.5 chmod - change file permissions

The command to change permissions on an item (file, directory, etc) is ***chmod*** (change mode). The syntax involves using the command with three digits (representing the **user** (owner, **u**) permissions, the **group** (**g**) permissions, and **other** (**o**) user's permissions) followed by the argument (which may be a file name or list of files and directories). Or by using symbolic representation for the permissions and who they apply to.

Each of the permission types is represented by either a numeric equivalent:

read=4, write=2, execute=1

or a single letter:

read=r, write=w, execute=x

A permission of **4** or **r** would specify **read** permissions. If the permissions desired are read and write, the 4 (representing read) and the 2 (representing write) are added together to make a permission of 6. Therefore, a permission setting of 6 would allow read and write permissions.

Alternatively, you could use symbolic notation which uses the one letter representation for who and for the permissions and an operator, where the operator can be:

+	add permissions
-	remove permissions
=	set permissions

So to set read and write for the owner we could use "**u=rw**" in symbolic notation.

Syntax

chmod nnn [argument list]	numeric mode
chmod [who]op[perm] [argument list]	symbolic mode

where **nnn** are the three numbers representing **user, group,** and **other** permissions, **who** is any of **u, g, o**, or **a** (all) and **perm** is any of **r**, **w**, **x**. In symbolic notation you can separate permission specifications by commas, as shown in the example below.

Common Options

-f	force (no error message is generated if the change is unsuccessful)
-R	recursively descend through the directory structure and change the modes

Examples

If the permission desired for file1 is **user**: read, write, execute, **group**: read, execute, **other**: read, execute, the command to use would be

chmod 755 file1 or ***chmod u=rwx,go=rx file1***

Reminder: When giving permissions to **group** and **other** to use a file, it is necessary to allow at least execute permission to the directories for the path in which the file is located. The easiest way to do this is to be in the directory for which permissions need to be granted:

chmod 711 . or *chmod u=rw,+x .* or *chmod u=rwx,go=x .*

where the dot (.) indicates **this directory**.

3.7.6 chown - change ownership

Ownership of a file can be changed with the ***chown*** command. On most versions of Unix this can only be done by the super-user, i.e. a normal user can't give away ownership of their files. ***chown*** is used as below, where # represents the shell prompt for the super-user:

Syntax

chown [options] user[:group] file	(SVR4)
chown [options] user[.group] file	(BSD)

Common Options

-R	recursively descend through the directory structure
-f	force, and don't report any errors

Examples

```
# chown new_owner file
```

3.7.7 chgrp - change group

Anyone can change the group of files they own, to another group they belong to, with the ***chgrp*** command.

Syntax

chgrp [options] group file

Common Options

-R	recursively descend through the directory structure
-f	force, and don't report any errors

Examples

```
% chgrp new_group file
```

3.8 Display Commands

There are a number of commands you can use to **display** or **view** a file. Some of these are editors which we will look at later. Here we will illustrate some of the commands normally used to display a file.

TABLE 3.5 Display Commands

Command/Syntax	What it will do
cat [options] *file*	concatenate (list) a file
echo [text string]	echo the text string to stdout
head [-number] *file*	display the first 10 (or number of) lines of a file
more (or ***less*** or *pg*) [options] *file*	page through a text file
tail [options] *file*	display the last few lines (or parts) of a file

3.8.1 echo - echo a statement

The ***echo*** command is used to repeat, or echo, the argument you give it back to the standard output device. It normally ends with a line-feed, but you can specify an option to prevent this.

Syntax

echo [string]

Common Options

-n	don't print <new-line> (BSD, shell built-in)
\c	don't print <new-line> (SVR4)
\0n	where **n** is the 8-bit ASCII character code (SVR4)
\t	tab (SVR4)
\f	form-feed (SVR4)
\n	new-line (SVR4)
\v	vertical tab (SVR4)

Examples

```
% echo Hello Class          or          echo "Hello Class"
```

To prevent the line feed:

```
% echo -n Hello Class       or          echo "Hello Class \c"
```

where the style to use in the last example depends on the ***echo*** command in use.

The \x options must be within pairs of single or double quotes, with or without other string characters.

3.8.2 cat - concatenate a file

Display the contents of a file with the concatenate command, *cat*.

Syntax

cat [options] [file]

Common Options

-n	precede each line with a line number
-v	display non-printing characters, except tabs, new-lines, and form-feeds
-e	display $ at the end of each line (prior to new-line) (when used with **-v** option)

Examples

```
% cat filename
```

You can list a series of files on the command line, and *cat* will concatenate them, starting each in turn, immediately after completing the previous one, e.g.:

```
% cat file1 file2 file3
```

3.8.3 more, less, and pg - page through a file

more, ***less***, and ***pg*** let you page through the contents of a file one screenful at a time. These may not all be available on your Unix system. They allow you to back up through the previous pages and search for words, etc.

Syntax

more [options] [+/pattern] [filename]
less [options] [+/pattern] [filename]
pg [options] [+/pattern] [filename]

Options

more	**less**	**pg**	**Action**
-c	**-c**	**-c**	clear display before displaying
	-i		ignore case
-w	default	default	don't exit at end of input, but prompt and wait
-lines		**-lines**	# of lines/screenful
+/pattern	**+/pattern**	**+/pattern**	search for the pattern

Internal Controls

more	displays (one screen at a time) the file requested
<space bar>	to view next screen
<return> or *<CR>*	to view one more line
q	to quit viewing the file
h	help
b	go back up one screenful
/word	search for **word** in the remainder of the file See the **man page** for additional options
less	similar to ***more***; see the **man page** for options
pg	the SVR4 equivalent of ***more*** (page)

3.8.4 head - display the start of a file

head displays the head, or start, of the file.

Syntax

head [options] file

Common Options

-n number	number of lines to display, counting from the top of the file
-number	same as above

Examples

By default *head* displays the first 10 lines. You can display more with the "**-n number**", or "**-number**" options, e.g., to display the first 40 lines:

```
% head -40 filename          or          head -n 40 filename
```

3.8.5 tail - display the end of a file

tail displays the tail, or end, of the file.

Syntax

tail [options] file

Common Options

-number	number of lines to display, counting from the bottom of the file

Examples

The default is to display the last 10 lines, but you can specify different line or byte numbers, or a different starting point within the file. To display the last 30 lines of a file use the **-number** style:

```
% tail -30 filename
```

CHAPTER 4 System Resources & Printing

4.1 System Resources

Commands to report or manage system resources.

TABLE 4.1 **System Resource Commands**

Command/Syntax	What it will do
chsh *(passwd -e/-s) username login_shell*	change the user's login shell (often only by the superuser)
date [options]	report the current date and time
df [options] [resource]	report the summary of disk blocks and inodes free and in use
du [options] [*directory* or *file*]	report amount of disk space in use+
hostname/uname	display or set (super-user only) the name of the current machine
kill [options] [-SIGNAL] [pid#] [%job]	send a signal to the process with the process id number (pid#) or job control number (%n). The default signal is to kill the process.
man [options] *command*	show the manual (**man**) page for a command
passwd [options]	set or change your password
ps [options]	show status of active processes
script file	saves everything that appears on the screen to file until *exit* is executed
stty [options]	set or display terminal control options
whereis [options] *command*	report the binary, source, and man page locations for the command named
which *command*	reports the path to the command or the shell alias in use
who or ***w***	report who is logged in and what processes are running

4.1.1 df - summarize disk block and file usage

df is used to report the number of disk blocks and inodes used and free for each file system. The output format and valid options are very specific to the OS and program version in use.

Syntax

df [options] [resource]

Common Options

-l	local file systems only (SVR4)
-k	report in kilobytes (SVR4)

Examples

```
{unix prompt 1} df
   Filesystem            kbytes    used    avail capacity  Mounted on
   /dev/sd0a              20895   19224        0   102%    /
   /dev/sd0h             319055  131293   155857    46%    /usr
   /dev/sd1g             637726  348809   225145    61%    /usr/local
   /dev/sd1a             240111  165489    50611    77%
   /home/guardian
   peri:/usr/local/backup
                        1952573  976558   780758    56%
   /usr/local/backup
   peri:/home/peri       726884  391189   263007    60%    /home/peri
   peri:/usr/spool/mail  192383    1081   172064     1%
   /var/spool/mail
   peri:/acs/peri/2      723934  521604   129937    80%    /acs/peri/2
```

4.1.2 du - report disk space in use

du reports the amount of disk space in use for the files or directories you specify.

Syntax

du [options] [directory or file]

Common Options

-a	display disk usage for each file, not just subdirectories
-s	display a summary total only
-k	report in kilobytes (SVR4)

Examples

```
{unix prompt 3} du
    1     ./.elm
    1     ./Mail
    1     ./News
    20    ./uc
    86    .

{unix prompt 4} du -a uc
    7     uc/unixgrep.txt
    5     uc/editors.txt
    1     uc/.emacs
    1     uc/.exrc
    4     uc/telnet.ftp
    1     uc/uniq.tee.txt
    20    uc
```

4.1.3 ps - show status of active processes

ps is used to report on processes currently running on the system. The output format and valid options are very specific to the OS and program version in use.

Syntax

ps [options]

Common Options

BSD	**SVR4**	
-a	**-e**	all processes, all users
-e		environment/everything
-g		process group leaders as well
-l	**-l**	long format
-u	**-u user**	user oriented report
-x	**-e**	even processes not executed from terminals
	-f	full listing
-w		report first 132 characters per line

note -- Because the *ps* command is highly system-specific, it is recommended that you consult the **man pages** of your system for details of options and interpretation of *ps* output.

Examples

```
{unix prompt 5} ps
      PID TT STAT  TIME COMMAND
    15549 p0 IW    0:00 -tcsh (tcsh)
    15588 p0 IW    0:00 man nice
    15594 p0 IW    0:00 sh -c less /tmp/man15588
    15595 p0 IW    0:00 less /tmp/man15588
    15486 p1 S     0:00 -tcsh (tcsh)
    15599 p1 T     0:00 emacs unixgrep.txt
    15600 p1 R     0:00 ps
```

4.1.4 kill - terminate a process

kill sends a signal to a process, usually to terminate it.

Syntax

kill [-signal] process-id

Common Options

-l	displays the available kill signals:

Examples

```
{unix prompt 9} kill -l
    HUP INT QUIT ILL TRAP IOT EMT FPE KILL BUS SEGV SYS PIPE ALRM TERM URG STOP
    TSTP CONT CHLD TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH LOST USR1 USR2
```

The **-KILL** signal, also specified as **-9** (because it is 9th on the above list), is the most commonly used *kill* signal. Once seen, it can't be ignored by the program whereas the other signals can.

```
{unix prompt 10} kill -9 15599
    [1] + Killed          emacs unixgrep.txt
```

4.1.5 who - list current users

who reports who is logged in at the present time.

Syntax

who [am i]

Examples

```
beauty condron>who
   wmtell   ttyp1  Apr 21 20:15   (apple.acs.ohio-s)
   fbwalk   ttyp2  Apr 21 23:21   (worf.acs.ohio-st)
   stwang   ttyp3  Apr 21 23:22   (127.99.25.8)
   david    ttyp4  Apr 21 22:27   (slip1-61.acs.ohi)
   tgardner ttyp5  Apr 21 23:07   (picard.acs.ohio-)
   awallace ttyp6  Apr 21 23:00   (ts31-4.homenet.o)
   gtl27    ttyp7  Apr 21 23:24   (data.acs.ohio-st)
   ccchang  ttyp8  Apr 21 23:32   (slip3-10.acs.ohi)
   condron  ttypc  Apr 21 23:38   (lcondron-mac.acs)
   dgildman ttype  Apr 21 22:30   (slip3-36.acs.ohi)
   fcbetz   ttyq2  Apr 21 21:12   (ts24-10.homenet.)

beauty condron>who am i
   beauty!condron  ttypc  Apr 21 23:38   (lcondron-mac.acs)
```

4.1.6 whereis - report program locations

whereis reports the filenames of source, binary, and manual page files associated with command(s).

Syntax

whereis [options] command(s)

Common Options

-b	report binary files only
-m	report manual sections only
-s	report source files only

Examples

```
brigadier: condron [69]> whereis Mail
   Mail: /usr/ucb/Mail /usr/lib/Mail.help /usr/lib/Mail.rc /usr/man/man1/Mail.1
```

```
brigadier: condron [70]> whereis -b Mail
    Mail: /usr/ucb/Mail /usr/lib/Mail.help /usr/lib/Mail.rc

brigadier: condron [71]> whereis -m Mail
    Mail: /usr/man/man1/Mail.1
```

4.1.7 which - report the command found

which will report the name of the file that is be executed when the command is invoked. This will be the full path name or the alias that's found first in your path.

Syntax

which command(s)

example--

```
brigadier: condron [73]> which Mail
    /usr/ucb/Mail
```

4.1.8 hostname/uname - name of machine

hostname (***uname -n*** on SysV) reports the host name of the machine the user is logged into, e.g.:

```
brigadier: condron [91]> hostname
    brigadier
```

uname has additional options to print information about system hardware type and software version.

4.1.9 script - record your screen I/O

script creates a script of your session input and output. Using the ***script*** command, you can capture all the data transmission from and to your terminal screen until you ***exit*** the script program. This can be useful during the programming-and-debugging process, to document the combination of things you have tried, or to get a printed copy of it all for later perusal.

Syntax

script [-a] [file] <. . .> exit

Common Options

-a	append the output to file

typescript is the name of the default file used by ***script***.

You must remember to type ***exit*** to end your script session and close your typescript file.

Examples

```
beauty condron>script
   Script started, file is typescript
beauty condron>ps
     PID TT STAT   TIME COMMAND
   23323 p8 S      0:00 -h -i (tcsh)
   23327 p8 R      0:00 ps
   18706 pa S      0:00 -tcsh (tcsh)
   23315 pa T      0:00 emacs
   23321 pa S      0:00 script
   23322 pa S      0:00 script
    3400 pb I      0:00 -tcsh (tcsh)
beauty condron>kill -9 23315
beauty condron>date
   Mon Apr 22 22:29:44 EDT 1996
beauty condron>exit
   exit
   Script done, file is typescript
   [1]  + Killed                  emacs

beauty condron>cat typescript
   Script started on Mon Apr 22 22:28:36 1996
   beauty condron>ps
     PID TT STAT   TIME COMMAND
   23323 p8 S      0:00 -h -i (tcsh)
   23327 p8 R      0:00 ps
   18706 pa S      0:00 -tcsh (tcsh)
   23315 pa T      0:00 emacs
   23321 pa S      0:00 script
   23322 pa S      0:00 script
    3400 pb I      0:00 -tcsh (tcsh)
   beauty condron>kill -9 23315
   beauty condron>date
   Mon Apr 22 22:29:44 EDT 1996
   beauty condron>exit
   exit

   script done on Mon Apr 22 22:30:02 1996
   beauty condron>
```

4.1.10 date - current date and time

date displays the current data and time. A superuser can set the date and time.

Syntax

date [options] [+format]

Common Options

-u	use Universal Time (or Greenwich Mean Time)
+format	specify the output format
%a	weekday abbreviation, Sun to Sat
%h	month abbreviation, Jan to Dec
%j	day of year, 001 to 366
%n	<new-line>
%t	<TAB>
%y	last 2 digits of year, 00 to 99
%D	MM/DD/YY date
%H	hour, 00 to 23
%M	minute, 00 to 59
%S	second, 00 to 59
%T	HH:MM:SS time

Examples

```
beauty condron>date
    Mon Jun 10 09:01:05 EDT 1996


beauty condron>date -u
    Mon Jun 10 13:01:33 GMT 1996


beauty condron>date +%a%t%D
    Mon                 06/10/96


beauty condron>date '+%y:%j'
    96:162
```

4.2 Print Commands

TABLE 4.2 **Printing Commands**

Command/Syntax	What it will do
lpq (*lpstat*) [options]	show the status of print jobs
lpr (*lp*) [options] *file*	print to defined printer
lprm (*cancel*) [options]	remove a print job from the print queue
pr [options] *[file]*	filter the file and print it on the terminal

The print commands allow us to print files to standard output (*pr*) or to a line printer (*lp/lpr*) while filtering the output. The **BSD** and **SysV** printer commands use different names and different options to produce the same results: ***lpr***, ***lprm***, and ***lpq*** vs ***lp***, ***cancel***, and ***lpstat*** for the BSD and SysV submit, cancel, and check the status of a print job, respectively.

4.2.1 lp/lpr - submit a print job

lp and ***lpr*** submit the specified file, or standard input, to the printer daemon to be printed. Each job is given a unique request-id that can be used to follow or cancel the job while it's in the queue.

Syntax

lp [options] filename

lpr [options] filename

Common Options

lp	**lpr**	**function**
-n number	**-#**number	number of copies
-t title	**-T**title	title for job
-d destination	**-P**printer	printer name
-c	(default)	copy file to queue before printing
(default)	**-s**	don't copy file to queue before printing
-o option		additional options, e.g. nobanner

Files beginning with the string "%!" are assumed to contain PostScript commands.

Examples

To print the file ssh.ps:

```
% lp ssh.ps
request id is lp-153 (1 file(s))
```

This submits the job to the queue for the default printer, **lp**, with the request-id lp-153.

4.2.2 lpstat/lpq - check the status of a print job

You can check the status of your print job with lpstat or lpq.

Syntax

lpstat [options]

lpq [options] [job#] [username]

Common Options

lpstat	**lpq**	**function**
-d	(defaults to lp)	list system default destination
-s		summarize print status
-t		print all status information
-u [login-ID-list]		user list
-v		list printers known to the system
-p printer_dest	**-P**printer_dest	list status of printer, printer_dest

Examples

```
% lpstat
    lp-153        frank        208068    Apr 29 15:14 on lp
```

4.2.3 cancel/lprm - cancel a print job

Any user can cancel only heir own print jobs.

Syntax

cancel [request-ID] [printer]

lprm [options] [job#] [username]

Common Options

cancel	**lprm**	**function**
	-Pprinter	specify printer
	-	all jobs for user
-u [login-ID-list]		user list

Examples

To cancel the job submitted above:

```
% cancel lp-153
```

4.2.4 pr - prepare files for printing

pr prints header and trailer information surrounding the formatted file. You can specify the number of pages, lines per page, columns, line spacing, page width, etc. to print, along with header and trailer information and how to treat **<tab>** characters.

Syntax

pr [options] file

Common Options

+page_number	start printing with page page_number of the formatted input file
-column	number of columns
-a	modify -column option to fill columns in round-robin order
-d	double spacing
-e[char][gap]	tab spacing
-h header_string	header for each page
-l lines	lines per page
-t	don't print the header and trailer on each page
-w width	width of page

Examples

The file containing the list of P. G. Wodehouse's Lord Emsworth books could be printed, at 14 lines per page (including 5 header and 5 (empty) trailer lines) below, where the **-e** option specifies the **<tab>** conversion style:

```
% pr -l 14 -e42 wodehouse


Apr 29 11:11 1996  wodehouse_emsworth_books Page 1


Something Fresh [1915]                    Uncle Dynamite [1948]
Leave it to Psmith [1923]                 Pigs Have Wings [1952]
Summer Lightning [1929]                   Cocktail Time [1958]
Heavy Weather [1933]                      Service with a Smile [1961]
```

```
Apr 29 11:11 1996  wodehouse_emsworth_books Page 2


Blandings Castle and Elsewhere [1935]     Galahad at Blandings [1965]
Uncle Fred in the Springtime [1939]       A Pelican at Blandings [1969]
Full Moon [1947]                          Sunset at Blandings [1977]
```

CHAPTER 5 Shells

The shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to ***command.com*** in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files.

The original shell was the Bourne shell, ***sh***. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available. It has very good features for controlling input and output, but is not well suited for the interactive user. To meet the latter need the C shell, ***csh***, was written and is now found on most, but not all, Unix systems. It uses C type syntax, the language Unix is written in, but has a more awkward input/output implementation. It has job control, so that you can reattach a job running in the background to the foreground. It also provides a history feature which allows you to modify and repeat previously executed commands.

The default prompt for the Bourne shell is $ (or #, for the root user). The default prompt for the C shell is %.

Numerous other shells are available from the network. Almost all of them are based on either ***sh*** or ***csh*** with extensions to provide job control to ***sh***, allow in-line editing of commands, page through previously executed commands, provide command name completion and custom prompt, etc. Some of the more well known of these may be on your favorite Unix system: the Korn shell, ***ksh***, by David Korn and the Bourne Again SHell, ***bash***, from the Free Software Foundations GNU project, both based on ***sh***, the T-C shell, ***tcsh***, and the extended C shell, ***cshe***, both based on ***csh***. Below we will describe some of the features of ***sh*** and ***csh*** so that you can get started.

5.1 Built-in Commands

The shells have a number of **built-in**, or native commands. These commands are executed directly in the shell and don't have to call another program to be run. These built-in commands are different for the different shells.

5.1.1 Sh

For the Bourne shell some of the more commonly used built-in commands are:

:	null command
.	source (read and execute) commands from a file
case	case conditional loop
cd	change the working directory (default is $HOME)
echo	write a string to standard output
eval	evaluate the given arguments and feed the result back to the shell
exec	execute the given command, replacing the current shell
exit	exit the current shell
export	share the specified environment variable with subsequent shells
for	for conditional loop
if	if conditional loop
pwd	print the current working directory
read	read a line of input from stdin
set	set variables for the shell
test	evaluate an expression as true or false
trap	trap for a typed signal and execute commands
umask	set a default file permission mask for new files
unset	unset shell variables
wait	wait for a specified process to terminate
while	while conditional loop

5.1.2 Csh

For the C shell the more commonly used built-in functions are:

alias	assign a name to a function
bg	put a job into the background
cd	change the current working directory
echo	write a string to stdout
eval	evaluate the given arguments and feed the result back to the shell
exec	execute the given command, replacing the current shell
exit	exit the current shell
fg	bring a job to the foreground
foreach	for conditional loop
glob	do filename expansion on the list, but no "\" escapes are honored
history	print the command history of the shell
if	if conditional loop
jobs	list or control active jobs
kill	kill the specified process
limit	set limits on system resources
logout	terminate the login shell
nice ***command***	lower the scheduling priority of the process, ***command***
nohup ***command***	do not terminate ***command*** when the shell exits
popd	pop the directory stack and return to that directory
pushd	change to the new directory specified and add the current one to the directory stack
rehash	recreate the hash table of paths to executable files
repeat	repeat a command the specified number of times
set	set a shell variable
setenv	set an environment variable for this and subsequent shells
source	source (read and execute) commands from a file
stop	stop the specified background job
switch	switch conditional loop
umask	set a default file permission mask for new files
unalias	remove the specified alias name
unset	unset shell variables
unsetenv	unset shell environment variables
wait	wait for all background processes to terminate
while	while conditional loop

5.2 Environment Variables

Environmental variables are used to provide information to the programs you use. You can have both **global environment** and **local shell variables.** Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell. Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

The current environment variables are displayed with the "***env***" or "***printenv***" commands. Some common ones are:

• **DISPLAY**	The graphical display to use, e.g. nyssa:0.0
• **EDITOR**	The path to your default editor, e.g. /usr/bin/vi
• **GROUP**	Your login group, e.g. staff
• **HOME**	Path to your home directory, e.g. /home/frank
• **HOST**	The hostname of your system, e.g. nyssa
• **IFS**	Internal field separators, usually any white space (defaults to tab, space and <newline>)
• **LOGNAME**	The name you login with, e.g. frank
• **PATH**	Paths to be searched for commands, e.g. /usr/bin:/usr/ucb:/usr/local/bin
• **PS1**	The primary prompt string, Bourne shell only (defaults to $)
• **PS2**	The secondary prompt string, Bourne shell only (defaults to >)
• **SHELL**	The login shell you're using, e.g. /usr/bin/csh
• **TERM**	Your terminal type, e.g. xterm
• **USER**	Your username, e.g. frank

Many environment variables will be set automatically when you login. You can modify them or define others with entries in your startup files or at anytime within the shell. Some variables you might want to change are **PATH** and **DISPLAY**. The **PATH** variable specifies the directories to be automatically searched for the command you specify. Examples of this are in the shell startup scripts below.

You set a **global environment variable** with a command similar to the following for the C shell:

```
% setenv NAME value
```

and for Bourne shell:

```
$ NAME=value; export NAME
```

You can list your global environmental variables with the ***env*** or ***printenv*** commands. You unset them with the ***unsetenv*** (C shell) or ***unset*** (Bourne shell) commands.

To set a **local shell variable** use the *set* command with the syntax below for C shell. Without options ***set*** displays all the local variables.

```
% set name=value
```

For the Bourne shell set the variable with the syntax:

```
$ name=value
```

The current value of the variable is accessed via the "**$name**", or "**${name}**", notation.

5.3 The Bourne Shell, sh

Sh uses the startup file **.profile** in your home directory. There may also be a system-wide startup file, e.g. **/etc/profile**. If so, the system-wide one will be sourced (executed) before your local one.

A simple **.profile** could be the following:

```
PATH=/usr/bin:/usr/ucb:/usr/local/bin:.          # set the PATH
export PATH                                      # so that PATH is available to subshells
# Set a prompt
PS1="{`hostname` `whoami`} "                     # set the prompt, default is "$"
# functions
ls() { /bin/ls -sbF "$@";}
ll() { ls -al "$@";}
# Set the terminal type
stty erase ^H                                    # set Control-H to be the erase key
eval `tset -Q -s -m ':?xterm'`                   # prompt for the terminal type, assume xterm
#
umask 077
```

Whenever a **#** symbol is encountered the remainder of that line is treated as a comment. In the **PATH** variable each directory is separated by a colon (**:**) and the dot (**.**) specifies that the current directory is in your path. If the latter is not set it's a simple matter to execute a program in the current directory by typing:

./program_name

It's actually a good idea not to have dot (**.**) in your path, as you may inadvertently execute a program you didn't intend to when you ***cd*** to different directories.

A variable set in **.profile** is set only in the login shell unless you "***export***" it or **source .profile** from another shell. In the above example **PATH** is exported to any subshells. You can source a file with the built-in "**.**" command of ***sh***, i.e.:

. ***./.profile***

You can make your own functions. In the above example the function ***ll*** results in an "l*s* ***-al***" being done on the specified files or directories.

With ***stty*** the erase character is set to Control-H (^H), which is usually the Backspace key.

The ***tset*** command prompts for the terminal type, and assumes "**xterm**" if we just hit <CR>. This command is run with the shell built-in, ***eval***, which takes the result from the tset command and uses it as an argument for the shell. In this case the "**-s**" option to tset sets the **TERM** and **TERMCAP** variables and exports them.

The last line in the example runs the ***umask*** command with the option such that any files or directories you create will not have read/write/execute permission for **group** and **other**.

For further information about ***sh*** type "***man sh***" at the shell prompt.

5.4 The C Shell, csh

Csh uses the startup files **.cshrc** and **.login**. Some versions use a system-wide startup file, e.g. **/etc/csh.login**. Your **.login** file is sourced (executed) only when you login. Your **.cshrc** file is sourced every time you start a ***csh***, including when you login. It has many similar features to **.profile**, but a different style of doing things. Here we use the ***set*** or ***setenv*** commands to initialize a variable, where ***set*** is used for this shell and *setenv* for this and any subshells. The environment variables: **USER**, **TERM**, and **PATH**, are automatically imported to and exported from the **user**, **term**, and **path** variables of the ***csh***. So ***setenv*** doesn't need to be done for these. The C shell uses the symbol, ~, to indicate the user's home directory in a path, as in **~/.cshrc**, or to specify another user's login directory, as in **~username/.cshrc**.

