INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY. CABLE CENTRATOM TRIESTE

The United Nations University

SMR/748 - 6

ICTP-INFN-UNU-MICROPROCESSOR LABORATORY
THIRD COURSE ON BASIC VLSI DESIGN TECHNIQUES
2 November - 16 December 1994

INTRODUCTION TO UNIX

Daniele BULFONE
Control Group
Sincrotrone Trieste
Padriciano 99
34012 Trieste
Italy

Third Course on Basic VLSI Design Techniques

ICTP-INFN-UNU-Microprocessor Laboratory
Trieste, Italy, 21 November - 16 December 1994

Introduction to UNIX

D. Bulfone
Sincrotrone Trieste, Padriciano 99, 34012 Trieste (Italy).

3rd Course on Basic VLSI Design Techniques ICTP, Trieste, 21 Nov/16 Dec 1994

1. Introduction

An operating system is a collection of programs that controls and organizes the resources of a computer system providing the user with a convenient and ready-to-use machine.
These resources consist of hardware components such as central processors, memory, disks, communication connections, terminals, printers, and the software programs that tell the computer to perform specific tasks.

Normally, personal computers have single-user operating systems (think of MS-DOS) -> only one person can use the system at a time, and the system can handle only one task at a time.

Other operating systems are multi-user, multi-tasking systems; several terminals are connected to the computer and the operating system manages access to the computer by a community of users. This kind of operating system allows the computer to perform many functions at the same time.

UNIX is a multi-user, multi-tasking operating system.

It also provides programs for editing text, sending e-mails, performing calculations and many other specialized functions that normally require separate application programs.

1.1 History & Versions of UNIX

1965 Bell Telephone Laboratories (from the AT&T group) joined an effort with MIT (Massachusetts Institute of Technology) to develop a new operating system called **Multics** (multi-user, big computation utilities, large data storage). Multics project never came to conclusion.

Members of the Computing Science Research Center at Bell Laboratories (like Ken Thompson, Dennis Ritchie) who were involved in the Multics project, started thinking of a single user operating system that could improve their programming environment.

1969 K. Thompson & D.Ritchie implement the new system on a PDP-7 machine from Digital. The development of programs (like the "space travel" simulator from Thomson) no longer needed a host machine as working environment but is made "on target". B. Kernighan calls the system **UNIX**, as a play on words referred to Multics. It is single user system, written in assembler.

1971 UNIX system moves to a PDP-11, for the supply of a text processing system.

1973 **UNIX multi-user** is written in "C", a step that was to have tremendous impact on its world-wide acceptance: the use of a high-level language makes UNIX implementations easily portable on different platforms. A new "C" compiler is rewritten using the "C" language itself, and can generate UNIX for different hosts, "just" by adapting the semantic of some low level routines.

At this time AT&T could not market computer products (because of an agreement with the Federal government), so it provided the UNIX system to the universities, where it was successively improved by students and researchers.

1977 The number of UNIX system sites is about 500, of which 125 are universities.
First porting on a non-PDP machine (Interdata 8/32).

With the growing popularity of microprocessors, other companies port the UNIX system to new machines, and enhance it in their own way, resulting in several variants of the basic system.

1977/82 Bell Laboratories/AT&T **UNIX System III**.

1981 University of California at Berkeley develops a UNIX system variant called **4.1 BSD** (Berkeley Standard Distribution), and starts using TCP/IP as network protocols.
XENIX, from Microsoft. First UNIX version available for all 16 bit microprocessors.

1983 Bell Laboratories/AT&T **UNIX System V**.
4.2 BSD for VAX machines (the most recent version is now 4.3 BSD).

1984 **100.000 UNIX** system installations: microprocessor, mainframes.

Linux is normally said to be System V and BSD compliant.

2. Entering the UNIX Environment

Before start using UNIX, the System Administrator has to set up a UNIX account for you (it is like your "home" in the "UNIX town").

Each user communicates with the computer from a terminal.

To get into the UNIX environment you have to **log in** at your terminal. Logging in does two things:
it identifies which user is at a specific terminal
it tells the computer that you are ready to begin working.