Predefined variables used by the C shell include:

• **argv**	The list of arguments of the current shell
• **cwd**	The current working directory
• **history**	Sets the size of the history list to save
• **home**	The home directory of the user; starts with $HOME
• **ignoreeof**	When set ignore EOF (^D) from terminals
• **noclobber**	When set prevent output redirection from overwriting existing files
• **noglob**	When set prevent filename expansion with wildcard pattern matching
• **path**	The command search path; starts with $PATH
• **prompt**	Set the command line prompt (default is %)
• **savehist**	number of lines to save in the history list to save in the .history file
• **shell**	The full pathname of the current shell; starts with $SHELL
• **status**	The exit status of the last command (0=normal exit, 1=failed command)
• **term**	Your terminal type, starts with $TERM
• **user**	Your username, starts with $USER

A simple **.cshrc** could be:

```
set path=(/usr/bin /usr/ucb /usr/local/bin ~/bin . )    # set the path
set prompt = "{'hostname' 'whoami' !} "                 # set the primary prompt; default is "%"
set noclobber                                           # don't redirect output to existing files
set ignoreeof                                           # ignore EOF (^D) for this shell
set history=100 savehist=50                             # keep a history list and save it between logins
# aliases
alias h history                                         # alias h to "history"
alias ls "/usr/bin/ls -sbF"                             # alias ls to "ls -sbF"
alias ll ls -al                 # alias ll to "ls -sbFal" (combining these options with those for "ls" above)
alias cd  'cd  \!*;pwd'         # alias cd so that it prints the current working directory after the change
umask 077
```

Some new features here that we didn't see in **.profile** are **noclobber**, **ignoreeof**, and **history**. **Noclobber** indicates that output will not be redirected to existing files, while **ignoreeof** specifies that **EOF** (^D) will not cause the login shell to exit and log you off the system.

With the **history** feature you can recall previously executed commands and re-execute them, with changes if desired.

An **alias** allows you to use the specified ***alias*** name instead of the full command. In the "***ls***" example above, typing "***ls***" will result in "***/usr/bin/ls -sbF***" being executed. You can tell which "***ls***" command is in your path with the built-in ***which*** command, i.e.:

```
which ls
ls:         aliased to /usr/bin/ls -sbF
```

A simple **.login** could be:

```
# .login
stty erase ^H                    # set Control-H to be the erase key
set noglob                       # prevent wild card pattern matching
eval 'tset -Q -s -m ':?xterm''   # prompt for the terminal type, assume "xterm"
unset noglob                     # re-enable wild card pattern matching
```

Setting and unsetting **noglob** around ***tset*** prevents it from being confused by any *csh* filename wild card pattern matching or expansion.

Should you make any changes to your startup files you can initiate the change by sourcing the changed file. For ***csh*** you do this with the built-in ***source*** command, i.e.:

```
source .cshrc
```

For further information about csh type "***man csh***" at the shell prompt.

5.5 Job Control

With the C shell, ***csh***, and many newer shells including some newer Bourne shells, you can put jobs into the background at anytime by appending "**&**" to the command, as with *sh*. After submitting a command you can also do this by typing **^Z** (Control-Z) to suspend the job and then "***bg***" to put it into the background. To bring it back to the foreground type "***fg***".

You can have many jobs running in the background. When they are in the background they are no longer connected to the keyboard for input, but they may still display output to the terminal, interspersing with whatever else is typed or displayed by your current job. You may want to redirect I/O to or from files for the job you intend to background. Your keyboard is connected only to the current, foreground, job.

The built-in ***jobs*** command allows you to list your background jobs. You can use the ***kill*** command to kill a background job. With the **%n** notation you can reference the **n**th background job with either of these commands, replacing **n** with the job number from the output of ***jobs***. So kill the second background job with "***kill %2***" and bring the third job to the foreground with "***fg %3***".

5.6 History

The C shell, the Korn shell and some other more advanced shells, retain information about the former commands you've executed in the shell. How history is done will depend on the shell used. Here we'll describe the C shell history features.

You can use the **history** and **savehist** variables to set the number of previously executed commands to keep track of in this shell and how many to retain between logins, respectively. You could put a line such as the following in **.cshrc** to save the last 100 commands in this shell and the last 50 through the next login.

```
set history=100 savehist=50
```

The shell keeps track of the history list and saves it in **~/.history** between logins.

You can use the built-in ***history*** command to recall previous commands, e.g. to print the last 10:

```
% history 10
52  cd workshop
53  ls
54  cd unix_intro
55  ls
56  pwd
57  date
58  w
59  alias
60  history
61  history 10
```

You can repeat the last command by typing **!!**:

```
% !!
53  ls
54  cd unix_intro
55  ls
56  pwd
57  date
58  w
59  alias
60  history
61  history 10
62  history 10
```

You can repeat any numbered command by prefacing the number with a **!**, e.g.:

```
% !57
date
Tue Apr 9 09:55:31 EDT 1996
```

Or repeat a command starting with any string by prefacing the starting unique part of the string with a **!**, e.g.:

```
% !da
date
Tue Apr 9 09:55:31 EDT 1996
```

When the shell evaluates the command line it first checks for history substitution before it interprets anything else. Should you want to use one of these special characters in a shell command you will need to escape, or quote it first, with a \ before the character, i.e. \!. The history substitution characters are summarized in the following table.

TABLE 5.1 C Shell History Substitution

Command	Substitution Function
!!	repeat last command
!n	repeat command number **n**
!-n	repeat command n from last
!str	repeat command that started with string **str**
!?str?	repeat command with **str** anywhere on the line
!?str?%	select the first argument that had **str** in it
!:	repeat the last command, generally used with a modifier
!:n	select the **n**th argument from the last command (n=0 is the command name)
!:n-m	select the **n**th through **m**th arguments from the last command
!^	select the first argument from the last command (same as !:1)
!$	select the last argument from the last command
!*	select all arguments to the previous command
!:n*	select the **n**th through last arguments from the previous command
!:n-	select the **n**th through next to last arguments from the previous command
^str1^str2^	replace **str1** with **str2** in its first occurrence in the previous command
!n:s/str1/str2/	substitute **str1** with **str2** in its first occurrence in the **n**th command, ending with a **g** substitute globally

Additional editing modifiers are described in the **man page**.

5.7 Changing your Shell

To change your shell you can usually use the "***chsh***" or "***passwd -e***" commands. The option flag, here **-e**, may vary from system to system (-s on BSD based systems), so check the **man page** on your system for proper usage. Sometimes this feature is disabled. If you can't change your shell check with your System Administrator.

The new shell must be the full path name for a valid shell on the system. Which shells are available to you will vary from system to system. The full path name of a shell may also vary. Normally, though, the Bourne and C shells are standard, and available as:

```
/bin/sh
/bin/csh
```

Some systems will also have the Korn shell standard, normally as:

```
/bin/ksh
```

Some shells that are quite popular, but not normally distributed by the OS vendors are bash and tcsh. These might be placed in /bin or a locally defined directory, e.g. /usr/local/bin or /opt/local/bin. Should you choose a shell not standard to the OS make sure that this shell, and all login shells available on the system, are listed in the file **/etc/shells**. If this file exists and your shell is not listed in this file the file transfer protocol daemon, *ftpd*, will not let you connect to this machine. If this file does not exist only accounts with "standard" shells are allowed to connect via **ftp**.

You can always try out a shell before you set it as your default shell. To do this just type in the shell name as you would any other command.

CHAPTER 6 Special Unix Features

One of the most important contributions Unix has made to Operating Systems is the provision of many utilities for doing common tasks or obtaining desired information. Another is the standard way in which data is stored and transmitted in Unix systems. This allows data to be transmitted **to** a file, the terminal screen, or a program, or **from** a file, the keyboard, or a program; always in a uniform manner. The standardized handling of data supports two important features of Unix utilities: I/O redirection and piping.

With **output redirection**, the output of a command is redirected to a file rather than to the terminal screen. With **input redirection**, the input to a command is given via a file rather than the keyboard. Other tricks are possible with input and output redirection as well, as you will see. With **piping**, the output of a command can be used as input (piped) to a subsequent command. In this chapter we discuss many of the features and utilities available to Unix users.

6.1 File Descriptors

There are 3 standard file descriptors:

• **stdin**	0	Standard input to the program
• **stdout**	1	Standard output from the program
• **stderr**	2	Standard error output from the program

Normally input is from the keyboard or a file. Output, both stdout and stderr, normally go to the terminal, but you can redirect one or both of these to one or more files.

You can also specify additional file descriptors, designating them by a number 3 through 9, and redirect I/O through them.

6.2 File Redirection

Output redirection takes the output of a command and places it into a named file. Input redirection reads the file as input to the command. The following table summarizes the redirection options.

TABLE 6.1 **File Redirection**

Symbol	Redirection
>	output redirect
>!	same as above, but overrides **noclobber** option of ***csh***
>>	append output
>>!	same as above, but overrides **noclobber** option on **csh** and creates the file if it doesn't already exist.
|	pipe output to another command
<	input redirection
<<String	read from standard input until "**String**" is encountered as the only thing on the line. Also known as a "**here document**" (see Chapter 8).
<<\String	same as above, but don't allow shell substitutions

An example of output redirection is:

cat file1 file2 > file3

The above command concatenates **file1** then **file2** and redirects (sends) the output to **file3**. If **file3** doesn't already exist it is created. If it does exist it will either be truncated to zero length before the new contents are inserted, or the command will be rejected, if the **noclobber** option of the ***csh*** is set. (See the ***csh*** in Chapter 4). The original files, **file1** and **file2**, remain intact as separate entities.

Output is appended to a file in the form:

cat file1 >> file2

This command appends the contents of **file1** to the end of what already exists in **file2**. (Does not overwrite **file2**).

Input is redirected from a file in the form:

program < file

This command takes the input for *program* from **file**.

To pipe output to another command use the form:

command | command

This command makes the output of the first command the input of the second command.

6.2.1 Csh

>& file	redirect stdout and stderr to **file**
>>&	append stdout and stderr to **file**
|& command	pipe stdout and stderr to **command**

To redirect stdout and stderr to two separate files you need to first redirect stdout in a sub-shell, as in:

```
% (command > out_file) >& err_file
```

6.2.2 Sh

2> file	direct stderr to **file**
> file 2>&1	direct both stdout and stderr to **file**
>> file 2>&1	append both stdout and stderr to **file**
2>&1 | command	pipe stdout and stderr to **command**

To redirect stdout and stderr to two separate files you can do:

```
$ command 1> out_file 2> err_file
```

or, since the redirection defaults to stdout:

```
$ command > out_file 2> err_file
```

With the Bourne shell you can specify other file descriptors (3 through 9) and redirect output through them. This is done with the form:

n>&m	redirect file descriptor **n** to file descriptor **m**

We used the above to send stderr (2) to the same place as stdout (1), **2>&1**, when we wanted to have error messages and normal messages to go to **file** instead of the terminal. If we wanted only the error messages to go to the file we could do this by using a place holder file descriptor, 3. We'll first redirect 3 to 2, then redirect 2 to 1, and finally, we'll redirect 1 to 3:

```
$ (command 3>&2 2>&1 1>&3) > file
```

This sends stderr to 3 then to 1, and stdout to 3, which is redirected to 2. So, in effect, we've reversed file descriptors 1 and 2 from their normal meaning. We might use this in the following example:

```
$ (cat file 3>&2 2>&1 1>&3) > errfile
```

So if **file** is read the information is discarded from the command output, but if **file** can't be read the error message is put in **errfile** for your later use.

You can close file descriptors when you're done with them:

m<&-	closes an input file descriptor
<&-	closes stdin
m>&-	closes an output file descriptor
>&-	closes stdout

6.3 Other Special Command Symbols

In addition to file redirection symbols there are a number of other special symbols you can use on a command line. These include:

;	command separator
&	run the command in the background
&&	run the command following this only if the previous command completes successfully, e.g.: ***grep* string file && *cat* file**
||	run the command following only if the previous command did not complete successfully, e.g.: grep string file || echo "String not found."
( )	the commands within the parentheses are executed in a subshell. The output of the subshell can be manipulated as above.
' '	literal quotation marks. Don't allow any special meaning to any characters within these quotations.
\	escape the following character (take it literally)
" "	regular quotation marks. Allow variable and command substitution with theses quotations (does not disable $ and \ within the string).
'command'	take the output of this command and substitute it as an argument(s) on the command line
#	everything following until <newline> is a comment

The \ character can also be used to escape the **<newline>** character so that you can continue a long command on more than one physical line of text.

6.4 Wild Cards

The shell and some text processing programs will allow **meta-characters**, or **wild cards**, and replace them with pattern matches. For filenames these **meta-characters** and their uses are:

?	match any single character at the indicated position
*	match any string of zero or more characters
[abc...]	match any of the enclosed characters
[a-e]	match any characters in the range a,b,c,d,e
[!def]	match any characters not one of the enclosed characters, **sh** only
{abc,bcd,cde}	match any set of characters separated by comma (,) (no spaces), **csh** only
~	home directory of the current user, **csh** only
~user	home directory of the specified user, **csh** only

CHAPTER 7 Text Processing

7.1 Regular Expression Syntax

Some text processing programs, such as ***grep***, ***egrep***, ***sed***, ***awk*** and ***vi***, let you search on patterns instead of fixed strings. These text patterns are known as **regular expressions**. You form a regular expression by combining normal characters and special characters, also known as **meta-characters,** with the rules below. With these regular expressions you can do **pattern matching** on text data. Regular expressions come in three different forms:

• **Anchors**	which tie the pattern to a location on the line
• **Character sets**	which match a character at a single position
• **Modifiers**	which specify how many times to repeat the previous expression

Regular expression syntax is as follows. Some programs will accept all of these, others may only accept some.

.	match **any** single character except <newline>
*	match **zero or more** instances of the single character (or meta-character) immediately preceding it
[abc]	match any of the characters enclosed
[a-d]	match any character in the enclosed range
[^exp]	match any character **not** in the following expression
^abc	the regular expression must start at the **beginning of the line** (Anchor)
abc$	the regular expression must end at the **end of the line** (Anchor)
\	treat the next character literally. This is normally used to escape the meaning of special characters such as "." and "*".
\{n,m\}	match the regular expression preceding this a minimum number of **n** times and a maximum of **m** times (0 through 255 are allowed for n and m). The \{ and \} sets should be thought of as single operators. In this case the \ preceding the bracket does not escape its special meaning, but rather turns on a new one.
\<abc\>	will match the enclosed regular expression as long as it is a separate word. Word boundaries are defined as beginning with a <newline> or anything except a letter, digit or underscore (_) or ending with the same or a end-of-line character. Again the \< and \> sets should be thought of as single operators.

\(abc\)	saves the enclosed pattern in a buffer. Up to nine patterns can be saved for each line. You can reference these latter with the **\n** character set. Again the \(and \) sets should be thought of as single operators.
\n	where **n** is between 1 and 9. This matches the nth expression previously saved for this line. Expressions are numbered starting from the left. The **\n** should be thought of as a single operator.
&	print the previous search pattern (used in the replacement string)

There are a few meta-characters used only by ***awk*** and ***egrep***. These are:

+	match one or more of the preceding expression
?	match zero or more of the preceding expression
|	separator. Match either the preceding or following expression.
( )	group the regular expressions within and apply the match to the set.

Some examples of the more commonly used **regular expressions** are:

regular expression	**matches**
cat	the string **cat**
.at	any occurrence of a letter, followed by **at**, such as cat, rat, mat, bat, fat, hat
xy*z	any occurrence of an **x**, followed by zero or more **y**'s, followed by a **z**.
^cat	**cat** at the beginning of the line
cat$	**cat** at the end of the line
*	any occurrence of an asterisk
[cC]at	**cat** or **Cat**
[^a-zA-Z]	any occurrence of a non-alphabetic character
[0-9]$	any line ending with a number
[A-Z][A-Z]*	one or more upper case letters
[A-Z]*	zero or more upper case letters (In other words, anything.)

7.2 Text Processing Commands

TABLE 7.1 **Text Processing Commands**

Command/Syntax	What it will do
awk/nawk [options] *file*	scan for patterns in a file and process the results
grep/egrep/fgrep [options] 'search string' *file*	search the argument (in this case probably a file) for all occurrences of the search string, and list them.
sed [options] *file*	stream editor for editing files from a script or from the command line

7.2.1 grep

This section provides an introduction to the use of **regular expressions** and *grep*.

The ***grep*** utility is used to search for generalized regular expressions occurring in Unix files. Regular expressions, such as those shown above, are best specified in apostrophes (or single quotes) when specified in the ***grep*** utility. The ***egrep*** utility provides searching capability using an extended set of meta-characters. The syntax of the ***grep*** utility, some of the available options, and a few examples are shown below.

Syntax

grep [options] regexp [file[s]]

Common Options

-i	ignore case
-c	report only a count of the number of lines containing matches, not the matches themselves
-v	invert the search, displaying only lines that do not match
-n	display the line number along with the line on which a match was found
-s	work silently, reporting only the final status: 0, for match(es) found 1, for no matches 2, for errors
-l	list filenames, but not lines, in which matches were found

Examples

Consider the following file:

```
{unix prompt 5} cat num.list
    1       15    fifteen
    2       14    fourteen
    3       13    thirteen
    4       12    twelve
    5       11    eleven
    6       10    ten
    7        9    nine
    8        8    eight
    9        7    seven
    10       6    six
    11       5    five
    12       4    four
    13       3    three
    14       2    two
    15       1    one
```

Here are some ***grep*** examples using this file. In the first we'll search for the number **15**:

```
{unix prompt 6} grep '15' num.list
    1       15    fifteen
    15       1    one
```

Now we'll use the "**-c**" option to count the number of lines matching the search criterion:

```
{unix prompt 7} grep -c '15' num.list
    2
```

Here we'll be a little more general in our search, selecting for all lines containing the character **1** followed by either of **1**, **2** or **5**:

```
{unix prompt 8} grep '1[125]' num.list
    1       15    fifteen
    4       12    twelve
    5       11    eleven
    11       5    five
    12       4    four
    15       1    one
```

Now we'll search for all lines that **begin** with a **space**:

```
{unix prompt 9} grep '^ ' num.list
     1          15    fifteen
     2          14    fourteen
     3          13    thirteen
     4          12    twelve
     5          11    eleven
     6          10    ten
     7           9   nine
     8           8   eight
     9           7   seven
```

Or all lines that **don't begin** with a **space**:

```
{unix prompt 10} grep '^[^ ]' num.list
    10           6    six
    11           5    five
    12           4    four
    13           3    three
    14           2    two
    15           1    one
```

The latter could also be done by using the **-v** option with the original search string, e.g.:

```
{unix prompt 11} grep -v '^ ' num.list
    10           6    six
    11           5    five
    12           4    four
    13           3    three
    14           2    two
    15           1    one
```

Here we search for all lines that **begin** with the characters **1 through 9**:

```
{unix prompt 12} grep '^[1-9]' num.list
    10           6    six
    11           5    five
    12           4    four
    13           3    three
    14           2    two
    15           1    one
```

This example will search for any instances of **t** followed by **zero or more** occurrences of **e**:

```
{unix prompt 13} grep 'te*' num.list
    1        15    fifteen
    2        14    fourteen
    3        13    thirteen
    4        12    twelve
    6        10    ten
    8         8   eight
   13         3    three
   14         2    two
```

This example will search for any instances of **t** followed by **one or more** occurrences of **e**:

```
{unix prompt 14} grep 'tee*' num.list
    1        15    fifteen
    2        14    fourteen
    3        13    thirteen
    6        10    ten
```

We can also take our input from a program, rather than a file. Here we report on any lines output by the ***who*** program that begin with the letter **l**.

```
{unix prompt 15} who | grep '^l'
    lcondron ttyp0   Dec  1 02:41   (lcondron-pc.acs.)
```

7.2.2 sed

The non-interactive, stream editor, ***sed***, edits the input stream, line by line, making the specified changes, and sends the result to standard output.

Syntax

sed [options] edit_command [file]

The format for the editing commands are:

[address1[,address2]][function][arguments]

where the addresses are optional and can be separated from the function by spaces or tabs. The function is required. The arguments may be optional or required, depending on the function in use.

Line-number Addresses are decimal line numbers, starting from the first input line and incremented by one for each. If multiple input files are given the counter continues cumulatively through the files. The last input line can be specified with the "**$**" character.

Context Addresses are the regular expression patterns enclosed in slashes (/).

Commands can have 0, 1, or 2 comma-separated addresses with the following affects:

# of addresses	lines affected
0	every line of input
1	only lines matching the address
2	first line matching the first address and all lines until, and including, the line matching the second address. The process is then repeated on subsequent lines.

Substitution functions allow context searches and are specified in the form:

s/regular_expression_pattern/replacement_string/flag

and should be quoted with single quotes (') if additional options or functions are specified. These patterns are identical to context addresses, except that while they are normally enclosed in slashes (/), any normal character is allowed to function as the delimiter, other than <space> and <newline>. The replacement string is not a regular expression pattern; characters do not have special meanings here, except:

&	substitute the string specified by regular_expression_pattern
\n	substitute the nth string matched by regular_expression_pattern enclosed in '\(', '\)' pairs.

These special characters can be escaped with a backslash (\) to remove their special meaning.

Common Options

-e script	edit script
-n	don't print the default output, but only those lines specified by p or s///p functions
-f script_file	take the edit scripts from the file, script_file

Valid flags on the substitution functions include:

d	delete the pattern
g	globally substitute the pattern
p	print the line

Examples

This example changes all incidents of a comma (,) into a comma followed by a space (,) when doing output:

```
% cat filey | sed s/,/,\ /g
```

The following example removes all incidents of **Jr** preceded by a space (**Jr**) in **filey**:

```
% cat filey | sed s/\ Jr//g
```

To perform multiple operations on the input precede each operation with the **-e** (edit) option and quote the strings. For example, to filter for lines containing "Date: " and "From: " and replace these without the colon (:), try:

```
sed -e 's/Date: /Date /' -e 's/From: /From /'
```

To print only those lines of the file from the one beginning with "Date:" up to, and including, the one beginning with "Name:" try:

```
sed -n '/^Date:/,/^Name:/p'
```

To print only the first 10 lines of the input (a replacement for ***head***):

```
sed -n 1,10p
```

7.2.3 awk, nawk, gawk

awk is a pattern scanning and processing language. Its name comes from the last initials of the three authors: Alfred. V. Aho, Brian. W. Kernighan, and Peter. J. Weinberger. ***nawk*** is **new *awk***, a newer version of the program, and ***gawk*** is **gnu *awk***, from the Free Software Foundation. Each version is a little different. Here we'll confine ourselves to simple examples which should be the same for all versions. On some OSs ***awk*** is really ***nawk***.

awk searches its input for patterns and performs the specified operation on each line, or fields of the line, that contain those patterns. You can specify the pattern matching statements for ***awk*** either on the command line, or by putting them in a file and using the **-f program_file** option.

Syntax

awk program [file]

where **program** is composed of one or more:

pattern { action }

fields. Each input line is checked for a pattern match with the indicated action being taken on a match. This continues through the full sequence of patterns, then the next line of input is checked.

Input is divided into **records** and **fields**. The default **record** separator is **<newline>**, and the variable **NR** keeps the record count. The default **field** separator is whitespace, **spaces** and **tabs**, and the variable **NF** keeps the field count. Input field, **FS**, and record, **RS**, separators can be set at any time to match any single character. Output field, **OFS**, and record, **ORS**, separators can also be changed to any single character, as desired. **$n**, where **n** is an integer, is used to represent the **n**th field of the input record, while **$0** represents the entire input record.

BEGIN and **END** are special patterns matching the beginning of input, before the first field is read, and the end of input, after the last field is read, respectively.

Printing is allowed through the ***print***, and formatted print, ***printf***, statements.

Patterns may be regular expressions, arithmetic relational expressions, string-valued expressions, and boolean combinations of any of these. For the latter the patterns can be combined with the boolean operators below, using parentheses to define the combination:

||	or
&&	and
!	not

Comma separated patterns define the **range** for which the pattern is applicable, e.g.:

/first/,/last/

selects all lines starting with the one containing **first**, and continuing inclusively, through the one containing **last**.

To select lines 15 through 20 use the pattern range:

```
NR == 15, NR == 20
```

Regular expressions must be enclosed with slashes (/) and meta-characters can be escaped with the backslash (\). Regular expressions can be grouped with the operators:

|	or, to separate alternatives
+	one or more
?	zero or one

A regular expression match can be either of:

~	contains the expression
!~	does not contain the expression

So the program:

```
$1 ~ /[Ff]rank/
```

is true if the first field, $1, contains "Frank" or "frank" anywhere within the field. To match a field identical to "Frank" or "frank" use:

```
$1 ~ /^[Ff]rank$/
```

Relational expressions are allowed using the relational operators:

<	less than
<=	less than or equal to
==	equal to
>=	greater than or equal to
!=	not equal to
>	greater than

Offhand you don't know if variables are strings or numbers. If neither operand is known to be numeric, than string comparisons are performed. Otherwise, a numeric comparison is done. In the absence of any information to the contrary, a string comparison is done, so that:

```
$1 > $2
```

will compare the string values. To ensure a numerical comparison do something similar to:

```
( $1 + 0 ) > $2
```

The **mathematical functions**: exp, log and sqrt are built-in.

Some other **built-in functions** include:

index(s,t)	returns the position of string s where **t** first occurs, or 0 if it doesn't
length(s)	returns the length of string s
substr(s,m,n)	returns the **n**-character substring of **s**, beginning at position **m**

Arrays are declared automatically when they are used, e.g.:

```
arr[i] = $1
```

assigns the first field of the current input record to the ith element of the array.

Flow control statements using **if-else**, **while**, and **for** are allowed with **C** type syntax:

```
for (i=1; i <= NF; i++) {actions}
while (i<=NF) {actions}
if (i<NF) {actions}
```

Common Options

-f program_file	read the commands from program_file
-Fc	use character **c** as the field separator character

Examples

```
% cat filex | tr a-z A-Z | awk -F: '{printf ("7R   %-6s %-9s   %-24s \n",$1,$2,$3)}'>upload.file
```

***cat*s filex**, which is formatted as follows:

```
nfb791:99999999:smith
7ax791:999999999:jones
8ab792:99999999:chen
8aa791:999999999:mcnulty
```

changes all lower case characters to upper case with the ***tr*** utility, and formats the file into the following which is written into the file **upload.file**:

```
7R NFB791 99999999  SMITH
7R 7AX791 999999999 JONES
7R 8AB792 99999999  CHEN
7R 8AA791 999999999 MCNULTY
```

CHAPTER 8 Other Useful Commands

8.1 Working With Files

This section will describe a number of commands that you might find useful in examining and manipulating the contents of your files.

TABLE 8.1 **File utilities**

Command/Syntax	What it will do
cmp [options] *file1 file2*	compare two files and list where differences occur (text or binary files)
cut [options] [*file*(s)]	cut specified field(s)/character(s) from lines in file(s)
diff [options] *file1 file2*	compare the two files and display the differences (text files only)
file [options] *file*	classify the file type
find *directory* [options] [actions]	find files matching a type or pattern
ln [options] *source_file target*	link the *source_file* to the *target*
paste [options] *file*	paste field(s) onto the lines in *file*
sort [options] *file*	sort the lines of the *file* according to the options chosen
strings [options] *file*	report any sequence of 4 or more printable characters ending in <NL> or <NULL>. Usually used to search binary files for ASCII strings.
tee [options] *file*	copy stdout to one or more files
touch [options] [date] *file*	create an empty file, or update the access time of an existing file
tr [options] *string1 string2*	translate the characters in string1 from stdin into those in string2 in stdout
uniq [options] *file*	remove repeated lines in a file
wc [options] [*file*(s)]	display word (or character or line) count for *file*(s)

8.1.1 cmp - compare file contents

The ***cmp*** command compares two files, and (without options) reports the location of the first difference between them. It can deal with both binary and ASCII file comparisons. It does a byte-by-byte comparison.