In order to log in:

1. Press the RETURN key until the login prompt appears on the screen.

2. Type in your login ID (usually your name or initials) in lowercase letter at the prompt, and press RETURN.
The system now prompts you to enter your password.

3. Type in your password (which is not displayed on the screen, for security reasons), and press RETURN.

If the login is successful, you will get some system messages and finally the UNIX system prompt.

2.1 The UNIX Shell

Once logged in, you are talking to a program called the **shell** (see fig. 1).

Note that the shell is just a program (like other UNIX utilities), written in the "C" language.
Different shell programs exist.

The default one is the so called **Bourne shell**, developed by Stephen Bourne at the AT&T laboratories. However, one of the most flexible and powerful shells is the so called **C shell**, developed by Bill Joy from the University of California at Berkeley. This is the one you will use.

You can use the shell in three ways:

1. Interactively, to execute UNIX commands.

2. In programming, to create new tools as shell programs, called *scripts* (see paragraph 7.4).

3. In customization, to define your working environment (see chapter 7).

Let's start from the first mode.

SYSTEM STRUCTURE

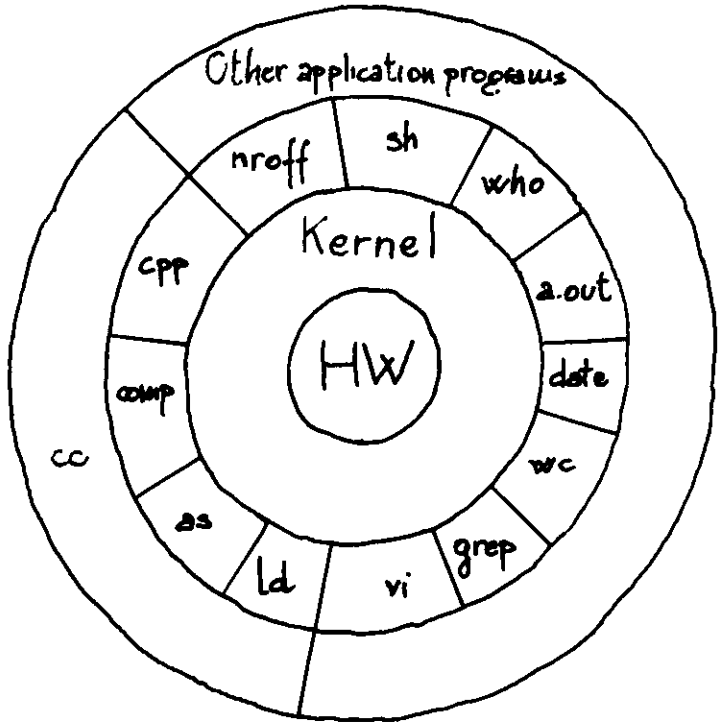


Fig 1. Architecture of UNIX System

The shell interprets the commands that you type, invokes the various programs that you ask for, and generally acts as a sort of buffer between you and the UNIX operating system proper.

When you are typing a command line, the shell simply collects your input from the keyboard. Pressing the RETURN key tells the system that you have finished entering text and that it can start executing the command.

The system prompt is the shell's way of saying that it is ready and waiting for you to type in a command.
When the system is finished running a command, the shell replies with a prompt to tell you that you can now type another command.
The shell uses "% " as default prompt, but this can be changed at will.

UNIX commands can be simple one-word entries or more complex in form and may require to specify certain arguments. An argument can be an option or a filename. The general format is:

command *option(s) filename(s)*

command *option(s) filename(s)*

General rules of the UNIX style for writing commands are:

• Commands are entered in lower case.

• Options modify the way in which a command works. They are often single letters prefixed with a minus (-) sign and set off by any number of spaces. If you make a mistake in specifying the options, some commands will display the allowable options.

• Multiple options in one command line can either be set off individually or combined after a single minus sign.

• The argument *filename* is the name of a file that you want to manipulate in some way.

• Spaces between commands, option(s) and filename(s) are important.

• Two commands can be entered on the same line, separated by a semicolon (;) and written on the same command line. Commands entered in this way are executed sequentially by UNIX.

What to do if you make a mistake in typing a command and press the RETURN key before you realize your mistake?

- The shell will give an error message (don't be afraid of error messages).