Syntax

cmp [options] file1 file2 [skip1] [skip2]

The **skip** numbers are the number of bytes to skip in each file before starting the comparison.

Common Options

-l	report on each difference
-s	report exit status only, not byte differences

Examples

Given the files mon.logins:and tues.logins:

ageorge	ageorge
bsmith	cbetts
cbetts	jchen
jchen	jdoe
jmarsch	jmarsch
lkeres	lkeres
mschmidt	proy
sphillip	sphillip
wyepp	wyepp

The comparison of the two files yields:

```
% cmp mon.logins tues.logins
    mon.logins tues.logins differ: char 9, line 2
```

The default it to report only the first difference found.

This command is useful in determining which version of a file should be kept when there is more than one version.

8.1.2 diff - differences in files

The ***diff*** command compares two files, directories, etc, and reports all differences between the two. It deals only with ASCII files. It's output format is designed to report the changes necessary to convert the first file into the second.

Syntax

diff [options] file1 file2

Common Options

-b	ignore trailing blanks
-i	ignore the case of letters
-w	ignore **<space>** and **<tab>** characters
-e	produce an output formatted for use with the editor, ***ed***
-r	apply diff recursively through common sub-directories

Examples

For the mon.logins and tues.logins files above, the difference between them is given by:

```
% diff mon.logins tues.logins
    2d1
    < bsmith
    4a4
    > jdoe
    7c7
    < mschmidt
    ---
    > proy
```

Note that the output lists the differences as well as in which file the difference exists. Lines in the first file are preceded by "< ", and those in the second file are preceded by "> ".

8.1.3 cut - select parts of a line

The ***cut*** command allows a portion of a file to be extracted for another use.

Syntax

cut [options] file

Common Options

-c character_list	character positions to select (first character is 1)
-d delimiter	field delimiter (defaults to <TAB>)
-f field_list	fields to select (first field is 1)

Both the character and field lists may contain comma-separated or blank-character-separated numbers (in increasing order), and may contain a hyphen (-) to indicate a range. Any numbers missing at either before (e.g. -5) or after (e.g. 5-) the hyphen indicates the full range starting with the first, or ending with the last character or field, respectively. Blank-character-separated lists must be enclosed in quotes. The field delimiter should be enclosed in quotes if it has special meaning to the shell, e.g. when specifying a **<space>** or **<TAB>** character.

Examples

In these examples we will use the file **users**:

```
jdoe        John Doe        4/15/96
lsmith      Laura Smith     3/12/96
pchen       Paul Chen       1/5/96
jhsu        Jake Hsu        4/17/96
sphilip     Sue Phillip     4/2/96
```

If you only wanted the username and the user's real name, the ***cut*** command could be used to get only that information:

```
% cut -f 1,2 users
   jdoe        John Doe
   lsmith      Laura Smith
   pchen       Paul Chen
   jhsu        Jake Hsu
   sphilip     Sue Phillip
```

The ***cut*** command can also be used with other options. The **-c** option allows characters to be the selected cut. To select the first 4 characters:

```
% cut -c 1-4 users
```

This yields:

```
jdoe
lsmi
pche
jhsu
sphi
```

thus cutting out only the first 4 characters of each line.

8.1.4 paste - merge files

The ***paste*** command allows two files to be combined side-by-side. The default delimiter between the columns in a paste is a tab, but options allow other delimiters to be used.

Syntax

paste [options] file1 file2

Common Options

-d list	list of delimiting characters
-s	concatenate lines

The list of **delimiters** may include a single character such as a comma; a quoted string, such as a space; or any of the following escape sequences:

\n	<newline> character
\t	<tab> character
\\	backslash character
\0	empty string (non-null character)

It may be necessary to quote delimiters with special meaning to the shell.

A hyphen (-) in place of a file name is used to indicate that field should come from standard input.

Examples

Given the file **users**:

```
jdoe        John Doe        4/15/96
lsmith      Laura Smith     3/12/96
pchen       Paul Chen       1/5/96
jhsu        Jake Hsu        4/17/96
sphilip     Sue Phillip     4/2/96
```

and the file **phone**:

```
John Doe        555-6634
Laura Smith     555-3382
Paul Chen       555-0987
Jake Hsu        555-1235
Sue Phillip     555-7623
```

the ***paste*** command can be used in conjunction with the ***cut*** command to create a new file, **listing**, that includes the username, real name, last login, and phone number of all the users. First, extract the phone numbers into a temporary file, **temp.file**:

```
% cut -f2 phone > temp.file
    555-6634
    555-3382
    555-0987
    555-1235
    555-7623
```

The result can then be pasted to the end of each line in **users** and directed to the new file, **listing**:

```
% paste users temp.file > listing
    jdoe        John Doe        4/15/96     237-6634
    lsmith      Laura Smith     3/12/96     878-3382
    pchen       Paul Chen       1/5/96      888-0987
    jhsu        Jake Hsu        4/17/96     545-1235
    sphilip     Sue Phillip     4/2/96      656-7623
```

This could also have been done on one line without the temporary file as:

```
% cut -f2 phone | paste users - > listing
```

with the same results. In this case the hyphen (-) is acting as a placeholder for an input field (namely, the output of the ***cut*** command).

8.1.5 touch - create a file

The touch command can be used to create a new (empty) file or to update the last access date/time on an existing file. The command is used primarily when a script requires the pre-existence of a file (for example, to which to append information) or when the script is checking for last date or time a function was performed.

Syntax

touch [options] [date_time] file

touch [options] [-t time] file

Common Options

-a	change the access time of the file (SVR4 only)
-c	don't create the file if it doesn't already exist
-f	force the touch, regardless of read/write permissions
-m	change the modification time of the file (SVR4 only)
-t time	use the time specified, not the current time (SVR4 only)

When setting the "**-t time**" option it should be in the form:

[[CC]YY]MMDDhhmm[.SS]

where:

CC	first two digits of the year
YY	second two digits of the year
MM	month, 01-12
DD	day of month, 01-31
hh	hour of day, 00-23
mm	minute, 00-59
SS	second, 00-61

The date_time options has the form:

MMDDhhmm[YY]

where these have the same meanings as above.

The date cannot be set to be before 1969 or after January 18, 2038.

Examples

To create a file:

```
% touch filename
```

8.1.6 wc - count words in a file

wc stands for "word count"; the command can be used to count the number of lines, characters, or words in a file.

Syntax

wc [options] file

Common Options

-c	count bytes
-m	count characters (SVR4)
-l	count lines
-w	count words

If no options are specified it defaults to "**-lwc**".

Examples

Given the file **users**:

```
jdoe        John Doe       4/15/96
lsmith      Laura Smith    3/12/96
pchen       Paul Chen      1/5/96
jhsu        Jake Hsu       4/17/96
sphilip     Sue Phillip    4/2/96
```

the result of using a *wc* command is as follows:

```
% wc users
    5    20    121 users
```

The first number indicates the number of lines in the file, the second number indicates the number of words in the file, and the third number indicates the number of characters.

Using the *wc* command with one of the options (**-l**, lines; **-w**, words; or **-c**, characters) would result in only one of the above. For example, "***wc -l users***" yields the following result:

```
5 users
```

8.1.7 ln - link to another file

The ***ln*** command creates a "link" or an additional way to access (or gives an additional name to) another file.

Syntax

ln [options] source [target]

If not specified **target** defaults to a file of the same name in the present working directory.

Common Options

-f	force a link regardless of target permissions; don't report errors (SVR4 only)
-s	make a symbolic link

Examples

A **symbolic link** is used to create a new path to another file or directory. If a group of users, for example, is accustomed to using a command called ***chkmag***, but the command has been rewritten and is now called ***chkit***, creating a symbolic link so the users will automatically execute ***chkit*** when they enter the command ***chkmag*** will ease transition to the new command.

A symbolic link would be done in the following way:

```
% ln -s chkit chkmag
```

The long listing for these two files is now as follows:

```
16 -rwxr-x---  1 lindadb  acs    15927 Apr 23 04:10 chkit
 1 lrwxrwxrwx  1 lindadb  acs        5 Apr 23 04:11 chkmag -> chkit
```

Note that while the permissions for ***chkmag*** are open to all, since it is linked to ***chkit***, the permissions, group and owner characteristics for ***chkit*** will be enforced when ***chkmag*** is run.

With a symbolic link, the link can exist without the file or directory it is linked to existing first.

A **hard link** can only be done to another file on the same file system, but not to a directory (except by the superuser). A hard link creates a new directory entry pointing to the same inode as the original file. The file linked to must exist before the hard link can be created. The file will not be deleted until all the hard links to it are removed. To link the two files above with a hard link to each other do:

```
% ln chkit chkmag
```

Then a long listing shows that the **inode** number (742) is the same for each:

```
% ls -il chkit chkmag
    742 -rwxr-x---  2 lindadb  acs    15927 Apr 23 04:10 chkit
    742 -rwxr-x---  2 lindadb  acs    15927 Apr 23 04:10 chkmag
```

8.1.8 sort - sort file contents

The *sort* command is used to order the lines of a file. Various options can be used to choose the order as well as the field on which a file is sorted. Without any options, the sort compares entire lines in the file and outputs them in ASCII order (numbers first, upper case letters, then lower case letters).

Syntax

sort [options] [+pos1 [-pos2]] file

Common Options

-b	ignore leading blanks (<**space**> & <**tab**>) when determining starting and ending characters for the sort key
-d	dictionary order, only letters, digits, <**space**> and <**tab**> are significant
-f	fold upper case to lower case
-k keydef	sort on the defined keys (not available on all systems)
-i	ignore non-printable characters
-n	numeric sort
-o outfile	output file
-r	reverse the sort
-t char	use char as the field separator character
-u	unique; omit multiple copies of the same line (after the sort)
+pos1 [-pos2]	(old style) provides functionality similar to the "-k keydef" option.

For the **+/-position** entries **pos1** is the starting word number, beginning with **0** and **pos2** is the ending word number. When **-pos2** is omitted the sort field continues through the end of the line. Both pos1 and **pos2** can be written in the form **w.c**, where **w** is the word number and **c** is the character within the word. For **c 0** specifies the delimiter preceding the first character, and **1** is the first character of the word. These entries can be followed by type modifiers, e.g. **n** for numeric, **b** to skip blanks, etc.

The **keydef** field of the "**-k**" option has the syntax:

start_field [type] [,end_field [type]]

where:

start_field, end_field	define the keys to restrict the sort to a portion of the line
type	modifies the sort, valid modifiers are given the single characters (bdfiMnr) from the similar sort options, e.g. a type **b** is equivalent to "**-b**", but applies only to the specified field

Examples

In the file **users**:

```
jdoe        John Doe        4/15/96
lsmith      Laura Smith     3/12/96
pchen       Paul Chen       1/5/96
jhsu        Jake Hsu        4/17/96
sphilip     Sue Phillip     4/2/96
```

sort users yields the following:

```
jdoe        John Doe        4/15/96
jhsu        Jake Hsu        4/17/96
lsmith      Laura Smith     3/12/96
pchen       Paul Chen       1/5/96
sphilip     Sue Phillip     4/2/96
```

If, however, a listing sorted by last name is desired, use the option to specify which field to sort on (fields are numbered starting at 0):

```
% sort +2 users:
    pchen       Paul Chen       1/5/96
    jdoe        John Doe        4/15/96
    jhsu        Jake Hsu        4/17/96
    sphilip     Sue Phillip     4/2/96
    lsmith      Laura Smith     3/12/96
```

To sort in reverse order:

```
% sort -r users:
    sphilip     Sue Phillip     4/2/96
    pchen       Paul Chen       1/5/96
    lsmith      Laura Smith     3/12/96
    jhsu        Jake Hsu        4/17/96
    jdoe        John Doe        4/15/96
```

A particularly useful ***sort*** option is the **-u** option, which eliminates any duplicate entries in a file while ordering the file. For example, the file todays.logins:

```
sphillip
jchen
jdoe
lkeres
jmarsch
ageorge
lkeres
proy
jchen
```

shows a listing of each username that logged into the system today. If we want to know how many unique users logged into the system today, using sort with the **-u** option will list each user only once. (The command can then be piped into "***wc -l***" to get a number):

```
% sort -u todays.logins
    ageorge
    jchen
    jdoe
    jmarsch
    lkeres
    proy
    sphillip
```

8.1.9 tee - copy command output

tee sends standard in to specified files and also to standard out. It's often used in command pipelines.

Syntax

tee [options] [file[s]]

Common Options

-a	append the output to the files
-i	ignore interrupts

Examples

In this first example the output of ***who*** is displayed on the screen and stored in the file **users.file**:

```
brigadier: condron [55]> who | tee users.file
    condron   ttyp0    Apr 22 14:10    (lcondron-pc.acs.)
    frank     ttyp1    Apr 22 16:19    (nyssa)
    condron   ttyp9    Apr 22 15:52    (lcondron-mac.acs)

brigadier: condron [56]> cat users.file
    condron   ttyp0    Apr 22 14:10    (lcondron-pc.acs.)
    frank     ttyp1    Apr 22 16:19    (nyssa)
    condron   ttyp9    Apr 22 15:52    (lcondron-mac.acs)
```

In this next example the output of ***who*** is sent to the files **users.a** and **users.b**. It is also piped to the ***wc*** command, which reports the line count.

```
brigadier: condron [57]> who | tee users.a users.b | wc -l
          3

brigadier: condron [58]> cat users.a
    condron   ttyp0    Apr 22 14:10    (lcondron-pc.acs.)
    frank     ttyp1    Apr 22 16:19    (nyssa)
    condron   ttyp9    Apr 22 15:52    (lcondron-mac.acs)

brigadier: condron [59]> cat users.b
    condron   ttyp0    Apr 22 14:10    (lcondron-pc.acs.)
    frank     ttyp1    Apr 22 16:19    (nyssa)
    condron   ttyp9    Apr 22 15:52    (lcondron-mac.acs)
```

In the following example a long directory listing is sent to the file **files.long**. It is also piped to the ***grep*** command which reports which files were last modified in August.

```
brigadier: condron [60]> ls -l | tee files.long |grep Aug
      1 drwxr-sr-x  2 condron       512 Aug  8  1995 News/
      2 -rw-r--r--  1 condron      1076 Aug  8  1995 magnus.cshrc
      2 -rw-r--r--  1 condron      1252 Aug  8  1995 magnus.login
brigadier: condron [63]> cat files.long
   total 34
      2 -rw-r--r--  1 condron      1253 Oct 10  1995 #.login#
      1 drwx------  2 condron       512 Oct 17  1995 Mail/
      1 drwxr-sr-x  2 condron       512 Aug  8  1995 News/
      5 -rw-r--r--  1 condron      4299 Apr 21 00:18 editors.txt
      2 -rw-r--r--  1 condron      1076 Aug  8  1995 magnus.cshrc
      2 -rw-r--r--  1 condron      1252 Aug  8  1995 magnus.login
      7 -rw-r--r--  1 condron      6436 Apr 21 23:50 resources.txt
      4 -rw-r--r--  1 condron      3094 Apr 18 18:24 telnet.ftp
      1 drwxr-sr-x  2 condron       512 Apr 21 23:56 uc/
      1 -rw-r--r--  1 condron      1002 Apr 22 00:14 uniq.tee.txt
      1 -rw-r--r--  1 condron      1001 Apr 20 15:05 uniq.tee.txt~
      7 -rw-r--r--  1 condron      6194 Apr 15 20:18 unixgrep.txt
```

8.1.10 uniq - remove duplicate lines

uniq filters duplicate adjacent lines from a file.

Syntax

```
uniq  [options]  [+|-n]  file  [file.new]
```

Common Options

-d	one copy of only the repeated lines
-u	select only the lines not repeated
+n	ignore the first **n characters**
-s n	same as above (SVR4 only)
-n	skip the first **n fields**, including any blanks (**<space>** & **<tab>**)
-f fields	same as above (SVR4 only)

Examples

Consider the following file and example, in which ***uniq*** removes the 4th line from **file** and places the result in a file called **file.new**.

```
{unix prompt 1} cat file
    1 2 3 6
    4 5 3 6
    7 8 9 0
    7 8 9 0
{unix prompt 2} uniq  file  file.new
{unix prompt 3} cat  file.new
    1 2 3 6
    4 5 3 6
    7 8 9 0
```

Below, the **-n** option of the ***uniq*** command is used to skip the first 2 fields in **file**, and filter out lines which are duplicates from the 3rd field onward.

```
{unix prompt 4} uniq  -2  file
    1 2 3 6
    7 8 9 0
```

8.1.11 strings - find ASCII strings

To search a binary file for printable, ASCII, strings use the ***strings*** command. It searches for any sequence of 4 or more ASCII characters terminated by a <**newline**> or null character. I find this command useful for searching for file names and possible error messages within compiled programs that I don't have source code for.

Syntax

strings [options] file

Common Options

-n number	use number as the minimum string length, rather than 4 (SVR4 only)
-number	same as above
-t format	precede the string with the byte offset from the start of the file, where format is one of: **d** = decimal, **o** = octal, **x** = hexadecimal (SVR4 only)
-o	precede the string with the byte offset in decimal (BSD only)

Examples

```
% strings /bin/cut
    SUNW_OST_OSCMD
    no delimiter specified
    invalid delimiter
    b:c:d:f:ns
    cut: -n may only be used with -b
    cut: -d may only be used with -f
    cut: -s may only be used with -f
    no list specified
    cut: cannot open %s
    invalid range specifier
    too many ranges specified
    ranges must be increasing
    invalid character in range
    Internal error processing input
    invalid multibyte character
    unable to allocate enough memory
    unable to allocate enough memory
    cut:
    usage: cut -b list [-n] [filename ...]
        cut -c list [filename ...]
        cut -f list [-d delim] [-s] [filename]
```

8.1.12 file - file type

This program, ***file***, examines the selected file and tries to determine what type of file it is. It does this by reading the first few bytes of the file and comparing them with the table in **/etc/magic**. It can determine ASCII text files, tar formatted files, compressed files, etc.

Syntax

file [options] [-m magic_file] [-f file_list] file

Common Options

-c	check the magic file for errors in format
-f file_list	**file_list** contains a list of files to examine
-h	don't follow symbolic links (SVR4 only)
-L	follow symbolic links (BSD only)
-m magic_file	use **magic_file** as the magic file instead of /etc/magic

Examples

Below we list the output from the command "***file filename***" for some representative files.

/etc/magic:	ascii text
/usr/local/bin/gzip:	Sun demand paged SPARC executable dynamically linked
/usr/bin/cut:	ELF 32-bit MSB executable SPARC Version 1, dynamically linked, stripped
source.tar:	USTAR tar archive
source.tar.Z:	compressed data block compressed 16 bits

8.1.13 tr - translate characters

The ***tr*** command translates characters from stdin to stdout.

Syntax

tr [options] string1 [string2]

With no options the characters in **string1** are translated into the characters in **string2**, character by character in the string arrays. The first character in **string1** is translated into the first character in **string2**, etc.

A range of characters in a string is specified with a hyphen between the upper and lower characters of the range, e.g. to specify all lower case alphabetic characters use '**[a-z]**'.

Repeated characters in **string2** can be represented with the '**[x*n]**' notation, where character **x** is repeated **n** times. If **n** is **0** or absent it is assumed to be as large as needed to match **string1**.

Characters can include **\octal** (BSD and SVR4) and **\character** (SVR4 only) notation. Here "**octal**" is replaced by the one, two, or three octal integer sequence encoding the ASCII character and "**character**" can be one of:

b	back space
f	form feed
n	new line
r	carriage return
t	tab
v	vertical tab

The SVR4 version of ***tr*** allows the operand "**:class:**" in the string field where **class** can take on character classification values, including:

alpha	alphabetic characters
lower	lower case alphabetic characters
upper	upper case alphabetic characters

Common Options

-c	complement the character set in **string1**
-d	delete the characters in **string1**
-s	squeeze a string of repeated characters in **string1** to a single character

Examples

The following examples will use as input the file, a list of P. G. Wodehouse Jeeves & Wooster books.

The Inimitable Jeeves [1923]	The Mating Season [1949]
Carry On, Jeeves [1925]	Ring for Jeeves [1953]
Very Good, Jeeves [1930]	Jeeves and the Feudal Spirit [1954]
Thank You, Jeeves [1934]	Jeeves in the Offing [1960]
Right Ho, Jeeves [1934]	Stiff Upper Lip, Jeeves [1963]
The Code of the Woosters [1938]	Much Obliged, Jeeves [1971]
Joy in the Morning [1946]	Aunts Aren't Gentlemen [1974]

To translate all lower case alphabetic characters to upper case we could use either of:

```
tr '[a-z]' '[A-Z]'          or          tr '[:lower:]' '[:upper:]'
```

Since ***tr*** reads from stdin we first ***cat*** the file and pipe the output to ***tr***, as in:

```
% cat wodehouse | tr '[a-z]' '[A-Z]'
    THE INIMITABLE JEEVES [1923]          THE MATING SEASON [1949]
    CARRY ON, JEEVES [1925]               RING FOR JEEVES [1953]
    VERY GOOD, JEEVES [1930]              JEEVES AND THE FEUDAL SPIRIT [1954]
    THANK YOU, JEEVES [1934]              JEEVES IN THE OFFING [1960]
    RIGHT HO, JEEVES [1934]               STIFF UPPER LIP, JEEVES [1963]
    THE CODE OF THE WOOSTERS [1938]       MUCH OBLIGED, JEEVES [1971]
    JOY IN THE MORNING [1946]             AUNTS AREN'T GENTLEMEN [1974]
```

We could delete all numbers with:

```
% cat wodehouse | tr -d '[0-9]'
    The Inimitable Jeeves []              The Mating Season []
    Carry On, Jeeves []                   Ring for Jeeves []
    Very Good, Jeeves []                  Jeeves and the Feudal Spirit []
    Thank You, Jeeves []                  Jeeves in the Offing []
    Right Ho, Jeeves []                   Stiff Upper Lip, Jeeves []
    The Code of the Woosters []           Much Obliged, Jeeves []
    Joy in the Morning []                 Aunts Aren't Gentlemen []
```

To squeeze all multiple occurrences of the characters e, r, and f:

```
% cat wodehouse | tr -s 'erf'
    The Inimitable Jeves [1923]           The Mating Season [1949]
    Cary On, Jeves [1925]                 Ring for Jeves [1953]
    Very Good, Jeves [1930]               Jeves and the Feudal Spirit [1954]
    Thank You, Jeves [1934]               Jeves in the Ofing [1960]
    Right Ho, Jeves [1934]                Stif Upper Lip, Jeves [1963]
    The Code of the Woosters [1938]       Much Obliged, Jeves [1971]
    Joy in the Morning [1946]             Aunts Aren't Gentlemen [1974]
```

8.1.14 find - find files

The *find* command will recursively search the indicated directory tree to find files matching a type or pattern you specify. *find* can then list the files or execute arbitrary commands based on the results.

Syntax

find directory [search options] [actions]

Common Options

For the time search options the notation in days, **n** is:

+n	more than **n** days
n	exactly **n** days
-n	less than **n** days

Some file characteristics that *find* can search for are:

time that the file was last accessed or changed	
-atime n	access time, true if accessed **n** days ago
-ctime n	change time, true if the files status was changed **n** days ago
-mtime n	modified time, true if the files data was modified **n** days ago
-newer filename	true if newer than **filename**
-type type	**type** of **file**, where **type** can be:
b	block special file
c	character special file
d	directory
l	symbolic link
p	named pipe (fifo)
f	regular file
-fstype type	**type** of **file system**, where **type** can be any valid file system type, e.g.: **ufs** (Unix File System) and **nfs** (Network File System)
-user username	true if the file belongs to the user **username**
-group groupname	true if the file belongs to the group **groupname**
-perm [-]mode	permissions on the file, where **mode** is the octal modes for the ***chmod*** command. When **mode** is precede by the minus sign only the bits that are set are compared.
-exec command	execute **command**. The end of **command** is indicated by and escaped semicolon (\;). The command argument, {}, replaces the current path name.
-name filename	true if the file is named **filename**. Wildcard pattern matches are allowed if the meta-character is escaped from the shell with a backslash (\).
-ls	always true. It prints a long listing of the current pathname.
-print	print the pathnames found (default for SVR4, not for BSD)

Complex expressions are allowed. Expressions should be grouped within parenthesis (escaping the parenthesis with a backslash to prevent the shell from interpreting them). The exclamation symbol (**!**) can be used to **negate** an expression. The operators: **-a** (**and**) and **-o** (**or**) are used to group expressions.

Examples

find will recursively search through sub-directories, but for the purpose of these examples we will just use the following files:

```
14 -rw-r--r--   1 frank    staff      6682 Feb  5 10:04 library
 6 -r--r-----   1 frank    staff      3034 Mar 16  1995 netfile
34 -rw-r--r--   1 frank    staff     17351 Feb  5 10:04 standard
 2 -rwxr-xr-x   1 frank    staff       386 Apr 26 09:51 tr25*
```

To find all files newer than the file, library:

```
% find . -newer library -print
    ./tr25
    ./standard
```

To find all files with general read or execute permission set, and then to change the permissions on those files to disallow this:

```
% find . \( -perm -004 -o -perm -001 \)  -exec chmod o-rx { } \; -exec ls -al { } \;
    -rw-r-----   1 frank    staff      6682 Feb  5 10:04 ./library
    -rwxr-x---   1 frank    staff       386 Apr 26 09:51 ./tr25
    -rw-r-----   1 frank    staff     17351 Feb  5 10:04 ./standard
```

In this example the parentheses and semicolons are escaped with a backslash to prevent the shell from interpreting them. The curly brackets are automatically replaced by the results from the previous search and the semicolon ends the command.

We could search for any file name containing the string "ar" with:

```
% find . -name \*ar\* -ls
    326584    7 -rw-r-----  1 frank staff  6682 Feb  5 10:04 ./library
    326585   17 -rw-r-----  1 frank staff 17351 Feb  5 10:04 ./standard
```

where the **-ls** option prints out a long listing, including the inode numbers.

8.2 File Archiving, Compression and Conversion

TABLE 8.2 **File Archiving, Compression and Conversion Commands**

Command/Syntax	What it will do
compress/uncompress/zcat [options] *file[.Z]*	compress or uncompress a file. Compressed files are stored with a .Z ending.
dd [*if*=infile] [*of*=outfile] [operand=value]	copy a file, converting between ASCII and EBCDIC or swapping byte order, as specified
gzip/gunzip/zcat [options] *file[.gz]*	compress or uncompress a file. Compressed files are stored with a **.gz** ending
od [options] *file*	octal dump a binary file, in octal, ASCII, hex, decimal, or character mode.
tar key[options] [*file*(s)]	tape archiver--refer to man pages for details on creating, listing, and retrieving from archive files. Tar files can be stored on tape or disk.
uudecode [*file*]	decode a uuencoded file, recreating the original file
uuencode [*file*] *new_name*	encode binary file to 7-bit ASCII, useful when sending via email, to be decoded as new_name at destination

8.2.1 File Compression

The ***compress*** command is used to reduce the amount of disk space utilized by a file. When a file has been compressed using the ***compress*** command, a suffix of ***.Z*** is appended to the file name. The ownership modes and access and modification times of the original file are preserved. ***uncompress*** restores the files originally compressed by ***compress***.

Syntax

compress [options] [file]

uncompress [options] [file.Z]

zcat [file.Z]

Common Options

-c	write to standard output and don't create or change any files
-f	force compression of a file, even if it doesn't reduce the size of the file or if the target file (file.Z) already exists.
-v	verbose. Report on the percentage reduction for the file.

zcat writes to standard output. It is equivalent to "***uncompress -c***".

Examples

Given the files:

```
 96 -rw-r--r--   1 lindadb   acs   45452 Apr 24 09:13 logins.beauty
184 -rw-r--r--   1 lindadb   acs   90957 Apr 24 09:13 logins.bottom
152 -rw-r--r--   1 lindadb   acs   75218 Apr 24 09:13 logins.photon
168 -rw-r--r--   1 lindadb   acs   85970 Apr 24 09:13 logins.top
```

These can be compressed with:

```
% compress logins.*
```

which creates the files:

```
24 -rw-r--r--   1 lindadb   acs    8486 Apr 24 09:13 logins.beauty.Z
40 -rw-r--r--   1 lindadb   acs   16407 Apr 24 09:13 logins.bottom.Z
24 -rw-r--r--   1 lindadb   acs   10909 Apr 24 09:13 logins.photon.Z
32 -rw-r--r--   1 lindadb   acs   16049 Apr 24 09:13 logins.top.Z
```

The original files are lost.

To display a compressed file, the ***zcat*** command is used:

```
% zcat logins.beauty.Z | head
    beauty:01/22/94:#total logins,4338:#different UIDs,2290
    beauty:01/23/94:#total logins,1864:#different UIDs,1074
    beauty:01/24/94:#total logins,2317:#different UIDs,1242
    beauty:01/25/94:#total logins,3673:#different UIDs,2215
    beauty:01/26/94:#total logins,3532:#different UIDs,2216
    beauty:01/27/94:#total logins,3096:#different UIDs,1984
    beauty:01/28/94:#total logins,3724:#different UIDs,2212
    beauty:01/29/94:#total logins,3460:#different UIDs,2161
    beauty:01/30/94:#total logins,1408:#different UIDs,922
    beauty:01/31/94:#total logins,2175:#different UIDs,1194
```

A display of the file using commands other than ***zcat*** yields an unreadable, binary, output.

The ***uncompress*** command is used to return the file to its original format:

```
% uncompress logins.*.Z ; ls -als logins.*
     96 -rw-r--r--   1 lindadb  acs  45452 Apr 24 09:13 logins.beauty
    184 -rw-r--r--   1 lindadb  acs  90957 Apr 24 09:13 logins.bottom
    152 -rw-r--r--   1 lindadb  acs  75218 Apr 24 09:13 logins.photon
    168 -rw-r--r--   1 lindadb  acs  85970 Apr 24 09:13 logins.top
```

In addition to the standard Unix ***compress, uncompress, zcat*** utilities there are a set of **GNU** ones freely available. These do an even better job of compression using a more efficient algorithm. The GNU programs to provide similar functions to those above are often installed as ***gzip***, ***gunzip***, and ***zcat***, respectively. Files compressed with gzip are given the endings **.z** or **.gz**. GNU software can be obtained via anonymous ftp from **ftp://prep.ai.mit.edu/pub/gnu**.

8.2.2 tar - archive files

The ***tar*** command combines files into one device or filename for archiving purposes. The ***tar*** command does not compress the files; it merely makes a large quantity of files more manageable.

Syntax

tar [options] [directory file]

Common Options

c	create an archive (begin writting at the start of the file)
t	table of contents list
x	extract from an archive
v	verbose
f	archive file name
b	archive block size

tar will accept its options either with or without a preceding hyphen (**-**). The archive file can be a disk file, a tape device, or standard input/output. The latter are represented by a hyphen.

Examples

Given the files and size indications below:

```
45 logs.beauty
89 logs.bottom
74 logs.photon
84 logs.top
```

tar can combine these into one file, **logfile.tar**:

```
% tar -cf logfile.tar logs.* ; ls -s logfile.tar
    304 logfile.tar
```

Many anonymous FTP archive sites on the Internet store their packages in compressed tar format, so the files will end in **.tar.Z** or **.tar.gz**. To extract the files from these files you would first uncompress them, or use the appropriate zcat command and pipe the output into tar, e.g.:

```
% zcat archive.tar.Z | tar -xvf -
```

where the hyphen at the end of the ***tar*** command indicates that the file is taken from **stdin**.

8.2.3 uuencode/uudecode - encode a file

To encode a binary file into 7-bit ASCII use the ***uuencode*** command. To decode the file back to binary use the ***uudecode*** command. The **uu** in the names comes because they are part of the Unix-to-Unix CoPy (UUCP) set of commands. The ***uuencode*** and ***uudecode*** commands are commonly used when sending binary files through e-mail. In e-mail there's no guarantee that 8-bit binary files will be transferred cleanly. So to ensure delivery you should encode the binary file, either directly, on the command line and then include the encoded file, or indirectly, by letting your MIME mailer program do it for you. In a similar manner, the user decodes the file on the receiving end.

Syntax

uuencode [source_file] pathname_to_uudecode_to [> new_file]

uudecode [-p] encoded_file

Common Options

-p	send output to standard output, rather than to the default file

Examples

The first line of encoded file includes the permission **modes** and **name** that ***uudecode*** will use when decoding the file. The file begins and ends with the **begin** and **end** keywords, respectively, e.g.:

```
begin 555 binary_filename
M?T5,1@$" 0      " (  ! %"W  #0 5"<    T "
M!0 H !4 %   8  T $ -     "@  H   4      P
M -0       !$      !      !     !        %"
M%P !0A<  % $     $ 4(8 -"&       W&  W%    < 0
M    @ !0B  T(@        )@       !P   O=7-R+VQI8B]L9"YS
M;RXQ       ?<  'Y    VP       "O  !VP  )8  &6  !G0
M    %[   U0       %G  !3   ;<  #Q            %Q  !
MEP  :P  !_                       '@  !PP  (P
M   N0  =H       _0       $D   Y  <   #F       /L
M01   $'            $        &     !    P #0A@  4(8
M"            !   0  !E   !@   , T(@  %"(  )@    $
M          0   (   ;@    $    ' -"N  !0K@   /H
M  $   #    ',   !    P #1J   4:@   #8              !
M     !Y    0   , TH   %*   !=X              0
M@     @    # -/X  !3^   "E,              $       (4   !
M             4_>   )0              0       ".    P
;    %0 P   )@              $

end
```

8.2.4 dd - block copy and convert

The *dd* command allows you to copy from raw devices, such as disks and tapes, specifying the input and output block sizes. *dd* was originally known as the disk-to-disk copy program. With *dd* you can also convert between different formats, for example, EBCDIC to ASCII, or swap byte order, etc.

Syntax

dd [*if*=input_device] [*of*=output_device] [operand=value]

Common Options

if=input_device	the input file or device
of=output_device	the output file or device

If the input or output devices are not specified they default to standard input and standard output, respectively.

Operands can include:

ibs=n	input block size (defaults to 512 byte blocks)
obs=n	output block size (defaults to 512 byte blocks)
bs=n	sets both input and output block sizes
files=n	copy **n** input files
skip=n	skip **n** input blocks before starting to copy
count=n	only copy **n** input blocks
conv=value[,value]	where **value** can include:
ascii	convert EBCDIC to ASCII
ebcdic	convert from ASCII to EBCDIC
lcase	convert upper case characters to lower case
ucase	convert lower case characters to upper case
swab	swap every pair of bytes of input data
noerror	don't stop processing on an input error
sync	pad every input block to the size of **ibs**, appending null bytes as needed

Block sizes are specified in bytes and may end in **k**, **b**, or **w** to indicate 1024 (kilo), 512 (block), or 2 (word), respectively.

Examples

To copy files from one tape drive to another:

```
% dd if=/dev/rmt/0 of=/dev/rmt/1
    20+0 records in
    20+0 records out
```

To copy files written on a tape drive on a big endian machine, written with a block size of 20 blocks, to a file on a little endian machine that now has the tape inserted in its drive, we would need to swap pairs of bytes, as in:

```
% dd if=/dev/rmt/0 of=new_file ibs=20b conv=swab
    1072+0 records in
    21440+0 records out
```

Upon completion *dd* reports the number of whole blocks and partial blocks for both the input and output files.

8.2.5 od - octal dump of a file

od dumps a file to stdout in different formats, including octal, decimal, floating point, hex, and character format.

Syntax

od [options] file

Common Options

-b	octal dump
-d|-D	decimal (-d) or long decimal (-D) dump
-s|-S	signed decimal (-s) and signed long decimal (-S) dump
-f|-F	floating point (-f) or long (double) floating point (-F) dump
-x|-X	hex (-x) or long hex (-X) dump
-c|-C	character (single byte) or long character dump (single or multi-byte characters, as determined by locale settings) dump
-v	verbose mode

Examples

To look at the actual contents of the following file, a list of P. G. Wodehouse's Lord Emsworth novels.

Something Fresh [1915]	Uncle Dynamite [1948]
Leave it to Psmith [1923]	Pigs Have Wings [1952]
Summer Lightning [1929]	Cocktail Time [1958]
Heavy Weather [1933]	Service with a Smile [1961]
Blandings Castle and Elsewhere [1935]	Galahad at Blandings [1965]
Uncle Fred in the Springtime [1939]	A Pelican at Blandings [1969]
Full Moon [1947]	Sunset at Blandings [1977]

we could do:

```
% od -c wodehouse
    0000000   S   o   m   e   t   h   i   n   g       F   r   e   s   h
    0000020   [   1   9   1   5   ]  \t   U   n   c   l   e       D   y   n
    0000040   a   m   i   t   e       [   1   9   4   8   ]  \n   L   e   a
    0000060   v   e       i   t       t   o       P   s   m   i   t   h
    0000100   [   1   9   2   3   ]  \t   P   i   g   s       H   a   v   e
    0000120       W   i   n   g   s       [   1   9   5   2   ]  \n   S   u
    0000140   m   m   e   r       L   i   g   h   t   n   i   n   g       [
    0000160   1   9   2   9   ]  \t   C   o   c   k   t   a   i   l       T
    0000200   i   m   e       [   1   9   5   8   ]  \n   H   e   a   v   y
    0000220       W   e   a   t   h   e   r       [   1   9   3   3   ]  \t
    0000240   S   e   r   v   i   c   e       w   i   t   h       a       S
    0000260   m   i   l   e       [   1   9   6   1   ]  \n   B   l   a   n
    0000300   d   i   n   g   s       C   a   s   t   l   e       a   n   d
    0000320       E   l   s   e   w   h   e   r   e       [   1   9   3   5
    0000340   ]  \t   G   a   l   a   h   a   d       a   t       B   l   a
    0000360   n   d   i   n   g   s       [   1   9   6   5   ]  \n   U   n
    0000400   c   l   e       F   r   e   d       i   n       t   h   e
    0000420   S   p   r   i   n   g   t   i   m   e       [   1   9   3   9
    0000440   ]  \t   A       P   e   l   i   c   a   n       a   t       B
    0000460   l   a   n   d   i   n   g   s       [   1   9   6   9   ]  \n
    0000500   F   u   l   l       M   o   o   n       [   1   9   4   7   ]
    0000520  \t   S   u   n   s   e   t       a   t       B   l   a   n   d
    0000540   i   n   g   s       [   1   9   7   7   ]  \n
    0000554
```

8.3 Remote Connections

TABLE 8.3 **Remote Connection Commands**

Command/Syntax	What it will do
finger [options] *user[@hostname]*	report information about users on local and remote machines
ftp [options] *host*	transfer file(s) using file transfer protocol
rcp [options] *hostname*	remotely copy files from this machine to another machine
rlogin [options] *hostname*	login remotely to another machine
rsh [options] *hostname*	remote shell to run on another machine
telnet [host [port]]	communicate with another host using telnet protocol

8.3.1 TELNET and FTP - remote login and file transfer protocols

TELNET and **FTP** are Application Level Internet protocols. The TELNET and FTP protocol specifications have been implemented by many different sources, including The National Center for Supercomputer Applications (NCSA), and many other public domain and shareware sources.

The programs implementing the **TELNET** protocol are usually called ***telnet***, but not always. Some notable exceptions are ***tn3270***, ***WinQVT***, and ***QWS3270***, which are also TELNET protocol implementations. TELNET is used for remote login to other computers on the Internet.

The programs implementing the **FTP** protocol are usually called ***ftp***, but there are exceptions to that too. A program called ***Fetch***, distributed by Dartmouth College, ***WS_FTP***, written and distributed by John Junod, and ***Ftptool***, written by a Mike Sullivan, are FTP protocol implementations with graphic user interfaces. There's an enhanced FTP version, ***ncftp***, that allows additional features, written by Mike Gleason. Also, FTP protocol implementations are often included in TELNET implementation programs, such as the ones distributed by NCSA. FTP is used for transferring files between computers on the Internet.

rlogin is a remote login service that was at one time exclusive to Berkeley 4.3 BSD UNIX. Essentially, it offers the same functionality as ***telnet***, except that it passes to the remote computer information about the user's login environment. Machines can be configured to allow connections from trusted hosts without prompting for the users' passwords. A more secure version of this protocol is the Secure SHell, **SSH**, software written by Tatu Ylonen and available via ftp://ftp.net.ohio-state.edu/pub/security/ssh.

From a Unix prompt, these programs are invoked by typing the command (program name) and the (Internet) name of the remote machine to which to connect. You can also specify various options, as allowed, for these commands.

Syntax

telnet [options] [remote_host [port_number]]

tn3270 [options] [remote_host [port_number]]

ftp [options] [remote_host]

Common Options

ftp	telnet	Action
-d		set debugging mode on
	-d	same as above (SVR4 only)
-i		turn off interactive prompting
-n		don't attempt auto-login on connection
-v		verbose mode on
	-l user	connect with username, **user**, on the remote host (SVR4 only)
	-8	8-bit data path (SVR4 only)

telnet and ***tn3270*** allow you the option of specifying a port number to connect to on the remote host. For both commands it defaults to port number 23, the telnet port. Other ports are used for debugging of network services and for specialized resources.

Examples

```
telnet oscar.us.ohio-state.edu
tn3270 ohstmvsa.acs.ohio-state.edu
ftp magnus.acs.ohio-state.edu
```

The remote machine will query you for your login identification and your password. Machines set up as archives for software or information distribution often allow anonymous ftp connections. You ***ftp*** to the remote machine and login as **anonymous** (the login **ftp** is equivalent on many machines), that is, when asked for your "login" you would type **anonymous**.

Once you have successfully connected to a remote computer with ***telnet*** and ***rlogin*** (and assuming terminal emulation is appropriate) you will be able to use the machine as you always do.

Once you have successfully connected to a remote computer with *ftp*, you will be able to transfer a file "up" to that computer with the ***put*** command, or "down" from that computer with the ***get*** command. The syntax is as follows:

```
put  local-file-name  remote-file-name
get  local-file-name  remote-file-name
```

Other commands are available in ***ftp*** as well, depending on the specific "local" and "remote" FTP implementations. The ***help*** command will display a list of available commands. The ***help*** command will also display the purpose of a specific command. Examples of valid commands are shown below:

help	display list of available commands
help mget	display the purpose of the mget command ("get multiple files")
pwd	present working directory
ls or ***dir***	directory list
cd	change directory
lcd	local change directory
open	specify the machine you wish to connect with
user	specify your login id (in cases where you are not prompted)
quit	quit out of the FTP program

8.3.2 finger - get information about users

finger displays the **.plan** file of a specific user, or reports who is logged into a specific machine. The user must allow general read permission on the **.plan** file.

Syntax

finger [options] [user[@hostname]]

Common Options

-l	force long output format
-m	match username only, not first or last names
-s	force short output format

Examples

```
brigadier: condron [77]> finger workshop@nyssa
    This is a sample .plan file for the nyssa id, workshop.
    This id is being used this week by Frank Fiamingo, Linda
    DeBula, and Linda Condron, while we teach a pilot version
    of the new Unix workshop we developed for UTS.

    Hope yer learnin' somethin'.
    Frank, Linda, & Linda

brigadier: condron [77]> finger
    Login      Name                TTY Idle   When      Where
    condron  Linda S Condron       p0       Sun 18:13  lcondron-mac.acs
    frank    Frank G. Fiamingo     p1       Mon 16:19  nyssa
```

8.3.3 Remote commands

A number of Unix machines can be connected together to form a local area network. When this is the case, it often happens that a user of one machine has valid login access to several of the other machines in the local network. There are Unix commands available to such users which provide convenience in carrying out certain common operations. Because these commands focus on communications with remote hosts in the local network, the command names begin with the letter "**r**": ***rlogin***, ***rsh***, and ***rcp***. The remote access capability of these commands is supported (optionally) by the dotfile, **~/.rhosts**, for individual users and by the system-wide file **/etc/hosts.equiv**. For security reasons these may be restricted on some hosts.

The ***rlogin*** command allows remote login access to another host in the local network. ***rlogin*** passes information about the local environment, including the value of the **TERM** environment variable, to the remote host.

The ***rsh*** command provides the ability to invoke a Unix shell on a remote host in the local network for the purpose of executing a shell command there. This capability is similar to the "shell escape" function commonly available from within such Unix software systems as editors and email.

The ***rcp*** command provides the ability to copy files from the local host to a remote host in the local network.

Syntax

rlogin [-l username] remote_host

rsh [-l username] remote_host [command]

rcp [[user1]@host1:]original_filename [[user2]@host2:]new_filename

where the parts in brackets ([]) are optional. ***rcp*** does not prompt for passwords, so you must have permission to execute remote commands on the specified machines as the selected user on each machine.

Common Options

-l username	connect as the user, **username**, on the remote host (***rlogin*** & ***rsh***)

The **.rhosts** file, if it exists in the user's home directory on the remote host, permits ***rlogin***, ***rsh***, or ***rcp*** access to that remote host without prompting for a password for that account. The **.rhosts** file contains an entry for each remote host and username from which the owner of the **.rhosts** file may wish to connect. Each entry in the **.rhosts** file is of the form:

```
remote_host  remote_user
```

where listing the remote_user is optional. For instance, if Heather Jones wants to be able to connect to machine1 (where her username is heather) from machine2 (where her username is jones), or from machine 3 (where her username is heather, the same as for machine1), she could create a **.rhosts** file in her home directory on machine1. The contents of this file could be:

```
machine2 jones
machine3
```

--or--

```
machine2 jones
machine3 heather
```

On a system-wide basis the file **/etc/hosts.equiv** serves the same purpose for all users, except the super-user. Such a file with the contents:

```
remote_machine
```

allows any user from remote_machine to remote connect to this machine without a password, as the same username on this machine.

An **/etc/hosts.equiv** file with the contents:

```
remote_machine   remote_user
```

allows remote_user, on remote_machine, to remote connect to this machine as **any** local user, except the super-user.

/etc/hosts.equiv and **~/.rhosts** files should be used with caution.

The Secure SHell (**SSH**) versions of the ***rcp***, ***rsh***, and ***rlogin*** programs are freely available and provide much greater security.

CHAPTER 9 Shell Programming

9.1 Shell Scripts

You can write shell programs by creating scripts containing a series of shell commands. The first line of the script should start with #! which indicates to the kernel that the script is directly executable. You immediately follow this with the name of the shell, or program (spaces are allowed), to execute, using the full path name. Generally you can count on having up to 32 characters, possibly more on some systems, and can include one option. So to set up a Bourne shell script the first line would be:

```
#! /bin/sh
```

or for the C shell:

```
#! /bin/csh -f
```

where the "**-f**" option indicates that it should not read your **.cshrc**. Any blanks following the magic symbols, **#!**, are optional.

You also need to specify that the script is executable by setting the proper bits on the file with ***chmod***, e.g.:

```
% chmod +x shell_script
```

Within the scripts # indicates a comment from that point until the end of the line, with **#!** being a special case if found as the first characters of the file.

9.2 Setting Parameter Values

Parameter values, e.g. **param**, are assigned as:

Bourne shell	**C shell**
param=value	set param = value

where **value** is any valid string, and can be enclosed within quotations, either single (**'value**) or double (**"value"**), to allow spaces within the string value. When enclosed with backquotes (**'value'**) the string is first evaluated by the shell and the result is substituted. This is often used to run a command, substituting the command output for **value**, e.g.:

```
$ day=`date +%a`
$ echo $day
    Wed
```

After the parameter values has been assigned the current value of the parameter is accessed using the **$param**, or **${param}**, notation.

9.3 Quoting

We quote strings to control the way the shell interprets any parameters or variables within the string. We can use single (') and double (") quotes around strings. Double quotes define the string, but allow variable substitution. Single quotes define the string and prevent variable substitution. A backslash (\) before a character is said to escape it, meaning that the system should take the character literally, without assigning any special meaning to it. These quoting techniques can be used to separate a variable from a fixed string. As an example lets use the variable, **var**, that has been assigned the value **bat**, and the constant string, **man**. If I wanted to combine these to get the result "batman" I might try:

```
$varman
```

but this doesn't work, because the shell will be trying to evaluate a variable called **varman**, which doesn't exist. To get the desired result we need to separate it by quoting, or by isolating the variable with curly braces ({}), as in:

"$var"man	- quote the variable
$var""man	- separate the parameters
$var"man"	- quote the constant
$var''man	- separate the parameters
$var'man'	- quote the constant
$var\man	- separate the parameters
${var}man	- isolate the variable

These all work because ", ', \, {, and } are not valid characters in a variable name.

We could not use either of

```
'$var'man
\$varman
```

because it would prevent the variable substitution from taking place.

When using the curly braces they should surround the variable only, and not include the $, otherwise, they will be included as part of the resulting string, e.g.:

```
% echo {$var}man
    {bat}man
```

9.4 Variables

There are a number of variables automatically set by the shell when it starts. These allow you to reference arguments on the command line.

These **shell variables** are:

TABLE 9.1 Shell Variables

Variable	Usage	sh	csh
$#	number of arguments on the command line	x	
$-	options supplied to the shell	x	
$?	exit value of the last command executed	x	
$$	process number of the current process	x	x
$!	process number of the last command done in background	x	
$n	argument on the command line, where n is from 1 through 9, reading left to right	x	x
$0	the name of the current shell or program	x	x
$*	all arguments on the command line ("$1 $2 ... $9")	x	x
$@	all arguments on the command line, each separately quoted ("$1" "$2" ... "$9")	x	
$argv[n]	selects the **n**th word from the input list		x
${argv[n]}	same as above		x
$#argv	report the number of words in the input list		x

We can illustrate these with some simple scripts. First for the Bourne shell the script will be:

```
#!/bin/sh
echo "$#:" $#
echo '$#:' $#
echo '$-:' $-
echo '$?:' $?
echo '$$:' $$
echo '$!:' $!
echo '$3:' $3
echo '$0:' $0
echo '$*:' $*
echo '$@:' $@
```

When executed with some arguments it displays the values for the shell variables, e.g.:

```
$ ./variables.sh one two three four five
    5: 5
    $#: 5
    $-:
    $?: 0
    $$: 12417
    $!:
    $3: three
    $0: ./variables.sh
    $*: one two three four five
    $@: one two three four five
```

As you can see, we needed to use single quotes to prevent the shell from assigning special meaning to $. The double quotes, as in the first echo statement, allowed substitution to take place.

Similarly, for the C shell variables we illustrate variable substitution with the script:

```
#!/bin/csh -f
echo '$$:' $$
echo '$3:' $3
echo '$0:' $0
echo '$*:' $*
echo '$argv[2]:' $argv[2]
echo '${argv[4]}:' ${argv[4]}
echo '$#argv:' $#argv
```

which when executed with some arguments displays the following:

```
% ./variables.csh one two three four five
    $$: 12419
    $3: three
    $0: ./variables.csh
    $*: one two three four five
    $argv[2]: two
    ${argv[4]}: four
    $#argv: 5
```

9.5 Parameter Substitution

You can reference parameters abstractly and substitute values for them based on conditional settings using the operators defined below. Again we will use the curly braces ({}) to isolate the variable and its operators.

$parameter	substitute the value of **parameter** for this string
${parameter}	same as above. The brackets are helpful if there's no separation between this parameter and a neighboring string.
$parameter=	sets **parameter** to **null**.
${parameter-default}	if **parameter** is not set, then use **default** as the value here. The parameter is not reset.
${parameter=default}	if **parameter** is not set, then set it to **default** and use the new value
${parameter+newval)	if **parameter** is set, then use **newval**, otherwise use nothing here. The parameter is not reset.
${parameter?message}	if **parameter** is not set, then display **message**. If **parameter** is set, then use its current value.

There are no spaces in the above operators. If a colon (**:**) is inserted before the **-**, **=**, **+**, or **?** then a test if first performed to see if the parameter has a **non-null** setting.

The C shell has a few additional ways of substituting parameters:

$list[n]	selects the **n**th word from list
${list[n]}	same as above
$#list	report the number of words in list
$?parameter	return 1 if parameter is set, 0 otherwise
${?parameter}	same as above
$<	read a line from stdin

The C shell also defines the array, **$argv[n]** to contain the **n** arguments on the command line and **$#argv** to be the number of arguments, as noted in Table 9.1.

To illustrate some of these features we'll use the test script below.

```
#!/bin/sh
    param0=$0
    test -n "$1" && param1=$1
    test -n "$2" && param2=$2
    test -n "$3" && param3=$3
    echo 0: $param0
    echo "1: ${param1-1}: \c" ;echo $param1
    echo "2: ${param2=2}: \c" ;echo $param2
    echo "3: ${param3+3}: \c" ;echo $param3
```

In the script we first test to see if the variable exists, if so we set a parameter to its value. Below this we report the values, allowing substitution.

In the first run through the script we won't provide any arguments:

```
$ ./parameter.sh
    0: ./parameter.sh               # always finds $0
    1: 1:                           # substitute 1, but don't assign this value
    2: 2: 2                         # substitute 2 and assign this value
    3: :                            # don't substitute
```

In the second run through the script we'll provide the arguments:

```
$ ./parameter one two three
    0: ./parameter.sh               # always finds $0
    1: one: one                     # don't substitute, it already has a value
    2: two: two                     # don't substitute, it already has a value
    3: 3: three                     # substitute 3, but don't assign this value
```

9.6 Here Document

A **here document** is a form of quoting that allows shell variables to be substituted. It's a special form of redirection that starts with **<<WORD** and ends with **WORD** as the only contents of a line. In the Bourne shell you can prevent shell substitution by escaping **WORD** by putting a \ in front of it on the redirection line, i.e. **<<\WORD**, but not on the ending line. To have the same effect the C shell expects the \ in front of **WORD** at both locations.

The following scripts illustrate this,

for the **Bourne shell**:

```
#!/bin/sh
does=does
not=""
cat << EOF
This here document
$does $not
do variable substitution
EOF
cat << \EOF
This here document
$does $not
do variable substitution
EOF
```

and for the **C shell**:

```
#!/bin/csh -f
set does = does
set not = ""
cat << EOF
This here document
$does $not
do variable substitution
EOF
cat << \EOF
This here document
$does $not
do variable substitution
\EOF
```

Both produce the output:

```
This here document
does
do variable substitution
This here document
$does $not
do variable substitution
```

In the top part of the example the shell variables **$does** and **$not** are substituted. In the bottom part they are treated as simple text strings without substitution.

9.7 Interactive Input

Shell scripts will accept interactive input to set parameters within the script.