- Use the CTRL-C "interrupt character" to interrupt or cancel a command, in order to quit what you are doing.

Let's look at an example. The **ls** command displays a list of files. It can be used with or without arguments with different results. If you enter:

```
% ls
```

a list of filenames will be displayed on the screen. But if you enter:

```
% ls -l
```

there will be an entire line of information for each file. The **-l** option modifies the normal output of the **ls** command and lists files in the "long" format.
You can also ask for information about a particular file by adding a filename as a second argument. For example, to find out about a file called *ictp1*, you would type:

```
% ls -l ictp1
```

When you want to specify multiple options, you can write the command in one of the following equivalent forms:

```
% ls -a -l
% ls -al
```

Remember to type a space after the command name and before the minus sign that introduces the options.

Other simple UNIX commands:

date	Finds out today's date
who	Finds out logged in users

Logging out is the process of ending a UNIX session. Type the command **logout** followed by RETURN. Once successfully logged out, the login prompt should appear on the screen to indicate that you are no longer logged into the system and can leave the terminal to the next user.

3. Your UNIX Account

Like a home is used to store our things, can be accommodated to our needs and has its own address, the UNIX account provides

•a place in the UNIX file system where you can store your files,

• a customizable environment that you can tailor to your liking (see chapter 7),

• a login ID that identifies you uniquely, allowing to control access to your files, and to receive mail from other users.

3.1 The UNIX File System

A *file* is the unit of storage in UNIX, as in many other systems. A file can contain anything: a program, executable object code, text, etc.

All are just sequences of raw data until they are interpreted by the right program.

Files are organized into **directories**.

A *directory* is actually a file itself which contains information for the system to use to find other files. It can be imagined as a place, so that files are said to be contained *in* directories and you are said to work *inside* a directory.

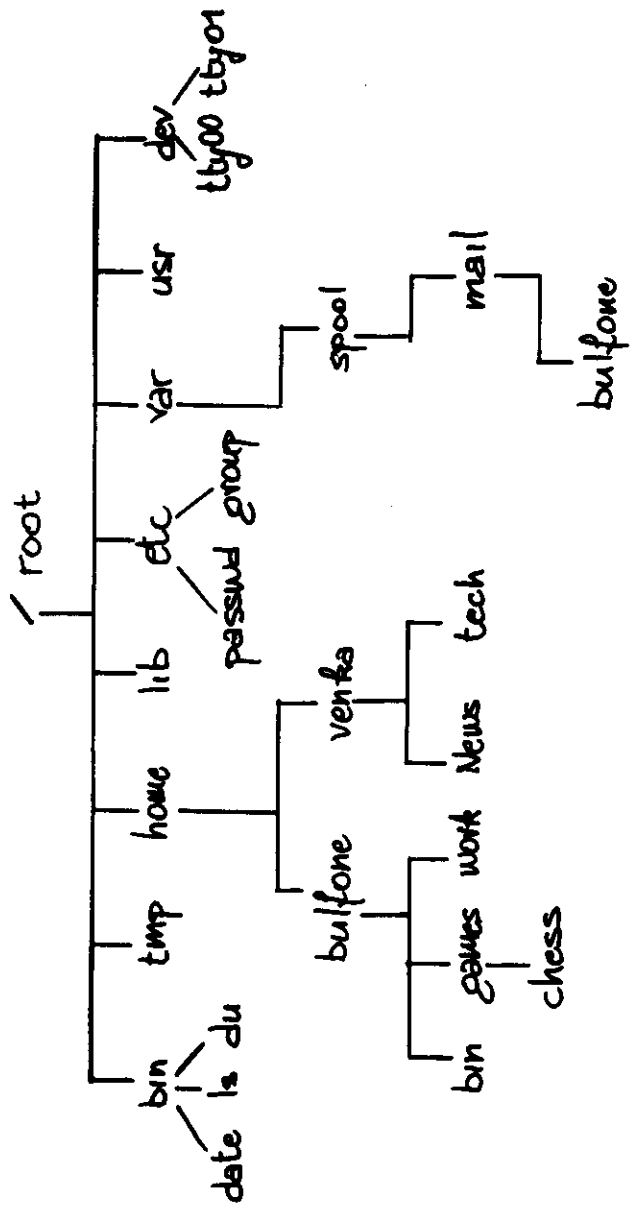


Fig. 2. UNIX Directory Tree

Directories are organized into a hierarchical structure that is usually imagined as a family tree. The parent directory of the tree is known as the *root* and is written as a slash (/).