9.7.1 Sh

Sh uses the built-in command, **read**, to read in a line, e.g.:

```
read param
```

We can illustrate this with the simple script:

```
#!/bin/sh
echo "Input a phrase \c"            # This is /bin/echo which requires "\c" to prevent <newline>
read param
echo param=$param
```

When we run this script it prompts for input and then echoes the results:

```
$ ./read.sh
    Input a phrase hello frank      # I type in hello frank <return>
    param=hello frank
```

9.7.2 Csh

Csh uses the **$<** symbol to read a line from stdin, e.g.:

```
set param = $<
```

The spaces around the equal sign are important. The following script illustrates how to use this.

```
#!/bin/csh -f
echo -n "Input a phrase "           # This built-in echo requires -n to prevent <newline>
set param = $<
echo param=$param
```

Again, it prompts for input and echoes the results:

```
% ./read.csh
    Input a phrase hello frank      # I type in hello frank <return>
    param=hello frank
```

9.8 Functions

The Bourne shell has functions. These are somewhat similar to aliases in the C shell, but allow you more flexibility. A function has the form:

```
fcn () { command; }
```

where the space after {, and the semicolon (;) are both required; the latter can be dispensed with if a <newline> precedes the }. Additional spaces and <newline>'s are allowed. We saw a few examples of this in the sample **.profile** in an earlier chapter, where we had functions for **ls** and **ll**:

```
ls() { /bin/ls -sbF "$@";}
ll() { ls -al "$@";}
```

The first one redefines ***ls*** so that the options **-sbF** are always supplied to the standard ***/bin/ls*** command, and acts on the supplied input, **"$@"**. The second one takes the current value for ***ls*** (the previous function) and tacks on the **-al** options.

Functions are very useful in shell scripts. The following is a simplified version of one I use to automatically backup up system partitions to tape.

```
#!/bin/sh
#       Cron script to do a complete backup of the system
HOST=`/bin/uname -n`
admin=frank
Mt=/bin/mt
Dump=/usr/sbin/ufsdump
Mail=/bin/mailx
device=/dev/rmt/0n
Rewind="$Mt -f $device rewind"
Offline="$Mt -f $device rewoffl"
# Failure - exit
failure () {
        $Mail -s "Backup Failure - $HOST" $admin << EOF_failure
$HOST
Cron backup script failed. Apparently there was no tape in the device.

EOF_failure
        exit 1
        }

# Dump failure - exit
dumpfail () {
        $Mail -s "Backup Failure - $HOST" $admin << EOF_dumpfail
$HOST
Cron backup script failed.  Initial tape access was okay, but dump failed.
EOF_dumpfail
        exit 1
        }
# Success
success () {
        $Mail -s "Backup completed successfully - $HOST" $admin << EOF_success
$HOST
Cron backup script was apparently successful.  The /etc/dumpdates file is:
`/bin/cat /etc/dumpdates`
EOF_success
        }
# Confirm that the tape is in the device
$Rewind || failure
$Dump 0uf $device / || dumpfail
$Dump 0uf $device /usr || dumpfail
$Dump 0uf $device /home || dumpfail
$Dump 0uf $device /var || dumpfail
($Dump 0uf $device /var/spool/mail || dumpfail) && success
$Offline
```

This script illustrates a number of topics that we've looked at in this document. It starts by setting various parameter values. **HOST** is set from the output of a command, **admin** is the administrator of the system, **Mt**, **Dump**, and **Mail** are program names, device is the special device file used to access the tape drive, **Rewind** and **Offline** contain the commands to rewind and off-load the tape drive, respectively, using the previously referenced Mt and the necessary options. There are three functions defined: **failure**, **dumpfail**, and **success**. The functions in this script all use a **here document** to form the contents of the function. We also introduce the logical **OR** (||) and **AND** (**&&**) operators here; each is position between a pair of commands. For the **OR** operator, the second command will be run only if the first command does not complete successfully. For the **AND** operator, the second command will be run only if the first command does complete successfully.

The main purpose of the script is done with the Dump commands, i.e. backup the specified file systems. First an attempt is made to rewind the tape. Should this fail, || **failure**, the **failure** function is run and we exit the program. If it succeeds we proceed with the backup of each partition in turn, each time checking for successful completion (|| **dumpfail**). Should it not complete successfully we run the **dumpfail** subroutine and then exit. If the last backup succeeds we proceed with the **success** function (**(...) && success**). Lastly, we rewind the tape and take it offline so that no other user can accidently write over our backup tape.

9.9 Control Commands

9.9.1 Conditional if

The **conditional if** statement is available in both shells, but has a different syntax in each.

9.9.1.1 Sh

```
if condition1
then
        command list if condition1 is true
[elif condition2
        then command list if condition2 is true]
[else
        command list if condition1 is false]
fi
```

The conditions to be tested for are usually done with the ***test***, or *[]* command (see Section 8.9.6). The **if** and **then** must be separated, either with a <newline> or a semicolon (;).

```
#!/bin/sh
if [ $# -ge 2 ]
then
        echo $2
elif [ $# -eq 1 ]; then
        echo $1
else
        echo No input
fi
```

There are required spaces in the format of the conditional test, one after [and one before]. This script should respond differently depending upon whether there are zero, one or more arguments on the command line. First with no arguments:

```
$ ./if.sh
    No input
```

Now with one argument:

```
$ ./if.sh one
    one
```

And now with two arguments:

```
$ ./if.sh one two
    two
```

9.9.1.2 Csh

```
if (condition) command
        -or-
if (condition1) then
        command list if condition1 is true
[else if (condition2) then
        command list if condition2 is true]
[else
        command list if condition1 is false]
endif
```

The **if** and **then** must be on the same line.

```
#!/bin/csh -f
if ( $#argv >= 2 ) then
        echo $2
else if ( $#argv == 1 ) then
        echo $1
else
        echo No input
endif
```

Again, this script should respond differently depending upon whether I have zero, one or more arguments on the command line. First with no arguments:

```
% ./if.csh
    No input
```

Now with one argument:

```
% ./if.csh one
    one
```

And now with two arguments:

```
% ./if.csh one two
    two
```

9.9.2 Conditional switch and case

To choose between a set of string values for a parameter use ***case*** in the Bourne shell and ***switch*** in the C shell.

9.9.2.1 Sh

```
case parameter in
        pattern1[|pattern1a]) command list1;;
        pattern2) command list2
                command list2a;;
        pattern3) command list3;;
        *) ;;
esac
```

You can use any valid filename meta-characters within the patterns to be matched. The **;;** ends each choice and can be on the same line, or following a <newline>, as the last command for the choice. Additional alternative patterns to be selected for a particular case are separated by the vertical bar, **|**, as in the first pattern line in the example above. The wildcard symbols,: **?** to indicate any one character and ***** to match any number of characters, can be used either alone or adjacent to fixed strings.

This simple example illustrates how to use the conditional case statement.

```
#!/bin/sh
case $1 in
        aa|ab)  echo A
                ;;
        b?)     echo "B \c"
                echo $1;;
        c*)     echo C;;
        *)      echo D;;
esac
```

So when running the script with the arguments on the left, it will respond as on the right:

aa	A
ab	A
ac	D
bb	B bb
bbb	D
c	C
cc	C
fff	D

9.9.2.2 Csh

```
switch (parameter)
case pattern1:
        command list1
        [breaksw]
case pattern2:
        command list2
        [breaksw]
default:
        command list for default behavior
        [breaksw]
endsw
```

breaksw is optional and can be used to break out of the switch after a match to the string value of the parameter is made. **Switch** doesn't accept "|" in the pattern list, but it will allow you to string several **case** statements together to provide a similar result. The following C shell script has the same behavior as the Bourne shell **case** example above.

```
#!/bin/csh -f
switch ($1)
        case aa:
        case ab:
                echo A
                breaksw
        case    b?:
                echo -n "B "
                echo $1
                breaksw
        case    c*:
                echo C
                breaksw
        default:
                echo D
endsw
```

9.9.3 for and foreach

One way to loop through a list of string values is with the *for* and *foreach* commands.

9.9.3.1 Sh

```
for variable [in list_of_values]
do
        command list
done
```

The **list_of_values** is optional, with **$@** assumed if nothing is specified. Each value in this list is sequentially substituted for **variable** until the list is emptied. Wildcards can be used and are applied to file names in the current directory. Below we illustrate the for loop in copying all files ending in **.old** to similar names ending in **.new**. In these examples the ***basename*** utility extracts the base part of the name so that we can exchange the endings.

```
#!/bin/sh
for file in *.old
do
        newf=`basename $file .old`
        cp $file $newf.new
done
```

9.9.3.2 Csh

```
foreach variable (list_of_values)
        command list
end
```

The equivalent C shell script to copy all files ending in **.old** to **.new** is:

```
#!/bin/csh -f
foreach file (*.old)
        set newf = `basename $file .old`
        cp $file $newf.new
end
```

9.9.4 while

The *while* commands let you loop as long as the condition is true.

9.9.4.1 Sh

```
while condition
do
        command list
        [break]
        [continue]
done
```

A simple script to illustrate a **while** loop is:

```
#!/bin/sh
while [ $# -gt 0 ]
do
        echo $1
        shift
done
```

This script takes the list of arguments, echoes the first one, then shifts the list to the left, losing the original first entry. It loops through until it has shifted all the arguments off the argument list.

```
$ ./while.sh one two three
one
two
three
```

9.9.4.2 Csh

```
while (condition)
        command list
        [break]
        [continue]
end
```

If you want the condition to always be true specify 1 within the conditional test.

A C shell script equivalent to the one above is:

```
#!/bin/csh -f
while ($#argv != 0 )
        echo $argv[1]
        shift
end
```

9.9.5 until

This looping feature is only allowed in the Bourne shell.

```
until condition
do
        command list while condition is false
done
```

The condition is tested at the start of each loop and the loop is terminated when the condition is true. A script equivalent to the **while** examples above is:

```
#!/bin/sh
until [ $# -le 0 ]
do
        echo $1
        shift
done
```

Notice, though, that here we're testing for *less than or equal*, rather than *greater than or equal*, because the **until** loop is looking for a **false** condition.

Both the **until** and **while** loops are only executed if the condition is satisfied. The condition is evaluated before the commands are executed.

9.9.6 test

Conditional statements are evaluated for **true** or **false** values. This is done with the ***test***, or its equivalent, the *[]* operators. It the condition evaluates to true, a zero (**TRUE**) exit status is set, otherwise a non-zero (**FALSE**) exit status is set. If there are no arguments a non-zero exit status is set. The operators used by the Bourne shell conditional statements are given below.

For **filenames** the options to ***test*** are given with the syntax:

-option filename

The options available for the ***test*** operator for **files** include:

-r	true if it exists and is readable
-w	true if it exists and is writable
-x	true if it exists and is executable
-f	true if it exists and is a regular file (or for csh, exists and is not a directory)
-d	true if it exists and is a directory
-h or **-L**	true if it exists and is a symbolic link
-c	true if it exists and is a character special file (i.e. the special device is accessed one character at a time)
-b	true if it exists and is a block special file (i.e. the device is accessed in blocks of data)
-p	true if it exists and is a named pipe (fifo)
-u	true if it exists and is setuid (i.e. has the set-user-id bit set, s or S in the third bit)
-g	true if it exists and is setgid (i.e. has the set-group-id bit set, s or S in the sixth bit)
-k	true if it exists and the sticky bit is set (a t in bit 9)
-s	true if it exists and is greater than zero in size

There is a test for **file descriptors**:

-t [file_descriptor]	true if the open file descriptor (default is 1, stdin) is associated with a terminal

There are tests for **strings**:

-z string	true if the string length is zero
-n string	true if the string length is non-zero
string1 = string2	true if string1 is identical to string2
string1 != string2	true if string1 is non identical to string2
string	true if string is not NULL

There are **integer comparisons**:

n1 -eq n2	true if integers n1 and n2 are equal
n1 -ne n2	true if integers n1 and n2 are not equal
n1 -gt n2	true if integer n1 is greater than integer n2
n1 -ge n2	true if integer n1 is greater than or equal to integer n2
n1 -lt n2	true if integer n1 is less than integer n2
n1 -le n2	true if integer n1 is less than or equal to integer n2

The following **logical operators** are also available:

!	negation (unary)
-a	and (binary)
-o	or (binary)
()	expressions within the () are grouped together. You may need to quote the () to prevent the shell from interpreting them.

9.9.7 C Shell Logical and Relational Operators

The C shell has its own set of built-in logical and relational expression operators. In descending order of precedence they are:

(...)	group expressions with ()
~	inversion (one's complement)
!	logical negation
*, /, %	multiply, divide, modulus
+, -	add, subtract
<<, >>	bitwise shift left, bitwise shift right
<=	less than or equal
>=	greater than or equal
<	less than
>	greater than
==	equal
!=	not equal
=~	match a string
!~	don't match the string
&	bitwise AND
^	bitwise XOR (exclusive or)
|	bitwise OR
&&	logical AND
||	logical OR
{command}	true (1) if command terminates with a zero exit status, false (0) otherwise.

The C shell also allows file type and permission inquiries with the operators:

-r	return true (1) if it exists and is readable, otherwise return false (0)
-w	true if it exists and is writable
-x	true if it exists and is executable
-f	true if it exists and is a regular file (or for csh, exists and is not a directory)
-d	true if it exists and is a directory
-e	true if the file exists
-o	true if the user owns the file
-z	true if the file has zero length (empty)

CHAPTER 10 Editors

There are numerous text processing utilities available with Unix, as is noted throughout this document (e.g., ***ed***, ***ex***, ***sed***, ***awk***, the ***grep*** family, and the ***roff*** family). Among the editors, the standard "visual" (or fullscreen) editor on Unix is ***vi***. It comprises a super-set, so to speak, of ***ed*** and ***ex*** (the Unix line editors) capabilities.

Vi is a modal editor. This means that it has specific modes that allow text insertion, text deletion, and command entering. You leave the insert mode by typing the **<escape>** key. This brings you back to command mode. The line editor, ***ex***, is incorporated within ***vi***. You can switch back and forth between full-screen and line mode as desired. In **vi** mode type **Q** to go to **ex** mode. In **ex** mode at the **:** prompt type **vi** to return to **vi** mode. There is also a read-only mode of ***vi***, which you can invoke as ***view***.

Another editor that is common on Unix systems, especially in college and university environments, is ***emacs*** (which stands for "editing macros"). While ***vi*** usually comes with the Unix operating system, ***emacs*** usually does not. It is distributed by The Free Software Foundation. It is arguably the most powerful editor available for Unix. It is also a very large software system, and is a heavy user of computer system resources.

The Free Software Foundation and the GNU Project (of which ***emacs*** is a part) were founded by Richard Stallman and his associates, who believe (as stated in the GNU Manifesto) that sharing software is the "fundamental act of friendship among programmers." Their General Public License guarantees your rights to use, modify, and distribute emacs (including its source code), and was specifically designed to prevent anyone from hoarding or turning a financial profit from ***emacs*** or any software obtained through the Free Software Foundation. Most of their software, including ***emacs***, is available at: ftp://prep.ai.mit.edu/pub/gnu.

Both ***vi*** and ***emacs*** allow you to create start-up files that you can populate with macros to control settings and functions in the editors.

10.1 Configuring Your vi Session

To configure the ***vi*** environment certain options can be set with the line editor command **:set** during a ***vi*** editing session. Alternatively, frequently used options can be **set** automatically when ***vi*** is invoked, by use of the **.exrc** file. This file can also contain macros to map keystrokes into functions using the **map** function. Within ***vi*** these macros can be defined with the **:map** command. Control characters can be inserted by first typing **<control>-V** (^V), then the desired control character. The options available in ***vi*** include, but are not limited to, the following. Some options are not available on every Unix system.

:set all	display all option settings
:set ignorecase	ignore the case of a character in a search
:set list	display tabs and carriage returns
:set nolist	turn off list option
:set number	display line numbers
:set nonumber	turn off line numbers
:set showmode	display indication that insert mode is on
:set noshowmode	turn off showmode option
:set wrapmargin=n	turn on word-wrap n spaces from the right margin
:set wrapmargin=0	turn off wrapmargin option
:set warn	display "No write since last change"
:set nowarn	turn off "write" warning

The following is a sample **.exrc** file:

```
set wrapmargin=10
set number
set list
set warn
set ignorecase
map K {!}fmt -80      # reformat this paragraph, {!}, using fmt to 80 characters per line
map ^Z :!spell        # invoke spell, :!, to check a word spelling (return to vi with ^D)
```

10.2 Configuring Your emacs Session

Configuring the ***emacs*** environment amounts to making calls to LISP functions. ***Emacs*** is infinitely customizable by means of ***emacs*** variables and built-in functions and by using Emacs LISP programming. Settings can be specified from the minibuffer (or command line) during an ***emacs*** session. Alternatively, frequently used settings can be established automatically when ***emacs*** is invoked, by use of a **.emacs** file. Though a discussion of Emacs LISP is beyond the scope of this document, a few examples of common ***emacs*** configurations follow.

To set or toggle ***emacs*** variables, or to use ***emacs*** built-in functions, use the **<escape> key** ("**Meta**" is how ***emacs*** refers to it), followed by the letter **x**, then by the variable or function and its arguments.

M-x what-line	what line is the cursor on?
M-x auto-fill-mode	turn on word-wrap
M-x auto-fill-mode	turn off word-wrap
M-x set-variable<return>	
fill-column<return>	set line-length to
45	45 characters
M-x set-variable<return>	
auto-save-interval<return>	save the file automatically after every
300	300 keystrokes
M-x goto-line<return>16	move the cursor to line 16
M-x help-for-help	invoke emacs help when C-h has been bound to the backspace key

The following is a sample **.emacs** file:

```
(message "Loading ~/.emacs...")
; Comments begin with semi-colons and continue to the end of the line.
(setq text-mode-hook 'turn-on-auto-fill)     ;turn on word-wrap
(setq fill-column 45)                        ;line-length=45 chars
(setq auto-save-interval 300)                ;save after every 300 keystrokes
; Bind (or map) the rubout (control-h) function to the backspace key
(global-set-key "\C-h" 'backward-delete-char-untabify)
; Bind the emacs help function to the keystroke sequence "C-x ?".
(global-set-key "\C-x?" 'help-for-help)
; To jump to line 16, type M-#<return>16
(global-set-key "\M-#" 'goto-line)
; To find out what line you are on, type M-n
(global-set-key "\M-n" 'what-line)
(message "~/.emacs loaded.")
(message "")
```

10.3 vi Quick Reference Guide

All commands in *vi* are preceded by pressing the escape key. Each time a different command is to be entered, the escape key needs to be used. Except where indicated, *vi* is case sensitive.

Cursor Movement Commands:

(n) indicates a number, and is optional

(n)**h**	left (n) space(s)
(n)**j**	down (n) space(s)
(n)**k**	up (n) space(s)
(n)**l**	right (n) space(s)

(The arrow keys usually work also)

^F	forward one screen
^B	back one screen
^D	down half screen
^U	up half screen

(^ indicates control key; case does not matter)

H	beginning of top line of screen
M	beginning of middle line of screen
L	beginning of last line of screen
G	beginning of last line of file
(n)**G**	move to beginning of line (n)
0	(zero) beginning of line
$	end of line
(n)**w**	forward (n) word(s)
(n)**b**	back (n) word(s)
e	end of word

Inserting Text:

i	insert text before the cursor
a	append text after the cursor (does not overwrite other text)
I	insert text at the beginning of the line
A	append text to the end of the line
r	replace the character under the cursor with the next character typed
R	Overwrite characters until the end of the line (or until escape is pressed to change command)
o	(alpha o) open new line after the current line to type text
O	(alpha O) open new line before the current line to type text

Deleting Text:

dd	deletes current line
(n)**dd**	deletes (n) line(s)
(n)**dw**	deletes (n) word(s)
D	deletes from cursor to end of line
x	deletes current character
(n)**x**	deletes (n) character(s)
X	deletes previous character

Change Commands:

(n)**cc**	changes (n) characters on line(s) until end of the line (or until escape is pressed)
cw	changes characters of word until end of the word (or until escape is pressed)
(n)**cw**	changes characters of the next (n) words
c$	changes text to the end of the line
ct(x)	changes text to the letter (x)
C	changes remaining text on the current line (until stopped by escape key)
~	changes the case of the current character
J	joins the current line and the next line
u	undo the last command just done on this line
.	repeats last change
s	substitutes text for current character
S	substitutes text for current line
:s	substitutes new word(s) for old :<line nos effected> s/old/new/g
&	repeats last substitution (:s) command.
(n)**yy**	yanks (n) lines to buffer
y(n)**w**	yanks (n) words to buffer
p	puts yanked or deleted text after cursor
P	puts yanked or deleted text before cursor

File Manipulation:

:w (file)	writes changes to file (default is current file)
:wq	writes changes to current file and quits edit session
:w! (file)	overwrites file (default is current file)
:q	quits edit session w/no changes made
:q!	quits edit session and discards changes
:n	edits next file in argument list
:f (name)	changes name of current file to (name)
:r (file)	reads contents of file into current edit at the current cursor position (insert a file)
:!(command)	shell escape
:r!(command)	inserts result of shell command at cursor position
ZZ	write changes to current file and exit

10.4 emacs Quick Reference Guide

Emacs commands are accompanied either by simultaneously holding down the control key (indicated by **C-**) or by first hitting the escape key (indicated by **M-**).

Essential Commands	
C-h	help
C-x u	undo
C-x C-g	get out of current operation or command
C-x C-s	save the file
C-x C-c	close Emacs

Cursor movement	
C-f	forward one character
C-b	back one character
C-p	previous line
C-n	next line
C-a	beginning of line
C-e	end of line
C-l	center current line on screen
C-v	scroll forward
M-v	scroll backward
M-f	forward one word
M-b	back one word
M-a	beginning of sentence
M-e	end of sentence
M-[	beginning of paragraph
M-]	end of paragraph
M-<	beginning of buffer
M->	end of buffer

Other Important Functions	
M-(n)	repeat the next command (n) times
C-d	delete a character
M-d	delete a word
C-k	kill line
M-k	kill sentence
C-s	search forward
C-r	search in reverse
M-%	query replace
M-c	capitalize word
M-u	uppercase word
M-l	lowercase word
C-t	transpose characters
M-t	transpose words
C-@	mark beginning of region
C-w	cut--wipe out everything from mark to point
C-y	paste--yank deleted text into current location
M-q	reformat paragraph
M-g	reformat each paragraph in region
M-x auto-fill-mode	turn on word wrap
M-x set-variable <return> fill-column <return> 45	set length of lines to 45 characters
M-x goto-line <return> 16	move cursor to line 16
M-w	copy region marked
C-x C-f	find file and read it
C-x C-v	find and read alternate file
C-x i	insert file at cursor position
C-x C-s	save file
C-x C-w	write buffer to a different file
C-x C-c	exit emacs, and be prompted to save

CHAPTER 11 Unix Command Summary

11.1 Unix Commands

In the table below we summarize the more frequently used commands on a Unix system. In this table, as in general, for most Unix commands, ***file***, could be an actual file name, or a list of file names, or input/output could be redirected to or from the command.

TABLE 11.1 Unix Commands

Command/Syntax	What it will do
awk/nawk [options] *file*	scan for patterns in a file and process the results
cat [options] *file*	concatenate (list) a file
cd [directory]	change directory
chgrp [options] *group file*	change the group of the file
chmod [options] *file*	change file or directory access permissions
chown [options] *owner file*	change the ownership of a file; can only be done by the superuser
chsh (***passwd -e/-s***) *username login_shell*	change the user's login shell (often only by the superuser)
cmp [options] *file1 file2*	compare two files and list where differences occur (text or binary files)
compress [options] *file*	compress file and save it as *file.Z*
cp [options] *file1 file2*	copy *file1* into *file2*; *file2* shouldn't already exist. This command creates or overwrites *file2*.
cut (options) [*file*(s)]	cut specified field(s)/character(s) from lines in file(s)
date [options]	report the current date and time
dd [if=infile] [of=outfile] [operand=value]	copy a file, converting between ASCII and EBCDIC or swapping byte order, as specified
diff [options] *file1 file2*	compare the two files and display the differences (text files only)
df [options] [resource]	report the summary of disk blocks and inodes free and in use
du [options] [*directory* or *file*]	report amount of disk space in use
echo [text string]	echo the text string to stdout
ed or ***ex*** [options] *file*	Unix line editors
emacs [options] *file*	full-screen editor
expr *arguments*	evaluate the arguments. Used to do arithmetic, etc. in the shell.
file [options] *file*	classify the file type

TABLE 11.1 Unix Commands

Command/Syntax	What it will do
find directory [options] [actions]	find files matching a type or pattern
finger [options] *user[@hostname]*	report information about users on local and remote machines
ftp [options] *host*	transfer file(s) using file transfer protocol
grep [options] 'search string' *argument* ***egrep*** [options] 'search string' *argument* ***fgrep*** [options] 'search string' *argument*	search the argument (in this case probably a file) for all occurrences of the search string, and list them.
gzip [options] *file* ***gunzip*** [options] *file* ***zcat*** [options] *file*	compress or uncompress a file. Compressed files are stored with a **.gz** ending
head [-number] *file*	display the first 10 (or number of) lines of a file
hostname	display or set (super-user only) the name of the current machine
kill [options] [-SIGNAL] [pid#] [%job]	send a signal to the process with the process id number (pid#) or job control number (%n). The default signal is to kill the process.
ln [options] *source_file target*	link the *source_file* to the *target*
lpq [options] ***lpstat*** [options]	show the status of print jobs
lpr [options] *file* ***lp*** [options] *file*	print to defined printer
lprm [options] ***cancel*** [options]	remove a print job from the print queue
ls [options] [*directory* or *file*]	list *directory* contents or *file* permissions
mail [options] [user] ***mailx*** [options] [user] ***Mail*** [options] [user]	simple email utility available on Unix systems. Type a period as the first character on a new line to send message out, question mark for help.
man [options] *command*	show the manual (**man**) page for a command
mkdir [options] *directory*	make a *directory*
more [options] *file* ***less*** [options] *file* ***pg*** [options] *file*	page through a text file
mv [options] *file1 file2*	move *file1* into *file2*
od [options] *file*	octal dump a binary file, in octal, ASCII, hex, decimal, or character mode.
passwd [options]	set or change your password
paste [options] *file*	paste field(s) onto the lines in *file*
pr [options] *file*	filter the file and print it on the terminal
ps [options]	show status of active processes

TABLE 11.1 **Unix Commands**

Command/Syntax	What it will do
pwd	print working (current) directory
rcp [options] *hostname*	remotely copy files from this machine to another machine
rlogin [options] *hostname*	login remotely to another machine
rm [options] *file*	remove (delete) a file or directory (**-r** recursively deletes the directory and its contents) (**-i** prompts before removing files)
rmdir [options] *directory*	remove a *directory*
rsh [options] *hostname*	remote shell to run on another machine
script *file*	saves everything that appears on the screen to file until ***exit*** is executed
sed [options] *file*	stream editor for editing files from a script or from the command line
sort [options] *file*	sort the lines of the *file* according to the options chosen
source *file* **.** *file*	read commands from the *file* and execute them in the current shell. *source*: C shell, **.**: Bourne shell.
strings [options] *file*	report any sequence of 4 or more printable characters ending in <NL> or <NULL>. Usually used to search binary files for ASCII strings.
stty [options]	set or display terminal control options
tail [options] *file*	display the last few lines (or parts) of a file
tar key[options] [*file*(s)]	tape archiver--refer to man pages for details on creating, listing, and retrieving from archive files. Tar files can be stored on tape or disk.
tee [options] *file*	copy stdout to one or more files
telnet [host [port]]	communicate with another host using telnet protocol
touch [options] [date] *file*	create an empty file, or update the access time of an existing file
tr [options] *string1 string2*	translate the characters in string1 from stdin into those in string2 in stdout
uncompress *file.Z*	uncompress *file.Z* and save it as a file
uniq [options] *file*	remove repeated lines in a file
uudecode [*file*]	decode a uuencoded file, recreating the original file
uuencode [*file*] *new_name*	encode binary file to 7-bit ASCII, useful when sending via email, to be decoded as new_name at destination
vi [options] *file*	visual, full-screen editor
wc [options] [*file*(s)]	display word (or character or line) count for *file*(s)
whereis [options] *command*	report the binary, source, and man page locations for the command named
which *command*	reports the path to the command or the shell alias in use
who or ***w***	report who is logged in and what processes are running
zcat *file.Z*	concatenate (list) uncompressed file to screen, leaving file compressed on disk

CHAPTER 12 A Short Unix Bibliography

12.1 Highly Recommended

UNIX for the Impatient, Paul W. Abrahams & Bruce R. Larson (Addison-Wesley Publishing Company, 1992, ISBN 0-201-55703-7). (A current favorite. Recommended in the CIS Department for Unix beginners.)