The root contains several directories (see fig. 2).

bin, *home*, *etc*, *usr*, *tmp*, *dev* and *lib* are subdirectories of root. These are fairly standard directories and contain specific kinds of system files (e.g. *bin* contains many UNIX commands).

Each directory has one parent directory and may have one or more child directories.

The notation used to specify file and directory names is called a *pathname*.

An *absolute* pathname specifies the path of directories you must travel to get from the root to where you want to go.

For example, */home/bulfone* is an absolute pathname and defines a unique directory as follows:

- the root is designated by "/"
- the directory *home* (a subdirectory of *root*)
- the directory *bulfone* (a subdirectory of *home*).

A *relative* pathname points to a file relative to your current directory.

Unless you specify an absolute pathname, the shell assumes you are using a relative pathname.

You can go up the tree by using the shorthand ".." for the parent directory.
You can also go down the tree by specifying directory names. You just have to name each step along the way, separated by a slash (/).

Referring to the example of fig. 2, if the current directory is */home/bulfone*, the relative pathname to *work* is simply *work* (remember not to start with a "/", as this would specify an absolute address relative to the root).

In order to address the *games* subdirectory of *bulfone* when you are in *venka*, you may use the absolute pathname */home/bulfone/games*.
Alternatively, you can go up one level to *home*, then go down the tree to *bulfone* and finally to *games*. The relative path name is *../bulfone/games*.

Absolute and relative pathnames are totally interchangeable.

The directory you enter when you log in to UNIX is called *home directory*.

This directory contains the files you use almost every time you log in. It is a unique place in the UNIX file system from which you can create your own files and build your own directory tree.

In the definition of a pathname, you can always refer to your home directory by the symbol ~.

The directory you are currently in is the *working directory*.

At the start of every session, the home directory is also your working directory.

All commands that you enter apply to the files in your current working directory. When you create files, they are created in your working directory.

3rd Course on Basic VLSI Design Techniques ICTP, Trieste, 21 Nov/16 Dec 1994

3.2 Exploring the UNIX File System

- pwd

It prints out the pathname of your working directory. It takes no arguments.

- cd

It is used to change your working directory to a parent or child directory or to any other directory (if permissions allow it).
It has the form:

cd *pathname*

The argument *pathname* can be an absolute or a relative pathname to the directory you want to move to.

The single-word command **cd**, with no arguments, will always take you back to your home directory, wherever you are in the file system.

- ls

The **ls** command is used to list the files and the subdirectories contained in the working directory. The syntax is:

ls *option(s) filename(s)*

Note that the form **ls** *dirname* lists files in the directory called *dirname*.

For example:

```
% ls
emacs games ictp1 pippo work  disk_storage
```

There are a number of different options that **ls** can take to display different amounts of information related to files and subdirectories as well as to change the format of the display.
The *-a* option shows you some more files, as in the following example:

```
% ls -a
.   .login ictp1  emacs  work
..  .cshrc games  pippo  disk_storage
```

Two new files have appeared, with the names "." (dot) and ".." (dot dot). As already mentioned, .. is a special notation for the parent directory. A single . is a special notation for the current directory. Any file whose name begins with a dot is hidden and will only be listed if you type **ls -a** (*.login* and *.cshrc* contain commands that are automatically executed whenever you log in, see paragraph 7.5).

To get more information about each file, add the **-l** option. This option, in combination with **-a**, will give:

```
total 13
drwxr-xr-x  6  bulfone users 1024  Nov  18 17:55  .
drwxr-xr-x  6  bulfone users 1024  Nov  18 17:55  ..
-rwxr--r--  1  bulfone users 251   Nov  18 16:58  .cshrc
-rwxr--r--  1  bulfone users 272   Nov  18 17:31  .login
-rw-r--r--  1  bulfone users 131   Nov  18 15:08  disk_storage
-rw-r--r--  1  bulfone users 0     Nov  17 11:08  emacs
drwxr-xr-x  2  bulfone users 1024  Oct  14 12:34  games
-rw-r--r--  1  bulfone users 84    Nov  14 15:06  ictp1
-rw-r--r--  1  bulfone users 50    Nov  12 23:08  pippo
drwxr-xr-x  2  bulfone users 1024  Nov  18 14:35  work
|    |      |     |      |     |          |        |
| access   | owner    |     |  creation date |
| modes    |          group |     and time   |
type   # links          # bytes              filename
```