UNIX in a Nutshell for BSD 4.3: A Desktop Quick Reference For Berkeley (O'Reilly & Associates, Inc., 1990, ISBN 0-937175-20-X). (A handy reference for BSD.)

UNIX in a Nutshell: A Desktop Quick Reference for System V & Solaris 2.0 (O'Reilly & Associates, Inc., 1992, ISBN 0-56592-001-5). (A handy reference for SysV and Solaris 2.)

The UNIX Programming Environment, Brian W. Kernighan & Rob Pike (Prentice Hall, 1984). (A classic. For serious folks.)

When You Can't Find Your UNIX System Administrator, Linda Mui (O'Reilly & Associates, Inc., 1995, ISBN 1-56592-104-6).

UNIX Power Tools, Jerry Peek, Tim O'Reilly, and Mike Loukides (O'Reilly & Associates, 1993, ISBN 0-679-79073-X). (Includes a CDROM of useful software for various OSs.)

12.2 Assorted Others

Understanding UNIX: A Conceptual Guide, James R. Groff & Paul N. Weinberg (Que Corporation, 1983).

Exploring the UNIX System, Stephen G. Kochan & Patrick H. Wood (SAMS, a division of Macmillan Computer Publishing, 1989, ISBN 0-8104-6268-0).

Learning GNU Emacs, Debra Cameron and Bill Rosenblatt (O'Reilly & Associates, 1992, ISBN 0-937175-84-6).

UNIX for Dummies, John R. Levine & Margaret Levine Young (IDG Books Worldwide, Inc., 1993, ISBN 0-878058-58-4).

A Practical Guide to UNIX System V, Mark G. Sobell (The Benjamin/Cummings Publishing Company, Inc., 1985, ISBN 0-80-530243-3).

UNIX Primer Plus, Mitchell Waite, Donald Martin, & Stephen Prata, (Howard W. Sams & Co., Inc., 1983, ISBN 0-672-30194-6).

An Introduction to Berkeley UNIX, Paul Wang, (Wadsworth Publishing Company, 1988).

Unix Shell Programming, Stephen G. Kochan & Patrick H. Wood (Hayden Book Co., 1990, ISBN 0-8104-6309-1).

The Unix C Shell Field Guide, Gail Anderson and Paul Anderson (Prentice Hall, 1986, ISBN 0-13-937468-X).

A Student's Guide to UNIX, Harley Hahn. (McGraw-Hill, 1993, ISBN 0-07-025511-3).

Tricks of the UNIX Masters, Russell G. Sage (Howard W. Sams & Co., Inc., 1987, ISBN 0-672-22449-6).

CHAPTER 12 A Short Unix Bibliography

12.1 Highly Recommended

UNIX for the Impatient, Paul W. Abrahams & Bruce R. Larson (Addison-Wesley Publishing Company, 1992, ISBN 0-201-55703-7). (A current favorite. Recommended in the CIS Department for Unix beginners.)

UNIX in a Nutshell for BSD 4.3: A Desktop Quick Reference For Berkeley (O'Reilly & Associates, Inc., 1990, ISBN 0-937175-20-X). (A handy reference for BSD.)

UNIX in a Nutshell: A Desktop Quick Reference for System V & Solaris 2.0 (O'Reilly & Associates, Inc., 1992, ISBN 0-56592-001-5). (A handy reference for SysV and Solaris 2.)

The UNIX Programming Environment, Brian W. Kernighan & Rob Pike (Prentice Hall, 1984). (A classic. For serious folks.)

When You Can't Find Your UNIX System Administrator, Linda Mui (O'Reilly & Associates, Inc., 1995, ISBN 1-56592-104-6).

UNIX Power Tools, Jerry Peek, Tim O'Reilly, and Mike Loukides (O'Reilly & Associates, 1993, ISBN 0-679-79073-X). (Includes a CDROM of useful software for various OSs.)

12.2 Assorted Others

Understanding UNIX: A Conceptual Guide, James R. Groff & Paul N. Weinberg (Que Corporation, 1983).

Exploring the UNIX System, Stephen G. Kochan & Patrick H. Wood (SAMS, a division of Macmillan Computer Publishing, 1989, ISBN 0-8104-6268-0).

Learning GNU Emacs, Debra Cameron and Bill Rosenblatt (O'Reilly & Associates, 1992, ISBN 0-937175-84-6).

UNIX for Dummies, John R. Levine & Margaret Levine Young (IDG Books Worldwide, Inc., 1993, ISBN 0-878058-58-4).

A Practical Guide to UNIX System V, Mark G. Sobell (The Benjamin/Cummings Publishing Company, Inc., 1985, ISBN 0-80-530243-3).

UNIX Primer Plus, Mitchell Waite, Donald Martin, & Stephen Prata, (Howard W. Sams & Co., Inc., 1983, ISBN 0-672-30194-6).

An Introduction to Berkeley UNIX, Paul Wang, (Wadsworth Publishing Company, 1988).

Unix Shell Programming, Stephen G. Kochan & Patrick H. Wood (Hayden Book Co., 1990, ISBN 0-8104-6309-1).

The Unix C Shell Field Guide, Gail Anderson and Paul Anderson (Prentice Hall, 1986, ISBN 0-13-937468-X).

A Student's Guide to UNIX, Harley Hahn. (McGraw-Hill, 1993, ISBN 0-07-025511-3).

Tricks of the UNIX Masters, Russell G. Sage (Howard W. Sams & Co., Inc., 1987, ISBN 0-672-22449-6).

An Introduction to the Z Shell

Paul Falstad
pf@z-code.com

Bas de Bakker
bas@phys.uva.nl

An Introduction to the Z Shell

Paul Falstad
pf@z-code.com

Bas de Bakker
bas@phys.uva.nl

Introduction

zsh is a shell designed for interactive use, although it is also a powerful scripting language. Many of the useful features of bash, ksh, and tcsh were incorporated into **zsh**; many original features were added. This document details some of the unique features of **zsh**. It assumes basic knowledge of the standard UNIX shells; the intent is to show a reader already familiar with one of the other major shells what makes **zsh** more useful or more powerful. This document is not at all comprehensive; read the manual entry for a description of the shell that is complete and concise, although somewhat overwhelming and devoid of examples.

The text will frequently mention options that you can set to change the behaviour of **zsh**. You can set these options with the command

```
% setopt optionname
```

and unset them again with

```
% unsetopt optionname
```

Case is ignored in option names, as are embedded underscores.

Filename Generation

Otherwise known as *globbing*, filename generation is quite extensive in **zsh**. Of course, it has all the basics:

```
% ls
Makefile   file.pro   foo.o      main.o     q.c        run234     stuff
bar.o      foo        link       morestuff  run123     run240     sub
file.h     foo.c      main.h     pipe       run2       run303
% ls *.c
foo.c  q.c
% ls *.[co]
bar.o    foo.c    foo.o    main.o  q.c
% ls foo.?
foo.c  foo.o
% ls *.[^c]
bar.o    file.h   foo.o    main.h  main.o
% ls *.[^oh]
foo.c  q.c
```

Also, if the *EXTENDEDGLOB* option is set, some new features are activated. For example, the ^ character negates the pattern following it:

```
% setopt extendedglob
% ls -d ^*.c
Makefile   file.pro   link       morestuff  run2       run303
bar.o      foo        main.h     pipe       run234     stuff
file.h     foo.o      main.o     run123     run240     sub
% ls -d ^*.*
Makefile   link       pipe       run2       run240     stuff
foo        morestuff  run123     run234     run303     sub
% ls -d ^Makefile
bar.o      foo        link       morestuff  run123     run240     sub
file.h     foo.c      main.h     pipe       run2       run303
file.pro   foo.o      main.o     q.c        run234     stuff
% ls -d *.^c
.rhosts   bar.o     file.h    file.pro  foo.o     main.h    main.o
```

An expression of the form <*x*–*y*> matches a range of integers:

```
% ls run<200-300>
run234  run240
% ls run<300-400>
run303
% ls run<-200>
run123  run2
% ls run<300->
run303
% ls run<>
run123  run2    run234  run240  run303
```

The *NUMERICGLOBSORT* option will sort files with numbers according to the number. This will not work with `ls` as it resorts its arguments:

```
% setopt numericglobsort
% echo run<>
run2 run123 run234 run240 run303
```

Grouping is possible:

```
% ls (foo|bar).*
bar.o  foo.c  foo.o
% ls *.(c|o|pro)
bar.o     file.pro  foo.c     foo.o     main.o    q.c
```

Also, the string `**/` forces a recursive search of subdirectories:

```
% ls -R
Makefile    file.pro    foo.o       main.o      q.c         run234      stuff
bar.o       foo         link        morestuff   run123      run240      sub
file.h      foo.c       main.h      pipe        run2        run303

morestuff:

stuff:
file  xxx   yyy

stuff/xxx:
foobar

stuff/yyy:
frobar
% ls **/*bar
stuff/xxx/foobar  stuff/yyy/frobar
% ls **/f*
file.h            foo               foo.o             stuff/xxx/foobar
file.pro          foo.c             stuff/file        stuff/yyy/frobar
% ls *bar*
bar.o
% ls **/*bar*
bar.o             stuff/xxx/foobar  stuff/yyy/frobar
% ls stuff/**/*bar*
stuff/xxx/foobar  stuff/yyy/frobar
```

It is possible to exclude certain files from the patterns using the ~ character. A pattern of the form `*.c~bar.c` lists all files matching `*.c`, except for the file `bar.c`.

```
% ls *.c
foo.c    foob.c    bar.c
% ls *.c~bar.c
foo.c    foob.c
% ls *.c~f*
bar.c
```

One can add a number of *qualifiers* to the end of any of these patterns, to restrict matches to certain file types. A qualified pattern is of the form

pattern (...)

with single-character qualifiers inside the parentheses.

```
% alias l='ls -dF'
% l *
Makefile    foo*        main.h      q.c         run240
bar.o       foo.c       main.o      run123      run303
file.h      foo.o       morestuff/  run2        stuff/
file.pro    link@       pipe        run234      sub
% l *(/)
morestuff/  stuff/
% l *(@)
link@
% l *(*)
foo*        link@       morestuff/  stuff/
% l *(x)
foo*        link@       morestuff/  stuff/
% l *(X)
foo*        link@       morestuff/  stuff/
% l *(R)
bar.o       foo*        link@       morestuff/  run123      run240
file.h      foo.c       main.h      pipe        run2        run303
file.pro    foo.o       main.o      q.c         run234      stuff/
```

Note that `*(x)` and `*(*)` both match executables. `*(X)` matches files executable by others, as opposed to `*(x)`, which matches files executable by the owner. `*(R)` and `*(r)` match readable files; `*(W)` and `*(w)`, which checks for writable files. `*(W)` is especially important, since it checks for world-writable files:

```
% l *(w)
bar.o       foo*        link@       morestuff/  run123      run240
file.h      foo.c       main.h      pipe        run2        run303
file.pro    foo.o       main.o      q.c         run234      stuff/
% l *(W)
link@   run240
% l -l link run240
lrwxrwxrwx  1 pfalstad      10 May 23 18:12 link -> /usr/bin/
-rw-rw-rw-  1 pfalstad       0 May 23 18:12 run240
```

If you want to have all the files of a certain type as well as all symbolic links pointing to files of that type, prefix the qualifier with a `-`:

```
% l *(-/)
link@       morestuff/  stuff/
```

You can filter out the symbolic links with the ^ character:

```
% l *(W^@)
run240
% l *(x)
foo*        link@       morestuff/  stuff/
% l *(x^@/)
foo*
```

To find all plain files, you can use `.`:

```
% l *(.)
Makefile  file.h    foo*      foo.o     main.o    run123    run234    run303
bar.o     file.pro  foo.c     main.h    q.c       run2      run240    sub
% l *(^.)
link@       morestuff/  pipe        stuff/
% l s*(.)
stuff/   sub
% l *(p)
pipe
% l -l *(p)
prw-r--r--  1 pfalstad       0 May 23 18:12 pipe
```

`*(U)` matches all files owned by you. To search for all files not owned by you, use `*(^U)`:

```
% l -l *(^U)
-rw-------  1 subbarao      29 May 23 18:13 sub
```

This searches for setuid files:

```
% l -l *(s)
-rwsr-xr-x  1 pfalstad      16 May 23 18:12 foo*
```

This checks for a certain user's files:

```
% l -l *(u[subbarao])
-rw-------  1 subbarao      29 May 23 18:13 sub
```

Startup Files

There are five startup files that **zsh** will read commands from:

```
$ZDOTDIR/.zshenv
$ZDOTDIR/.zprofile
$ZDOTDIR/.zshrc
$ZDOTDIR/.zlogin
$ZDOTDIR/.zlogout
```

If **ZDOTDIR** is not set, then the value of **HOME** is used; this is the usual case.

`.zshenv` is sourced on all invocations of the shell, unless the `-f` option is set. It should contain commands to set the command search path, plus other important environment variables. `.zshenv` should not contain commands that produce output or assume the shell is attached to a tty.

`.zshrc` is sourced in interactive shells. It should contain commands to set up aliases, functions, options, key bindings, etc.

`.zlogin` is sourced in login shells. It should contain commands that should be executed only in login shells. `.zlogout` is sourced when login shells exit. `.zprofile` is similar to `.zlogin`, except that it is sourced before `.zshrc`. `.zprofile` is meant as an alternative to `.zlogin` for ksh fans; the two are not intended to be used together, although this could certainly be done if desired. `.zlogin` is not the place for alias definitions, options, environment variable settings, etc.; as a general rule, it should not change the shell environment at all. Rather, it should be used to set the terminal type and run a series of external commands (`fortune`, `msgs`, etc).

Shell Functions

zsh also allows you to create your own commands by defining shell functions. For example:

```
% yp () {
>       ypmatch $1 passwd.byname
> }
% yp pfalstad
pfalstad:*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
```

This function looks up a user in the NIS password map. The `$1` expands to the first argument to `yp`. The function could have been equivalently defined in one of the following ways:

```
% function yp {
>       ypmatch $1 passwd.byname
> }
% function yp () {
>       ypmatch $1 passwd.byname
> }
% function yp () ypmatch $1 passwd.byname
```

Note that aliases are expanded when the function definition is parsed, not when the function is executed. For example:

```
% alias ypmatch=echo
% yp pfalstad
pfalstad:*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
```

Since the alias was defined after the function was parsed, it has no effect on the function's execution. However, if we define the function again with the alias in place:

```
% function yp () { ypmatch $1 passwd.byname }
% yp pfalstad
pfalstad passwd.byname
```

it is parsed with the new alias definition in place. Therefore, in general you must define aliases before functions.

We can make the function take multiple arguments:

```
% unalias ypmatch
% yp () {
>       for i
>       do ypmatch $i passwd.byname
>       done
> }
% yp pfalstad subbarao sukthnkr
pfalstad:*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
subbarao:*:3338:35:Kartik Subbarao:/u/subbarao:/usr/princeton/bin/zsh
sukthnkr:*:1267:35:Rahul Sukthankar:/u/sukthnkr:/usr/princeton/bin/tcsh
```

The `for i` loops through each of the function's arguments, setting `i` equal to each of them in turn. We can also make the function do something sensible if no arguments are given:

```
% yp () {
>       if (( $# == 0 ))
>       then echo usage: yp name ...; fi
>       for i; do ypmatch $i passwd.byname; done
> }
% yp
usage: yp name ...
% yp pfalstad sukthnkr
pfalstad:*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
sukthnkr:*:1267:35:Rahul Sukthankar:/u/sukthnkr:/usr/princeton/bin/tcsh
```

`$#` is the number of arguments supplied to the function. If it is equal to zero, we print a usage message; otherwise, we loop through the arguments, and `ypmatch` all of them.

Here's a function that selects a random line from a file:

```
% randline () {
>       integer z=$(wc -l <$1)
>       sed -n $[RANDOM % z + 1]p $1
> }
% randline /etc/motd
PHOENIX WILL BE DOWN briefly Friday morning, 5/24/91 from 8 AM to
% randline /etc/motd
SunOS Release 4.1.1 (PHOENIX) #19: Tue May 14 19:03:15 EDT 1991
% randline /etc/motd
| Please use the "msgs" command to read announcements.  Refer to the    |
% echo $z

%
```

`randline` has a local variable, `z`, that holds the number of lines in the file. `$[RANDOM % z + 1]` expands to a random number between 1 and `z`. An expression of the form `$[...]` expands to the value of the arithmetic expression within the brackets, and the **RANDOM** variable returns a random number each time it is referenced. `%` is the modulus operator, as in C. Therefore, `sed -n $[RANDOM%z+1]p` picks a random line from its input, from 1 to `z`.

Function definitions can be viewed with the `functions` builtin:

```
% functions randline
randline () {
        integer z=$(wc -l <$1)
        sed -n $[RANDOM % z + 1]p $1

}
% functions
yp () {
        if let $# == 0

        then
                echo usage: yp name ...

        fi
        for i
        do
                ypmatch $i passwd.byname

                done

}
randline () {
        integer z=$(wc -l <$1)
        sed -n $[RANDOM % z + 1]p $1

}
```

Here's another one:

```
% cx () { chmod +x $* }
% ls -l foo bar
-rw-r--r--  1 pfalstad       29 May 24 04:38 bar
-rw-r--r--  1 pfalstad       29 May 24 04:38 foo
% cx foo bar
% ls -l foo bar
-rwxr-xr-x  1 pfalstad       29 May 24 04:38 bar
-rwxr-xr-x  1 pfalstad       29 May 24 04:38 foo
```

Note that this could also have been implemented as an alias:

```
% chmod 644 foo bar
% alias cx='chmod +x'
% cx foo bar
% ls -l foo bar
-rwxr-xr-x  1 pfalstad       29 May 24 04:38 bar
-rwxr-xr-x  1 pfalstad       29 May 24 04:38 foo
```

Instead of defining a lot of functions in your `.zshrc`, all of which you may not use, it is often better to use the `autoload` builtin. The idea is, you create a directory where function definitions are stored, declare the names in your `.zshrc`, and tell the shell where to look for them. Whenever you reference a function, the shell will automatically load it into memory.

```
% mkdir /tmp/funs
% cat >/tmp/funs/yp
ypmatch $1 passwd.byname
^D
% cat >/tmp/funs/cx
chmod +x $*
^D
% FPATH=/tmp/funs
% autoload cx yp
% functions cx yp
undefined cx ()
undefined yp ()
% chmod 755 /tmp/funs/{cx,yp}
% yp egsirer
egsirer:*:3214:35:Emin Gun Sirer:/u/egsirer:/bin/sh
% functions yp
yp () {
        ypmatch $1 passwd.byname
}
```

This idea has other benefits. By adding a `#!` header to the files, you can make them double as shell scripts. (Although it is faster to use them as functions, since a separate process is not created.)

```
% ed /tmp/funs/yp
25
i
#! /usr/local/bin/zsh
w
42
q
% </tmp/funs/yp
#! /usr/local/bin/zsh
ypmatch $1 passwd.byname
% /tmp/funs/yp sukthnkr
sukthnkr:*:1267:35:Rahul Sukthankar:/u/sukthnkr:/usr/princeton/bin/tcsh
```

Now other people, who may not use **zsh**, or who don't want to copy all of your `.zshrc`, may use these functions as shell scripts.

Directories

One nice feature of **zsh** is the way it prints directories. For example, if we set the prompt like this:

```
phoenix% PROMPT='%~> '
~> cd src
~/src>
```

the shell will print the current directory in the prompt, using the ~ character. However, **zsh** is smarter than most other shells in this respect:

```
~/src> cd ~subbarao
~subbarao> cd ~maruchck
~maruchck> cd lib
~maruchck/lib> cd fun
~maruchck/lib/fun> foo=/usr/princeton/common/src
~maruchck/lib/fun> cd ~foo
~foo> cd ..
/usr/princeton/common> cd src
~foo> cd news/nntp
~foo/news/nntp> cd inews
~foo/news/nntp/inews>
```

Note that **zsh** prints *other* users' directories in the form ~user. Also note that you can set a parameter and use it as a directory name; **zsh** will act as if foo is a user with the login directory /usr/princeton/common/src. This is convenient, especially if you're sick of seeing prompts like this:

```
phoenix:/usr/princeton/common/src/X.V11R4/contrib/clients/xv/docs>
```

If you get stuck in this position, you can give the current directory a short name, like this:

```
/usr/princeton/common/src/news/nntp/inews> inews=$PWD
/usr/princeton/common/src/news/nntp/inews> echo ~inews
/usr/princeton/common/src/news/nntp/inews
~inews>
```

When you reference a directory in the form ~inews, the shell assumes that you want the directory displayed in this form; thus simply typing echo ~inews or cd ~inews causes the prompt to be shortened. You can define a shell function for this purpose:

```
~inews> namedir () { $1=$PWD ;  : ~$1 }
~inews> cd /usr/princeton/bin
/usr/princeton/bin> namedir pbin
~pbin> cd /var/spool/mail
/var/spool/mail> namedir spool
~spool> cd .msgs
~spool/.msgs>
```

You may want to add this one-line function to your .zshrc.

zsh can also put the current directory in your title bar, if you are using a windowing system. One way to do this is with the chpwd function, which is automatically executed by the shell whenever you change directory. If you are using xterm, this will work:

```
chpwd () { print -Pn '^[]2;%~^G' }
```

The -P option tells print to treat its arguments like a prompt string; otherwise the %~ would not be expanded. The -n option suppresses the terminating newline, as with echo.

If you are using an IRIS wsh, do this:

```
chpwd () { print -Pn '\2201.y%~\234' }
```

The print -P command has other uses. For example, to print the current directory to standard output in short form, you can do this:

```
% print -D $PWD
~subbarao/src
```

and to print each component of the path in short form:

```
% print -D $path
/bin /usr/bin ~locbin ~locbin/X11 ~/bin
```

Directory Stacks

If you use csh, you may know about directory stacks. The pushd command puts the current directory on the stack, and changes to a new directory; the popd command pops a directory off the stack and changes to it.

```
phoenix% cd
phoenix% PROMPT='Z %~> '
Z ~> pushd /tmp
/tmp ~
Z /tmp> pushd /usr/etc
/usr/etc /tmp ~
Z /usr/etc> pushd /usr/bin
/usr/bin /usr/etc /tmp ~
Z /usr/bin> popd
/usr/etc /tmp ~
Z /usr/etc> popd
/tmp ~
Z /tmp> pushd /etc
/etc /tmp ~
Z /etc> popd
/tmp ~
```

zsh's directory stack commands work similarly. One difference is the way pushd is handled if no arguments are given. As in csh, this exchanges the top two elements of the directory stack:

```
Z /tmp> dirs
/tmp ~
Z /tmp> pushd
~ /tmp
```

unless the stack only has one entry:

```
Z ~> popd
/tmp
Z /tmp> dirs
/tmp
Z /tmp> pushd
~ /tmp
Z ~>
```

or unless the *PUSHDTOHOME* option is set:

```
Z ~> setopt pushdtohome
Z ~> pushd
~ ~ /tmp
```

As an alternative to using directory stacks in this manner, we can get something like a *directory history* by setting a few more options and parameters:

```
~> DIRSTACKSIZE=8
~> setopt autopushd pushdminus pushdsilent pushdtohome
~> alias dh='dirs -v'
~> cd /tmp
/tmp> cd /usr
/usr> cd bin
/usr/bin> cd ../pub
/usr/pub> dh
0       /usr/pub
1       /usr/bin
2       /usr
3       /tmp
4       ~
/usr/pub> cd -3
/tmp> dh
0       /tmp
1       /usr/pub
2       /usr/bin
3       /usr
4       ~
/tmp> ls =2/df
/usr/bin/df
/tmp> cd -4
~>
```

Note that =2 expanded to the second directory in the history list, and that `cd -3` recalled the third directory in the list.

You may be wondering what all those options do. *AUTOPUSHD* made `cd` act like `pushd`. (`alias cd=pushd` is not sufficient, for various reasons.) *PUSHDMINUS* swapped the meaning of `cd +1` and `cd -1`; we want them to mean the opposite of what they mean in csh, because it makes more sense in this scheme, and it's easier to type:

```
~> dh
0       ~
1       /tmp
2       /usr/pub
3       /usr/bin
4       /usr
~> unsetopt pushdminus
~> cd +1
/tmp> dh
0       /tmp
1       ~
2       /usr/pub
3       /usr/bin
4       /usr
/tmp> cd +2
/usr/pub>
```

PUSHDSILENT keeps the shell from printing the directory stack each time we do a `cd`, and *PUSHDTOHOME* we mentioned earlier:

```
/usr/pub> unsetopt pushdsilent
/usr/pub> cd /etc
/etc /usr/pub /tmp ~ /usr/bin /usr
/etc> cd
~ /etc /usr/pub /tmp ~ /usr/bin /usr
~> unsetopt pushdtohome
~> cd
/etc ~ /usr/pub /tmp ~ /usr/bin /usr
/etc>
```

DIRSTACKSIZE keeps the directory stack from getting too large, much like *HISTSIZE*:

```
/etc> setopt pushdsilent
/etc> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> dh
0       /
1       /
2       /
3       /
4       /
5       /
6       /
7       /
```

Command/Process Substitution

Command substitution in **zsh** can take two forms. In the traditional form, a command enclosed in backquotes (`...`) is replaced on the command line with its output. This is the form used by the older shells. Newer shells (like **zsh**) also provide another form, `$(...)`. This form is much easier to nest.

```
% ls -l `echo /vmunix`
-rwxr-xr-x  1 root      1209702 May 14 19:04 /vmunix
% ls -l $(echo /vmunix)
-rwxr-xr-x  1 root      1209702 May 14 19:04 /vmunix
% who | grep mad
subbarao ttyt7   May 23 15:02   (mad55sx15.Prince)
pfalstad ttyu1   May 23 16:25   (mad55sx14.Prince)
subbarao ttyu6   May 23 15:04   (mad55sx15.Prince)
pfalstad ttyv3   May 23 16:25   (mad55sx14.Prince)
% who | grep mad | awk '{print $2}'
ttyt7
ttyu1
ttyu6
ttyv3
% cd /dev; ls -l $(who |
> grep $(echo mad) |
> awk '{ print $2 }')
crwx-w----  1 subbarao  20,  71 May 23 18:35 ttyt7
crw--w----  1 pfalstad  20,  81 May 23 18:42 ttyu1
crwx-w----  1 subbarao  20,  86 May 23 18:38 ttyu6
crw--w----  1 pfalstad  20,  99 May 23 18:41 ttyv3
```

Many common uses of command substitution, however, are superseded by other mechanisms of **zsh**:

```
% ls -l `tty`
crw-rw-rw-  1 root      20,  28 May 23 18:35 /dev/ttyqc
% ls -l $TTY
crw-rw-rw-  1 root      20,  28 May 23 18:35 /dev/ttyqc
% ls -l `which rn`
-rwxr-xr-x  1 root       172032 Mar  6 18:40 /usr/princeton/bin/rn
% ls -l =rn
-rwxr-xr-x  1 root       172032 Mar  6 18:40 /usr/princeton/bin/rn
```

A command name with a = prepended is replaced with its full pathname. This can be very convenient. If it's not convenient for you, you can turn it off:

```
% ls
=foo    =bar
% ls =foo =bar
zsh: foo not found
% setopt noequals
% ls =foo =bar
=foo    =bar
```

Another nice feature is process substitution:

```
% who | fgrep -f =(print -l root lemke shgchan subbarao)
root     console May 19 10:41
lemke    ttyq0   May 22 10:05   (narnia:0.0)
lemke    ttyr7   May 22 10:05   (narnia:0.0)
lemke    ttyrd   May 22 10:05   (narnia:0.0)
shgchan  ttys1   May 23 16:52   (gaudi.Princeton.)
subbarao ttyt7   May 23 15:02   (mad55sx15.Prince)
subbarao ttyu6   May 23 15:04   (mad55sx15.Prince)
shgchan  ttyvb   May 23 16:51   (gaudi.Princeton.)
```

A command of the form =(...) is replaced with the name of a *file* containing its output. (A command substitution, on the other hand, is replaced with the output itself.) `print -l` is like `echo`, excepts that it prints its arguments one per line, the way `fgrep` expects them:

```
% print -l foo bar
foo
bar
```

We could also have written:

```
% who | fgrep -f =(echo 'root
> lemke
> shgchan
> subbarao')
```

Using process substitution, you can edit the output of a command:

```
% ed =(who | fgrep -f ~/.friends)
355
g/lemke/d
w /tmp/filbar
226
q
% cat /tmp/filbar
root     console May 19 10:41
shgchan  ttys1   May 23 16:52   (gaudi.Princeton.)
subbarao ttyt7   May 23 15:02   (mad55sx15.Prince)
subbarao ttyu6   May 23 15:04   (mad55sx15.Prince)
shgchan  ttyvb   May 23 16:51   (gaudi.Princeton.)
```

or easily read archived mail:

```
% mail -f =(zcat ~/mail/oldzshmail.Z)
"/tmp/zsha06024": 84 messages, 0 new, 43 unread
>  1  U  TO: pfalstad, zsh (10)
   2  U  nytim!tim@uunet.uu.net, Re: Zsh on Sparc1 /SunOS 4.0.3
   3  U  JAM%TPN@utrcgw.utc.com, zsh fix (15)
   4  U  djm@eng.umd.edu, way to find out if running zsh? (25)
   5  U  djm@eng.umd.edu, Re: way to find out if running zsh? (17)
   6   r djm@eng.umd.edu, Meta . (18)
   7  U  jack@cs.glasgow.ac.uk, Re: problem building zsh (147)
   8  U  nytim!tim@uunet.uu.net, Re: Zsh on Sparc1 /SunOS 4.0.3
   9     ursa!jmd, Another fix... (61)
  10  U  pplacewa@bbn.com, Re: v18i084: Zsh 2.00 - A small complaint (36)
  11  U  lubkin@cs.rochester.edu, POSIX job control (34)
  12  U  yale!bronson!tan@uunet.UU.NET
  13  U  brett@rpi.edu, zsh (36)
  14  S  subbarao, zsh sucks!!!! (286)
  15  U  snibru!d241s008!d241s013!ala@relay.EU.net, zsh (165)
  16  U  nytim!tim@uunet.UU.NET, Re: Zsh on Sparc1 /SunOS 4.0.3
  17  U  subbarao, zsh is a junk shell (43)
  18  U  amaranth@vela.acs.oakland.edu, zsh (33)
43u/84 1: x
% ls -l /tmp/zsha06024
/tmp/zsha06024 not found
```

Note that the shell creates a temporary file, and deletes it when the command is finished.

```
% diff =(ls) =(ls -F)
3c3
< fortune
---
> fortune*
10c10
< strfile
---
> strfile*
```

If you read **zsh**'s man page, you may notice that `<(...)` is another form of process substitution which is similar to `=(...)`. There is an important difference between the two. In the `<(...)` case, the shell creates a named pipe (FIFO) instead of a file. This is better, since it does not fill up the file system; but it does not work in all cases. In fact, if we had replaced `=(...)` with `<(...)` in the examples above, all of them would have stopped working except for `fgrep -f <(...)`. You can not edit a pipe, or open it as a mail folder; `fgrep`, however, has no problem with reading a list of words from a pipe. You may wonder why `diff <(foo) bar` doesn't work, since `foo | diff - bar` works; this is because `diff` creates a temporary file if it notices that one of its arguments is `-`, and then copies its standard input to the temporary file.

`>(...)` is just like `<(...)` except that the command between the parentheses will get its input from the named pipe.

```
% dvips -o >(lpr) zsh.dvi
```

Redirection

Apart from all the regular redirections like the Bourne shell has, **zsh** can do more. You can send the output of a command to more than one file, by specifying more redirections like

```
% echo Hello World >file1 >file2
```

and the text will end up in both files. Similarly, you can send the output to a file and into a pipe:

```
% make > make.log | grep Error
```

The same goes for input. You can make the input of a command come from more than one file.

```
% sort <file1 <file2 <file3
```

The command will first get the contents of file1 as its standard input, then those of file2 and finally the contents of file3. This, too, works with pipes.

```
    % cut -d: -f1 /etc/passwd | sort <newnames
```

The sort will get as its standard input first the output of `cut` and then the contents of `newnames`.

Suppose you would like to watch the standard output of a command on your terminal, but want to pipe the standard error to another command. An easy way to do this in **zsh** is by redirecting the standard error using `2> >(...)`.

```
    % find / -name games 2> >(grep -v 'Permission' > realerrors)
```

The above redirection will actually be implemented with a regular pipe, not a temporary named pipe.

Aliasing

Often-used commands can be abbreviated with an alias:

```
    % alias uc=uncompress
    % ls
    hanoi.Z
    % uc hanoi
    % ls
    hanoi
```

or commands with certain desired options:

```
    % alias fm='finger -m'
    % fm root
    Login name: root                            In real life: Operator
    Directory: /                                Shell: /bin/csh
    On since May 19 10:41:15 on console         3 days 5 hours Idle Time
    No unread mail
    No Plan.

    % alias lock='lock -p -60000'
    % lock
    lock: /dev/ttyr4 on phoenix. timeout in 60000 minutes
    time now is Fri May 24 04:23:18 EDT 1991
    Key:

    % alias l='ls -AF'
    % l /
    .bash_history             kadb*
    .bashrc                   lib@
    .cshrc                    licensed/
    .exrc                     lost+found/
    .login                    macsyma
    ...
```

Aliases can also be used to replace old commands:

```
    % alias grep=egrep ps=sps make=gmake
    % alias whoami='echo root'
    % whoami
    root
```

or to define new ones:

```
% cd /
% alias sz='ls -l | sort -n +3 | tail -10'
% sz
drwxr-sr-x  7 bin          3072 May 23 11:59 etc
drwxrwxrwx 26 root         5120 May 24 04:20 tmp
drwxr-xr-x  2 root         8192 Dec 26 19:34 lost+found
drwxr-sr-x  2 bin         14848 May 23 18:48 dev
-r--r--r--  1 root       140520 Dec 26 20:08 boot
-rwxr-xr-x  1 root       311172 Dec 26 20:08 kadb
-rwxr-xr-x  1 root      1209695 Apr 16 15:33 vmunix.old
-rwxr-xr-x  1 root      1209702 May 14 19:04 vmunix
-rwxr-xr-x  1 root      1209758 May 21 12:23 vmunix.new.kernelmap.old
-rwxr-xr-x  1 root      1711848 Dec 26 20:08 vmunix.org
% cd
% alias rable='ls -AFtrd *(R)' nrable='ls -AFtrd *(^R)'
% rable
README       func/        bin/         pub/         News/        src/
nicecolors   etc/         scr/         tmp/         iris/        zsh*
% nrable
Mailboxes/   mail/        notes
```

(The pattern `*(R)` matches all readable files in the current directory, and `*(^R)` matches all unreadable files.)

Most other shells have aliases of this kind (*command* aliases). However, **zsh** also has *global* aliases, which are substituted anywhere on a line. Global aliases can be used to abbreviate frequently-typed usernames, hostnames, etc.

```
% alias -g me=pfalstad gun=egsirer mjm=maruchck
% who | grep me
pfalstad ttyp0   May 24 03:39   (mickey.Princeton)
pfalstad ttyp5   May 24 03:42   (mickey.Princeton)
% fm gun
Login name: egsirer                    In real life: Emin Gun Sirer
Directory: /u/egsirer                  Shell: /bin/sh
Last login Thu May 23 19:05 on ttyq3 from bow.Princeton.ED
New mail received Fri May 24 02:30:28 1991;
  unread since Fri May 24 02:30:27 1991
% alias -g phx=phoenix.princeton.edu warc=wuarchive.wustl.edu
% ftp warc
Connected to wuarchive.wustl.edu.
```

Here are some more interesting uses.

```
% alias -g M='| more' GF='| fgrep -f ~/.friends'
% who M     # pipes the output of who through more
% who GF    # see if your friends are on
% w GF      # see what your friends are doing
```

Another example makes use of **zsh**'s process substitution. If you run NIS, and you miss being able to do this:

```
% grep pfalstad /etc/passwd
```

you can define an alias that will seem more natural than `ypmatch pfalstad passwd`:

```
% alias -g PASS='<(ypcat passwd)'
% grep pfalstad PASS
pfalstad:*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
```

If you're really crazy, you can even call it `/etc/passwd`:

```
% alias -g /etc/passwd='<(ypcat passwd)'
% grep pfalstad /etc/passwd
pfalstad:*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
```

The last example shows one of the perils of global aliases; they have a lot of potential to cause confusion. For example, if you defined a global alias called `|` (which is possible), **zsh** would begin to act very strangely; every pipe symbol would be replaced with the text of your alias. To some extent, global aliases are like macros in C; discretion is advised in using them and in choosing names for them. Using names in all caps is not a bad idea, especially for aliases which introduce shell metasyntax (like `M` and `GF` above).

Note that **zsh** aliases are not like csh aliases. The syntax for defining them is different, and they do not have arguments. All your favorite csh aliases will probably not work under **zsh**. For example, if you try:

```
alias rm mv '\!* /tmp/wastebasket'
```

no aliases will be defined, but **zsh** will not report an error. In csh, this line defines an alias that makes `rm` safe---files that are `rm`'d will be moved to a temporary directory instead of instantly destroyed. In **zsh**'s syntax, however, this line asks the shell to print any existing alias definitions for `rm`, `mv`, or `!* /tmp/wastebasket`. Since there are none, most likely, the shell will not print anything, although `alias` will return a nonzero exit code. The proper syntax is this:

```
alias rm='mv \!* /tmp/wastebasket'
```

However, this won't work either:

```
% rm foo.dvi
zsh: no matches found: !*
```

While this makes `rm` safe, it is certainly not what the user intended. In **zsh**, you must use a shell function for this:

```
% unalias rm
% rm () { mv $* /tmp/wastebasket }
% rm foo.dvi
% ls /tmp/wastebasket
foo.dvi
```

While this is much cleaner and easier to read (I hope you will agree), it is not csh-compatible. Therefore, a script to convert csh aliases and variables has been provided. You should only need to use it once, to convert all your csh aliases and parameters to **zsh** format:

```
% csh
csh> alias
l        ls -AF
more     less
on       last -2 !:1 ; who | grep !:1
csh> exit
% c2z >neat_zsh_aliases
% cat neat_zsh_aliases
alias l='ls -AF'
alias more='less'
on () { last -2 $1 ; who | grep $1 }
...
```

The first two aliases were converted to regular **zsh** aliases, while the third, since it needed to handle arguments, was converted to a function. `c2z` can convert most aliases to **zsh** format without any problems. However, if you're using some really arcane csh tricks, or if you have an alias with a name like `do` (which is reserved in **zsh**), you may have to fix some of the aliases by hand.

The `c2z` script checks your csh setup, and produces a list of **zsh** commands which replicate your aliases and parameter settings as closely as possible. You could include its output in your startup file, `.zshrc`.

History

There are several ways to manipulate history in **zsh**. One way is to use csh-style `!` history:

```
% /usr/local/bin/!:0 !-2*:s/foo/bar/ >>!$
```

If you don't want to use this, you can turn it off by typing `setopt nobanghist`. If you are afraid of accidentally executing the wrong command you can set the *HISTVERIFY* option. If this option is set, commands that result from history expansion will not be executed immediately, but will be put back into the editor buffer for further consideration.

If you're not familiar with `!` history, here follows some explanation. History substitutions always start with a `!`, commonly called "bang". After the `!` comes an (optional) designation of which "event" (command) to use, then a colon, and then a designation of what word of that command to use. For example, `!-`*n* refers to the command *n* commands ago.

```
% ls
foo  bar
% cd foo
% !-2
ls
baz  bam
```

No word designator was used, which means that the whole command referred to was repeated. Note that the shell will echo the result of the history substitution. The word designator can, among other things, be a number indicating the argument to use, where `0` is the command.

```
% /usr/bin/ls foo
foo
% !:0 bar
/usr/bin/ls bar
bar
```

In this example, no event designator was used, which tells **zsh** to use the previous command. A `$` specifies the last argument

```
% mkdir /usr/local/lib/emacs/site-lisp/calc
% cd !:$
cd /usr/local/lib/emacs/site-lisp/calc
```

If you use more words of the same command, only the first `!` needs an event designator.

```
% make prig >> make.log
make: *** No rule to make target 'prig'.  Stop.
% cd src
% !-2:0 prog >> !:$
make prog >> make.log
```

This is different from csh, where a bang with no event designator always refers to the previous command. If you actually like this behaviour, set the *CSHJUNKIEHISTORY* option.

```
% setopt cshjunkiehistory
% !-2:0 prog2 >> !:$
make prog2 >> cshjunkiehistory
```

Another way to use history is to use the `fc` command. For example, if you type an erroneous command:

```
% for i in 'cat /etc/clients'
 do
 rpu $i
 done
zsh: command not found: rpu
zsh: command not found: rpu
zsh: command not found: rpu
...
```

typing `fc` will execute an editor on this command, allowing you to fix it. (The default editor is `vi`, by the way, not `ed`).

```
% fc
49
/rpu/s//rup/p
 rup $i
w
49
q
for i in `cat /etc/clients`
 do
 rup $i
 done
        beam    up  2 days, 10:17,    load average: 0.86, 0.80, 0.50
         bow    up  4 days,  8:41,    load average: 0.91, 0.80, 0.50
        burn    up          17:18,    load average: 0.91, 0.80, 0.50
       burst    up  9 days,  1:49,    load average: 0.95, 0.80, 0.50
         tan    up          11:14,    load average: 0.91, 0.80, 0.50
       bathe    up  3 days, 17:49,    load average: 1.84, 1.79, 1.50
        bird    up  1 day,   9:13,    load average: 1.95, 1.82, 1.51
      bonnet    up  2 days, 21:18,    load average: 0.93, 0.80, 0.50
...
```

A variant of the `fc` command is `r`, which redoes the last command, with optional changes:

```
% echo foo
foo
% r
echo foo
foo

% echo foo
foo
% r foo=bar
echo bar
bar
```

Command Line Editing

zsh's command line editor, **ZLE**, is quite powerful. It is designed to emulate either emacs or vi; the default is emacs. To set the bindings for vi mode, type `bindkey -v`. If your **EDITOR** or **VISUAL** environment variable is vi, **zsh** will use vi emulation by default. You can then switch to emacs mode with `bindkey -e`.

In addition to basic editing, the shell allows you to recall previous lines in the history. In emacs mode, this is done with *^P* (control-P) or (on many terminals) with the cursor-up key:

```
% ls ~
-            README      file      mail        pub      tmp
Mailboxes    bin         func      nicecolors  scr      zsh
News         etc         iris      notes       src
% echo foobar
foobar
% ^P
% echo foobar^P
% ls ~_
```

Pressing *^P* once brings up the previous line (`echo foobar`); pressing it again brings up the line before that (`ls ~`). The cursor is left at the end of the line, allowing you to edit the line if desired before executing it. In many cases, **ZLE** eliminates the need for the `fc` command, since it is powerful enough to handle even multiline commands:

```
% for i in a b c d e
> do
> echo $i
> done
a
b
c
d
e
% ^P
% for i in a b c d e
 do
 echo $i
 done_
```

Now you can just move up to the part you want to change...

```
% for i in a b c d e
 do
 echo $i
 done
```

change it, and execute the new command.

```
% for i in f g h i j
 do
 echo $i
 done
f
g
h
i
j
```

Also, you can search the history for a certain command using *ESC-P*, this will look for the last command that started with the (part of the) word at the beginning of the current line. Hitting *ESC-P* another time gets you the command before that, etc.

```
% set ESC-P
% setopt autolist ESC-P
% setopt nocorrect_
```

Another way is to do an incremental search, emacs-style:

```
% ^R
% _
i-search:

% ls /usr/bin
i-search: l

% date > foofile.c
i-search: le
```

Suppose you have retrieved an old history event in one of these ways and would like to execute several consecutive old commands starting with this one. `^O` will execute the current command and then put the next command from the history into the editor buffer. Typing `^O` several times will therefore reexecute several consecutive commands from the history. Of course, you can edit some of those commands in between.

In addition to completion (see below), *TAB* performs expansion if possible.

```
% ls *.cTAB
% ls foofile.c fortune.c rnd.c strfile.c unstr.c_
```

For example, suppose you have a bunch of weird files in an important directory:

```
% ls
  * * *        ; & % $??foo  dspfok        foo.c
  !"foo"!        ' \ '          foo           rrr
```

You want to remove them, but you don't want to damage `foo.c`. Here is one way to do this:

```
% rm *TAB
% rm \ \ \*\ \*\ \*\ \ \  \!\"foo\"\! \;\ \&\ %\ \$'
'foo \'\ \\\ \' dspfok foo foo.c rrr_
```

When you expand *, **zsh** inserts the names of all the files into the editing buffer, with proper shell quoting. Now, just move back and remove `foo.c` from the buffer:

```
% rm \ \ \*\ \*\ \*\ \ \  \!\"foo\"\! \;\ \&\ %\ \$'
'foo \'\ \\\ \' dspfok foo rrr
```

and press return. Everything except `foo.c` will be deleted from the directory. If you do not want to actually expand the current word, but would like to see what the matches are, type `^Xg`.

```
% rm f*^Xg
foo    foo.c
% rm f*_
```

Here's another trick; let's say you have typed this command in:

```
% gcc -o x.out foob.c -g -Wpointer-arith -Wtrigraphs_
```

and you forget which library you want. You need to escape out for a minute and check by typing `ls /usr/lib`, or some other such command; but you don't want to retype the whole command again, and you can't press return now because the current command is incomplete. In **zsh**, you can put the line on the *buffer stack*, using *ESC-Q*, and type some other commands. The next time a prompt is printed, the `gcc` line will be popped off the stack and put in the editing buffer automatically; you can then enter the proper library name and press return (or, *ESC-Q* again and look for some other libraries whose names you forgot).

A similar situation: what if you forget the option to gcc that finds bugs using AI techniques? You could either use *ESC-Q* again, and type `man gcc`, or you could press *ESC-H*, which essentially does the same thing; it puts the current line on the buffer stack, and executes the command `run-help gcc`, where `run-help` is an alias for `man`.

Another interesting command is *ESC-A*. This executes the current line, but retains it in the buffer, so that it appears again when the next prompt is printed. Also, the cursor stays in the same place. This is useful for executing a series of similar commands:

```
% cc grok.c -g -lc -lgl -lsun -lmalloc -Bstatic -o b.out
% cc fubar.c -g -lc -lgl -lsun -lmalloc -Bstatic -o b.out
% cc fooble.c -g -lc -lgl -lsun -lmalloc -Bstatic -o b.out
```

The *ESC-'* command is useful for managing the shell's quoting conventions. Let's say you want to print this string:

```
don't do that; type 'rm -rf \*', with a \ before the *.
```

All that is necessary is to type it into the editing buffer:

```
% don't do that; type 'rm -rf \*', with a \ before the *.
```

press *ESC-'* (escape-quote):

```
% 'don'\''t do that; type '\''rm -rf \*'\'', with a \ before the *.'
```

then move to the beginning and add the `echo` command.

```
% echo 'don'\''t do that; type '\''rm -rf \*'\'', with a \ before the *.'
don't do that; type 'rm -rf \*', with a \ before the *.
```

Let's say you want to create an alias to do this `echo` command. This can be done by recalling the line with *^P* and pressing *ESC-'* again:

```
% 'echo '\''don'\''\'\'''\''t do that; type '\''\'\'''\''rm -rf
\*'\''\'\'''\'', with a \ before the *.'\'''
```

and then move to the beginning and add the command to create an alias.

```
% alias zoof='echo '\''don'\''\'\'''\''t do that; type '\''\'\'''\''rm
-rf \*'\''\'\'''\'', with a \ before the *.'\'''
% zoof
don't do that; type 'rm -rf \*', with a \ before the *.
```

If one of these fancy editor commands changes your command line in a way you did not intend, you can undo changes with `^_`, if you can get it out of your keyboard, or `^X^U`, otherwise.

Another use of the editor is to edit the value of variables. For example, an easy way to change your path is to use the `vared` command:

```
% vared PATH
> /u/pfalstad/scr:/u/pfalstad/bin/sun4:/u/maruchck/scr:/u/subbarao/bin:/u/maruc
hck/bin:/u/subbarao/scripts:/usr/princeton/bin:/usr/ucb:/usr/bin:/bin:/usr/host
s:/usr/princeton/bin/X11:/./usr/lang:/./usr/etc:/./etc
```

You can now edit the path. When you press return, the contents of the edit buffer will be assigned to **PATH**.

Completion

Another great **zsh** feature is completion. If you hit *TAB*, **zsh** will complete all kinds of stuff. Like commands or filenames:

```
% compTAB
% compress _

% ls nicTAB
% ls nicecolors _

% ls /usr/prTAB
% ls /usr/princeton/_

% ls -l =comTAB
% ls -l =compress _
```

If the completion is ambiguous, the editor will beep. If you find this annoying, you can set the *NOLISTBEEP* option. Completion can even be done in the middle of words. To use this, you will have to set the *COMPLETEINWORD* option:

```
% setopt completeinword
% ls /usr/ptonTAB
% ls /usr/princeton/
% setopt alwaystoend
% ls /usr/ptonTAB
% ls /usr/princeton/_
```

You can list possible completions by pressing *^D*:

```
% ls /vmuTAB —beep—
% ls /vmunix_
% ls /vmunix^D
vmunix                       vmunix.old
vmunix.new.kernelmap.old     vmunix.org
```

Or, you could just set the *AUTOLIST* option:

```
% setopt autolist
% ls /vmuTAB —beep—
vmunix                       vmunix.old
vmunix.new.kernelmap.old     vmunix.org
% ls /vmunix_
```

If you like to see the types of the files in these lists, like in `ls -F`, you can set the *LISTTYPES* option. Together with *AUTOLIST* you can use *LISTAMBIGUOUS*. This will only list the possibilities if there is no unambiguous part to add:

```
% setopt listambiguous
% ls /vmuTAB —beep—
% ls /vmunix_TAB —beep—
vmunix                    vmunix.old
vmunix.new.kernelmap.old  vmunix.org
```

If you don't want several of these listings to scroll the screen so much, the *ALWAYSLASTPROMPT* option is useful. If set, you can continue to edit the line you were editing, with the completion listing appearing beneath it.

Another interesting option is *MENUCOMPLETE*. This affects the way *TAB* works. Let's look at the /vmunix example again:

```
% setopt menucomplete
% ls /vmuTAB
% ls /vmunixTAB
% ls /vmunix.new.kernelmap.oldTAB
% ls /vmunix.old_
```

Each time you press *TAB*, it displays the next possible completion. In this way, you can cycle through the possible completions until you find the one you want.

The *AUTOMENU* option makes a nice compromise between this method of completion and the regular method. If you set this option, pressing *TAB* once completes the unambiguous part normally, pressing the *TAB* key repeatedly after an ambiguous completion will cycle through the possible completions.

Another option you could set is *RECEXACT*, which causes exact matches to be accepted, even if there are other possible completions:

```
% setopt recexact
% ls /vmuTAB —beep—
vmunix                    vmunix.old
vmunix.new.kernelmap.old  vmunix.org
% ls /vmunix_TAB
% ls /vmunix _
```

To facilitate the typing of pathnames, a slash will be added whenever a directory is completed. Some computers don't like the spurious slashes at the end of directory names. In that case, the *AUTOREMOVESLASH* option comes to rescue. It will remove these slashes when you type a space or return after them.

The *fignore* variable lists suffixes of files to ignore during completion.

```
% ls fooTAB —beep—
foofile.c  foofile.o
% fignore=( .o \~ .bak .junk )
% ls fooTAB
% ls foofile.c _
```

Since foofile.o has a suffix that is in the fignore list, it was not considered a possible completion of foo.

Username completion is also supported:

```
% ls ~pfalTAB
% ls ~pfalstad/_
```

and parameter name completion:

```
% echo $ORGTAB
% echo $ORGANIZATION _
% echo ${ORGTAB
% echo ${ORGANIZATION _
```

Note that in the last example a space is added after the completion as usual. But if you want to add a colon or closing brace, you probably don't want this extra space. Setting the

AUTOPARAMKEYS option will automatically remove this space if you type a colon or closing brace after such a completion.

There is also option completion:

```
% setopt noclTAB
% setopt noclobber _
```

and binding completion:

```
% bindkey '^X^X' puTAB
% bindkey '^X^X' push-line _
```

The `compctl` command is used to control completion of the arguments of specific commands. For example, to specify that certain commands take other commands as arguments, you use `compctl -c`:

```
% compctl -c man nohup
% man uptTAB
% man uptime _
```

To specify that a command should complete filenames, you should use `compctl -f`. This is the default. It can be combined with `-c`, as well.

```
% compctl -cf echo
% echo uptTAB
% echo uptime _

% echo foTAB
% echo foo.c
```

Similarly, use `-o` to specify options, `-v` to specify variables, and `-b` to specify bindings.

```
% compctl -o setopt unsetopt
% compctl -v typeset vared unset export
% compctl -b bindkey
```

You can also use `-k` to specify a custom list of keywords to use in completion. After the `-k` comes either the name of an array or a literal array to take completions from.

```
% ftphosts=(ftp.uu.net wuarchive.wustl.edu)
% compctl -k ftphosts ftp
% ftp wuTAB
% ftp wuarchive.wustl.edu _

% compctl -k '(cpirazzi subbarao sukthnkr)' mail finger
% finger cpTAB
% finger cpirazzi _
```

To better specify the files to complete for a command, use the `-g` option which takes any glob pattern as an argument. Be sure to quote the glob patterns as otherwise they will be expanded when the `compctl` command is run.

```
% ls
letter.tex  letter.dvi  letter.aux  letter.log  letter.toc
% compctl -g '*.tex' latex
% compctl -g '*.dvi' xdvi dvips
% latex lTAB
% latex letter.tex _
% xdvi lTAB
% xdvi letter.dvi _
```

Glob patterns can include qualifiers within parentheses. To rmdir only directories and cd to directories and symbolic links pointing to them:

```
% compctl -g '*(-/)' cd
% compctl -g '*(/)' rmdir
```

RCS users like to run commands on files which are not in the current directory, but in the RCS subdirectory where they all get `,v` suffixes. They might like to use

```
% compctl -g 'RCS/*(:t:s/\,v//)' co rlog rcs
% ls RCS
builtin.c,v  lex.c,v      zle_main.c,v
% rlog buTAB
% rlog builtin.c _
```

The :t modifier keeps only the last part of the pathname and the :s/\,v// will replace any ,v by nothing.

The -s flag is similar to -g, but it uses all expansions, instead of just globbing, like brace expansion, parameter substitution and command substitution.

```
% compctl -s '$(setopt)' unsetopt
```

will only complete options which are actually set to be arguments to unsetopt.

Sometimes a command takes another command as its argument. You can tell **zsh** to complete commands as the first argument to such a command and then use the completion method of the second command. The -l flag with a null-string argument is used for this.

```
% compctl -l '' nohup exec
% nohup compTAB
% nohup compress _
% nohup compress filTAB
% nohup compress filename _
```

Sometimes you would like to run really complicated commands to find out what the possible completions are. To do this, you can specify a shell function to be called that will assign the possible completions to a variable called reply. Note that this variable must be an array. Here's another (much slower) way to get the completions for co and friends:

```
% function getrcs {
> reply=()
> for i in RCS/*
>   do
>   reply=($reply[*] $(basename $i ,v))
>   done
> }
% compctl -K getrcs co rlog rcs
```

Some command arguments use a prefix that is not a part of the things to complete. The kill builtin command takes a signal name after a -. To make such a prefix be ignored in the completion process, you can use the -P flag.

```
% compctl -P - -k signals kill
% kill -HTAB
% kill -HUP _
```

TeX is usually run on files ending in .tex, but also sometimes on other files. It is somewhat annoying to specify that the arguments of TeX should end in .tex and then not be able to complete these other files. Therefore you can specify things like "Complete to files ending in .tex if available, otherwise complete to any filename.". This is done with *xor*ed completion:

```
% compctl -g '*.tex' + -f tex
```

The + tells the editor to only take the next thing into account if the current one doesn't generate any matches. If you have not changed the default completion, the above example is in fact equivalent to

```
% compctl -g '*.tex' + tex
```

as a lone + at the end is equivalent to specifying the default completion after the +. This form of completion is also frequently used if you want to run some command only on a certain type of files, but not necessarily in the current directory. In this case you will want to complete both files of this type and directories. Depending on your preferences you can use either of

```
% compctl -g '*.ps' + -g '*(-/)' ghostview
% compctl -g '*.ps *(-/)' ghostview
```

where the first one will only complete directories (and symbolic links pointing to directories) if no postscript file matches the already typed part of the argument.

Extended completion

If you play with completion, you will soon notice that you would like to specify what to complete, depending on what flags you give to the command and where you are on the command line. For example, a command could take any filename argument after a `-f` flag, a username after a `-u` flag and an executable after a `-x` flag. This section will introduce you to the ways to specify these things. To many people it seems rather difficult at first, but taking the trouble to understand it can save you lots of typing in the end. Even I keep being surprised when **zsh** manages to complete a small or even empty prefix to the right file in a large directory.

To tell **zsh** about these kinds of completion, you use "extended completion" by specifying the `-x` flag to compctl. The `-x` flag takes a list of patterns/flags pairs. The patterns specify when to complete and the flags specify what. The flags are simply those mentioned above, like `-f` or `-g` *glob pattern*.

As an example, the `r[`*string1*`,`*string2*`]` pattern matches if the cursor is after something that starts with *string1* and before something that starts with *string2*. The *string2* is often something that you do not want to match anything at all.

```
% ls
fool   barl   foo.Z  bar.Z
% compctl -g '^*.Z' -x 'r[-d,---]' -g '*.Z' -- compress
% compress fTAB
% compress fool _
% compress -d fTAB
% compress -d foo.Z _
```

In the above example, if the cursor is after the `-d` the pattern will match and therefore **zsh** uses the `-g *.Z` flag that will only complete files ending in `.Z`. Otherwise, if no pattern matches, it will use the flags before the `-x` and in this case complete every file that does not end in `.Z`.

The `s[`*string*`]` pattern matches if the current word starts with *string*. The *string* itself is not considered to be part of the completion.

```
% compctl -x 's[-]' -k signals -- kill
% kill -HTAB
% kill -HUP _
```

The `tar` command takes a tar file as an argument after the `-f` option. The `c[`*offset*`,`*string*`]` pattern matches if the word in position *offset* relative to the current word is *string*. More in particular, if *offset* is -1, it matches if the previous word is *string*. This suggests

```
% compctl -f -x 'c[-1,-f]' -g '*.tar' -- tar
```

But this is not enough. The `-f` option could be the last of a longer string of options. `C[...,...]` is just like `c[...,...]`, except that it uses glob-like pattern matching for *string*. So

```
% compctl -f -x 'C[-1,-*f]' -g '*.tar' -- tar
```

will complete tar files after any option string ending in an `f`. But we'd like even more. Old versions of tar used all options as the first argument, but without the minus sign. This might be inconsistent with option usage in all other commands, but it is still supported by newer versions of `tar`. So we would also like to complete tar files if the first argument ends in an `f` and we're right behind it.

We can 'and' patterns by putting them next to each other with a space between them. We can 'or' these sets by putting comma's between them. We will also need some new patterns. `p[`*num*`]` will match if the current argument (the one to be completed) is the *num*th argument. `W[`*index*`,`*pattern*`]` will match if the argument in place *index* matches the *pattern*. This gives us

```
% compctl -f -x 'C[-1,-*f] , W[1,*f] p[2]' -g '*.tar' -- tar
```

In words: If the previous argument is an option string that ends in an `f`, or the first argument ended in an `f` and it is now the second argument, then complete only filenames ending in `.tar`.

All the above examples used only one set of patterns with one completion flag. You can use several of these pattern/flag pairs separated by a -. The first matching pattern will be used. Suppose you have a version of tar that supports compressed files by using a -Z option. Leaving the old tar syntax aside for a moment, we would like to complete files ending in .tar.Z if a -Z option has been used and files ending in .tar otherwise, all this only after a -f flag. Again, the -Z can be alone or it can be part of a longer option string, perhaps the same as that of the -f flag. Here's how to do it; note the backslash and the secondary prompt which are not part of the compctl command.

```
% compctl -f -x 'C[-1,-*Z*f] , R[-*Z*,---] C[-1,-*f]' -g '*.tar.Z' - \
> 'C[-1,-*f]' -g '*.tar' -- tar
```

The first pattern set tells us to match if either the previous argument was an option string including a Z and ending in an f or there was an option string with a Z somewhere and the previous word was any option string ending in an f. If this is the case, we need a compressed tar file. Only if this is not the case the second pattern set will be considered. By the way, R[*pattern1*,*pattern2*] is just like r[...,...] except that it uses pattern matching with shell metacharacters instead of just strings.

You will have noticed the -- before the command name. This ends the list of pattern/flag pairs of -x. It is usually used just before the command name, but you can also use an extended completion as one part of a list of xored completions, in which case the -- appears just before one of the + signs.

Note the difference between using extended completion as part of a list of xored completions as in

```
% ls
foo  bar
% compctl -x 'r[-d,---]' -g '*.Z' -- + -g '^*.Z' compress
% compress -d fTAB
% compress -d foo _
```

and specifying something before the -x as in

```
% compctl -g '^*.Z' -x 'r[-d,---]' -g '*.Z' -- compress
% compress -d fTAB
% compress -d f_
```

In the first case, the alternative glob pattern (^*.Z) will be used if the first part does not generate any possible completions, while in the second case the alternative glob pattern will only be used if the r[...] pattern doesn't match.

Bindings

Each of the editor commands we have seen was actually a function bound by default to a certain key. The real names of the commands are:

expand-or-complete	*TAB*
push-line	*ESC-Q*
run-help	*ESC-H*
accept-and-hold	*ESC-A*
quote-line	*ESC-'*

These bindings are arbitrary; you could change them if you want. For example, to bind accept-line to *^Z*:

```
% bindkey '^Z' accept-line
```

Another idea would be to bind the delete key to delete-char; this might be convenient if you use *^H* for backspace.

```
% bindkey '^?' delete-char
```

Or, you could bind *^X^H* to run-help:

```
% bindkey '^X^H' run-help
```

Other examples:

```
% bindkey '^X^Z' universal-argument
% bindkey ' ' magic-space
% bindkey -s '^T' 'uptime
> '
% bindkey '^Q' push-line-or-edit
```

`universal-argument` multiplies the next command by 4. Thus *^X^Z^W* might delete the last four words on the line. If you bind space to `magic-space`, then csh-style history expansion is done on the line whenever you press the space bar.

Something that often happens is that I am typing a multiline command and discover an error in one of the previous lines. In this case, `push-line-or-edit` will put the entire multiline construct into the editor buffer. If there is only a single line, it is equivalent to `push-line`.

The `-s` flag to `bindkey` specifies that you are binding the key to a string, not a command. Thus `bindkey -s '^T' 'uptime\n'` lets you VMS lovers get the load average whenever you press *^T*.

If you have a NeXT keyboard, the one with the | and \ keys very inconveniently placed, the following bindings may come in handy:

```
% bindkey -s '\e/' '\\'
% bindkey -s '\e=' '|'
```

Now you can type *ALT-/* to get a backslash, and *ALT-=* to get a vertical bar. This only works inside **zsh**, of course; `bindkey` has no effect on the key mappings inside `talk` or `mail`, etc.

Some people like to bind `^S` and `^Q` to editor commands. Just binding these has no effect, as the terminal will catch them and use them for flow control. You could unset them as stop and start characters, but most people like to use these for external commands. The solution is to set the *NOFLOWCONTROL* option. This will allow you to bind the start and stop characters to editor commands, while retaining their normal use for external commands.

Parameter Substitution

In **zsh**, parameters are set like this:

```
% foo=bar
% echo $foo
bar
```

Spaces before or after the = are frowned upon:

```
% foo = bar
zsh: command not found: foo
```

Also, `set` doesn't work for setting parameters:

```
% set foo=bar
% set foo = bar
% echo $foo

%
```

Note that no error message was printed. This is because both of these commands were perfectly valid; the `set` builtin assigns its arguments to the *positional parameters* (`$1`, `$2`, etc.).

```
% set foo=bar
% echo $1
foo=bar
% set foo = bar
% echo $3 $2
bar =
```

If you're really intent on using the csh syntax, define a function like this:

```
% set () {
>     eval "$1$2$3"
> }
% set foo = bar
% set fuu=brrr
% echo $foo $fuu
bar brrr
```

But then, of course you can't use the form of `set` with options, like `set -F` (which turns off filename generation). Also, the `set` command by itself won't list all the parameters like it should. To get around that you need a `case` statement:

```
% set () {
>     case $1 in
>     -*|+*|'') builtin set $* ;;
>     *) eval "$1$2$3" ;;
>     esac
> }
```

For the most part, this should make csh users happy.

The following sh-style operators are supported in **zsh**:

```
% unset null
% echo ${foo-xxx}
bar
% echo ${null-xxx}
xxx
% unset null
% echo ${null=xxx}
xxx
% echo $null
xxx
% echo ${foo=xxx}
bar
% echo $foo
bar
% unset null
% echo ${null+set}

% echo ${foo+set}
set
```

Also, csh-style `:` modifiers may be appended to a parameter substitution.

```
% echo $PWD
/home/learning/pf/zsh/zsh2.00/src
% echo $PWD:h
/home/learning/pf/zsh/zsh2.00
% echo $PWD:h:h
/home/learning/pf/zsh
% echo $PWD:t
src
% name=foo.c
% echo $name
foo.c
% echo $name:r
foo
% echo $name:e
c
```

The equivalent constructs in ksh (which are also supported in **zsh**) are a bit more general and easier to remember. When the shell expands `${foo#`*pat*`}`, it checks to see if *pat* matches a substring at the beginning of the value of `foo`. If so, it removes that portion of `foo`, using the shortest possible match. With `${foo##`*pat*`}`, the longest possible match is removed. `${foo%`*pat*`}` and `${foo%%`*pat*`}` remove the match from the end. Here are the ksh equivalents of the `:` modifiers:

```
% echo ${PWD%/*}
/home/learning/pf/zsh/zsh2.00
% echo ${PWD%/*/*}
/home/learning/pf/zsh
% echo ${PWD##*/}
src
% echo ${name%.*}
foo
% echo ${name#*.}
c
```

zsh also has upper/lowercase modifiers:

```
% xx=Test
% echo $xx:u
TEST
% echo $xx:l
test
```

and a substitution modifier:

```
% echo $name:s/foo/bar/
bar.c
% ls
foo.c    foo.h    foo.o    foo.pro
% for i in foo.*; mv $i $i:s/foo/bar/
% ls
bar.c    bar.h    bar.o    bar.pro
```

There is yet another syntax to modify substituted parameters. You can add certain modifiers in parentheses after the opening brace like:

`${(`*modifiers*`)`*parameter*`}`

For example, `o` sorts the words resulting from the expansion:

```
% echo ${path}
/usr/bin /usr/bin/X11 /etc
% echo ${(o)path}
/etc /usr/bin /usr/bin/X11
```

One possible source of confusion is the fact that in **zsh**, the result of parameter substitution is *not* split into words. Thus, this will not work:

```
% srcs='glob.c exec.c init.c'
% ls $srcs
glob.c exec.c init.c not found
```

This is considered a feature, not a bug. If splitting were done by default, as it is in most other shells, functions like this would not work properly:

```
$ ll () { ls -F $* }
$ ll 'fuu bar'
fuu not found
bar not found

% ll 'fuu bar'
fuu bar not found
```

Of course, a hackish workaround is available in sh (and **zsh**):

```
% setopt shwordsplit
% ll () { ls -F "$@" }
% ll 'fuu bar'
fuu bar not found
```

If you like the sh behaviour, **zsh** can accomodate you:

```
% ls ${=srcs}
exec.c  glob.c  init.c
% setopt shwordsplit
% ls $srcs
exec.c  glob.c  init.c
```

Another way to get the `$srcs` trick to work is to use an array:

```
% unset srcs
% srcs=( glob.c exec.c init.c )
% ls $srcs
exec.c  glob.c  init.c
```

or an alias:

```
% alias -g SRCS='exec.c glob.c init.c'
% ls SRCS
exec.c  glob.c  init.c
```

Another option that modifies parameter expansion is *RCEXPANDPARAM*:

```
% echo foo/$srcs
foo/glob.c exec.c init.c
% setopt rcexpandparam
% echo foo/$srcs
foo/glob.c foo/exec.c foo/init.c
% echo foo/${^srcs}
foo/glob.c foo/exec.c foo/init.c
% echo foo/$^srcs
foo/glob.c foo/exec.c foo/init.c
```

Shell Parameters

The shell has many predefined parameters that may be accessed. Here are some examples:

```
% sleep 10 &
[1] 3820
% echo $!
3820
% set a b c
% echo $#
3
% echo $ARGC
3
% ( exit 20 ) ; echo $?
20
% false; echo $status
1
```

(`$?` and `$status` are equivalent.)

```
% echo $HOST $HOSTTYPE
dendrite sun4
% echo $UID $GID
701 60
% cd /tmp
% cd /home
% echo $PWD $OLDPWD
/home /tmp
% ls $OLDPWD/.getwd
/tmp/.getwd
```

`~+` and `~-` are short for `$PWD` and `$OLDPWD`, respectively.

```
% ls ~-/.getwd
/tmp/.getwd
% ls -d ~+/learning
/home/learning
% echo $RANDOM
4880
% echo $RANDOM
11785
% echo $RANDOM
2062
% echo $TTY
/dev/ttyp4
% echo $VERSION
zsh v2.00.03
% echo $USERNAME
pf
```

The `cdpath` variable sets the search path for the `cd` command. If you do not specify . somewhere in the path, it is assumed to be the first component.

```
% cdpath=( /usr ~ ~/zsh )
% ls /usr
5bin         dict         lang         net          sccs         sys
5include     etc          lector       nserve       services     tmp
5lib         export       lib          oed          share        ucb
adm          games        local        old          skel         ucbinclude
bin          geac         lost+found   openwin      spool        ucblib
boot         hosts        macsyma_417  pat          src          xpg2bin
demo         include      man          princeton    stand        xpg2include
diag         kvm          mdec         pub          swap         xpg2lib
% cd spool
/usr/spool
% cd bin
/usr/bin
% cd func
~/func
% cd
% cd pub
% pwd
/u/pfalstad/pub
% ls -d /usr/pub
/usr/pub
```

PATH and **path** both set the search path for commands. These two variables are equivalent, except that one is a string and one is an array. If the user modifies **PATH**, the shell changes **path** as well, and vice versa.

```
% PATH=/bin:/usr/bin:/tmp:.
% echo $path
/bin /usr/bin /tmp .
% path=( /usr/bin . /usr/local/bin /usr/ucb )
% echo $PATH
/usr/bin:.:/usr/local/bin:/usr/ucb
```

The same is true of **CDPATH** and **cdpath**:

```
% echo $CDPATH
/usr:/u/pfalstad:/u/pfalstad/zsh
% CDPATH=/u/subbarao:/usr/src:/tmp
% echo $cdpath
/u/subbarao /usr/src /tmp
```

In general, predefined parameters with names in all lowercase are arrays; assignments to them take the form:

name=(*elem* ...\0)

Predefined parameters with names in all uppercase are strings. If there is both an array and a string version of the same parameter, the string version is a colon-separated list, like **PATH**.

HISTFILE is the name of the history file, where the history is saved when a shell exits.

```
% zsh
phoenix% HISTFILE=/tmp/history
phoenix% SAVEHIST=20
phoenix% echo foo
foo
phoenix% date
Fri May 24 05:39:35 EDT 1991
phoenix% uptime
  5:39am  up 4 days, 20:02,  40 users,  load average: 2.30, 2.20, 2.00
phoenix% exit
% cat /tmp/history
HISTFILE=/tmp/history
SAVEHIST=20
echo foo
date
uptime
exit
% HISTSIZE=3
% history
   28  rm /tmp/history
   29  HISTSIZE=3
   30  history
```

If you have several incantations of **zsh** running at the same time, like when using the X window system, it might be preferable to append the history of each shell to a file when a shell exits instead of overwriting the old contents of the file. You can get this behaviour by setting the *APPENDHISTORY* option.

In **zsh**, if you say

```
% >file
```

the command `cat` is normally assumed:

```
% >file
foo!
^D
% cat file
foo!
```

Thus, you can view a file simply by typing:

```
% <file
foo!
```

However, this is not csh or sh compatible. To correct this, change the value of the parameter **NULLCMD**, which is `cat` by default.

```
% NULLCMD=:
% >file
% ls -l file
-rw-r--r--  1 pfalstad        0 May 24 05:41 file
```

If NULLCMD is unset, the shell reports an error if no command is specified (like csh).

```
% unset NULLCMD
% >file
zsh: redirection with no command
```

Actually, **READNULLCMD** is used whenever you have a null command reading input from a single file. Thus, you can set **READNULLCMD** to `more` or `less` rather than `cat`. Also, if you set **NULLCMD** to `:` for sh compatibility, you can still read files with `< file` if you leave

READNULLCMD set to more.

Prompting

The default prompt for **zsh** is:

```
phoenix% echo $PROMPT
%m%#
```

The %m stands for the short form of the current hostname, and the %# stands for a % or a #, depending on whether the shell is running as root or not. **zsh** supports many other control sequences in the **PROMPT** variable.

```
% PROMPT='%/> '
/u/pfalstad/etc/TeX/zsh>

% PROMPT='%~> '
~/etc/TeX/zsh>

% PROMPT='%h %~> '
6 ~/etc/TeX/zsh>
```

%h represents the number of current history event.

```
% PROMPT='%h %~ %M> '
10 ~/etc/TeX/zsh apple-gunkies.gnu.ai.mit.edu>

% PROMPT='%h %~ %m> '
11 ~/etc/TeX/zsh apple-gunkies>

% PROMPT='%h %t> '
12 6:11am>

% PROMPT='%n %w tty%l>'
pfalstad Fri 24 ttyp0>
```

PROMPT2 is used in multiline commands, like for-loops. The %_ escape sequence was made especially for this prompt. It is replaced by the kind of command that is being entered.

```
% PROMPT2='%_> '
% for i in foo bar
for>

% echo 'hi
quote>
```

Also available is the **RPROMPT** parameter. If this is set, the shell puts a prompt on the *right* side of the screen.

```
% RPROMPT='%t'
%                                                    6:14am

% RPROMPT='%~'
%                                             ~/etc/TeX/zsh

% PROMPT='%l %T %m[%h] ' RPROMPT=' %~'
p0 6:15 phoenix[5]                            ~/etc/TeX/zsh
```

These special escape sequences can also be used with the -P option to print:

```
% print -P %h tty%l
15 ttyp1
```

The **POSTEDIT** parameter is printed whenever the editor exits. This can be useful for termcap tricks. To highlight the prompt and command line while leaving command output unhighlighted, try this:

```
% POSTEDIT='echotc se'
% PROMPT='%S%% '
```

Login/logout watching

You can specify login or logout events to monitor by setting the **watch** variable. Normally, this is done by specifying a list of usernames.

```
% watch=( pfalstad subbarao sukthnkr egsirer )
```

The `log` command reports all people logged in that you are watching for.

```
% log
pfalstad has logged on p0 from mickey.
pfalstad has logged on p5 from mickey.
% ...
subbarao has logged on p8 from phoenix.
% ...
subbarao has logged off p8 from phoenix.
% ...
sukthnkr has logged on p8 from dew.
% ...
sukthnkr has logged off p8 from dew.
```

If you specify hostnames with an `@` prepended, the shell will watch for all users logging in from the specified host.

```
% watch=( @mickey @phoenix )
% log
djthongs has logged on q2 from phoenix.
pfalstad has logged on p0 from mickey.
pfalstad has logged on p5 from mickey.
```

If you give a tty name with a `%` prepended, the shell will watch for all users logging in on that tty.

```
% watch=( %ttyp0 %console )
% log
root has logged on console from .
pfalstad has logged on p0 from mickey.
```

The format of the reports may also be changed.

```
% watch=( pfalstad gettes eps djthongs jcorr bdavis )
% log
jcorr has logged on tf from 128.112.176.3:0.
jcorr has logged on r0 from 128.112.176.3:0.
gettes has logged on p4 from yo:0.0.
djthongs has logged on pe from grumpy:0.0.
djthongs has logged on q2 from phoenix.
bdavis has logged on qd from BRUNO.
eps has logged on p3 from csx30:0.0.
pfalstad has logged on p0 from mickey.
pfalstad has logged on p5 from mickey.
% WATCHFMT='%n on tty%l from %M'
% log
jcorr on ttytf from 128.112.176.3:0.
jcorr on ttyr0 from 128.112.176.3:0.
gettes on ttyp4 from yo:0.0
djthongs on ttype from grumpy:0.0
djthongs on ttyq2 from phoenix.Princeto
bdavis on ttyqd from BRUNO.pppl.gov
eps on ttyp3 from csx30:0.0
pfalstad on ttyp0 from mickey.Princeton
pfalstad on ttyp5 from mickey.Princeton
% WATCHFMT='%n fm %m'
% log
jcorr fm 128.112.176.3:0
jcorr fm 128.112.176.3:0
gettes fm yo:0.0
djthongs fm grumpy:0.0
djthongs fm phoenix
bdavis fm BRUNO
eps fm csx30:0.0
pfalstad fm mickey
pfalstad fm mickey
% WATCHFMT='%n %a at %t %w.'
% log
jcorr logged on at 3:15pm Mon 20.
jcorr logged on at 3:16pm Wed 22.
gettes logged on at 6:54pm Wed 22.
djthongs logged on at 7:19am Thu 23.
djthongs logged on at 7:20am Thu 23.
bdavis logged on at 12:40pm Thu 23.
eps logged on at 4:19pm Thu 23.
pfalstad logged on at 3:39am Fri 24.
pfalstad logged on at 3:42am Fri 24.
```

If you have a `.friends` file in your home directory, a convenient way to make **zsh** watch for all your friends is to do this:

```
% watch=( $(< ~/.friends) )
% echo $watch
subbarao maruchck root sukthnkr ...
```

If watch is set to `all`, then all users logging in or out will be reported.

Options

Some options have already been mentioned; here are a few more:

Using the *AUTOCD* option, you can simply type the name of a directory, and it will become the current directory.

```
% cd /
% setopt autocd
% bin
% pwd
/bin
% ../etc
% pwd
/etc
```

With *CDABLEVARS*, if the argument to `cd` is the name of a parameter whose value is a valid directory, it will become the current directory.

```
% setopt cdablevars
% foo=/tmp
% cd foo
/tmp
```

CORRECT turns on spelling correction for commands, and the *CORRECTALL* option turns on spelling correction for all arguments.

```
% setopt correct
% sl
zsh: correct 'sl' to 'ls' [nyae]? y
% setopt correctall
% ls x.v11r4
zsh: correct 'x.v11r4' to 'X.V11R4' [nyae]? n
/usr/princton/src/x.v11r4 not found
% ls /etc/paswd
zsh: correct to '/etc/paswd' to '/etc/passwd' [nyae]? y
/etc/passwd
```

If you press `y` when the shell asks you if you want to correct a word, it will be corrected. If you press `n`, it will be left alone. Pressing `a` aborts the command, and pressing `e` brings the line up for editing again, in case you agree the word is spelled wrong but you don't like the correction.

Normally, a quoted expression may contain a newline:

```
% echo '
> foo
> '

foo

%
```

With *CSHJUNKIEQUOTES* set, this is illegal, as it is in csh.

```
% setopt cshjunkiequotes
% ls 'foo
zsh: unmatched '
```

GLOBDOTS lets files beginning with a `.` be matched without explicitly specifying the dot.

```
% ls -d *x*
Mailboxes
% setopt globdots
% ls -d *x*
.exrc          .pnewsexpert  .xserverrc
.mushexpert    .xinitrc      Mailboxes
```

HISTIGNOREDUPS prevents the current line from being saved in the history if it is the same as the previous one; *HISTIGNORESPACE* prevents the current line from being saved if it begins with a space.

```
% PROMPT='%h> '
39> setopt histignoredups
40> echo foo
foo
41> echo foo
foo
41> echo foo
foo
41> echo bar
bar
42> setopt histignorespace
43>  echo foo
foo
43>  echo fubar
fubar
43>  echo fubar
fubar
```

IGNOREBRACES turns off csh-style brace expansion.

```
% echo x{y{z,a},{b,c}d}e
xyze xyae xbde xcde
% setopt ignorebraces
% echo x{y{z,a},{b,c}d}e
x{y{z,a},{b,c}d}e
```

IGNOREEOF forces the user to type `exit` or `logout`, instead of just pressing *^D*.

```
% setopt ignoreeof
% ^D
zsh: use 'exit' to exit.
```

INTERACTIVECOMMENTS turns on interactive comments; comments begin with a `#`.

```
% setopt interactivecomments
% date # this is a comment
Fri May 24 06:54:14 EDT 1991
```

NOBEEP makes sure the shell never beeps.

NOCLOBBER prevents you from accidentally overwriting an existing file.

```
% setopt noclobber
% cat /dev/null >~/.zshrc
zsh: file exists: /u/pfalstad/.zshrc
```

If you really do want to clobber a file, you can use the `>!` operator. To make things easier in this case, the `>` is stored in the history list as a `>!`:

```
% cat /dev/null >! ~/.zshrc
% cat /etc/motd > ~/.zshrc
zsh: file exists: /u/pfalstad/.zshrc
% !!
cat /etc/motd >! ~/.zshrc
% ...
```

RCQUOTES lets you use a more elegant method for including single quotes in a singly quoted string:

```
% echo '"don'\''t do that."'
"don't do that."
% echo '"don''t do that."'
"dont do that."
% setopt rcquotes
% echo '"don''t do that."'
"don't do that."
```

Finally, *SUNKEYBOARDHACK* wins the award for the strangest option. If a line ends with `, and there are an odd number of them on the line, the shell will ignore the trailing `. This is provided for keyboards whose RETURN key is too small, and too close to the ' key.

```
% setopt sunkeyboardhack
% date'
Fri May 24 06:55:38 EDT 1991
```

Closing Comments

I (Bas de Bakker) would be happy to receive mail if anyone has any tricks or ideas to add to this document, or if there are some points that could be made clearer or covered more thoroughly. Please notify me of any errors in this document.

Table of Contents