total *n*	*n* is the number of 1024-byte blocks of storage used by the files in the working directory
type	Tells whether the file is a directory *(d)* or a plain file *(-)*
permissions	(access modes) Specifies three types of users (yourself, your group, all others) who are allowed to read *(r)*, write *(w)*, or execute *(x)* files
links	Tells the number of files and directories linked to that file
owner	Specifies who created or owns the file
group	Specifies the group that owns the file
#bytes	Specifies the size of the file
creation date	Tells when the file was last modified
filename	Is the name of the file or directory

The person who creates a file is its owner; if you create any files, the owner column should show your login id.

You also belong to a group, to which you were assigned by the System Administrator. Any files you create will be marked with the name of your group.

Permissions control who can read, write (modify) or execute the file (if it is an executable program).

There are a total of ten characters in the permission listing. The first shows the file type (directory or plain file). The next three characters show the permissions for the file's owner (you, if you created the file). The next three characters show permissions for the other members of your group. The final three characters show permissions for all other users.

For example, the permissions for *.login* are **- rw- r-- r--**, meaning that anyone (you, your group and all the users) can read it, but only you can modify it.
In the case of directories, *x* means the permission to search the directory (for example, to list its contents with **ls**).

If you are interested only in knowing which files are directories and which are executable files, you can list the files using the **-F** option.

Directories are marked with / (slash) at the end of the filename. Files with an execute status (x), like programs and shell scripts, are marked with a * (star).

- chmod

The command **chmod** changes the mode (permissions, owner and group) of files or directories.

The mode can be changed by the owner or by the super-user.

chmod accepts specification of the required mode in one of two different ways:
a. as an absolute value given as an octal number
b. as what is termed a 'symbolic mode'.

The syntax is:

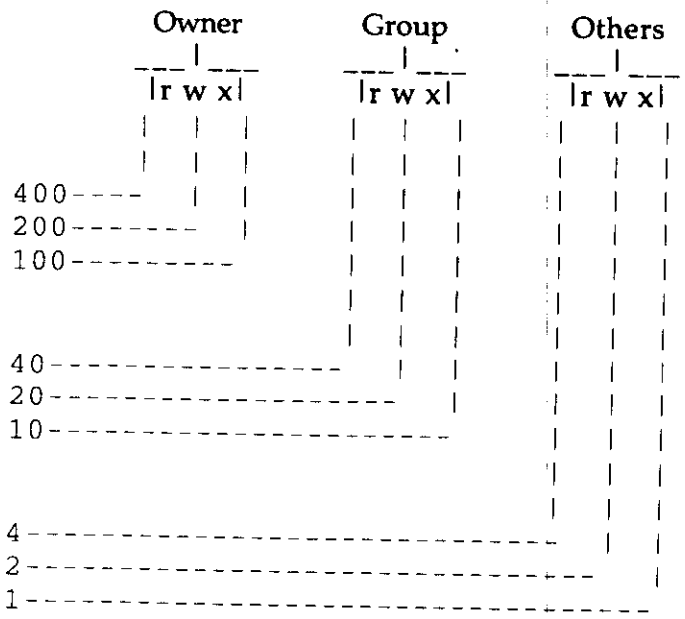
chmod *mode filename*

Numerical specification is not the easiest to use unless you are familiar with binary arithmetic. Each permission in the group of nine is represented by a one, each protection (lack of permission) is represented by a zero. So **rw-r--r--** translates to 110100100 or 644 in octal notation.

To translate the mode you require to a number, simply add up the numbers of table 1 corresponding to the individual permissions you want.
If you want the file *ictp1* to be readable and writeable by you, readable by the group and the other users, you should enter:

% chmod 644 ictp1

Table 1. Translation Diagram for Permissions



The symbolic mode is rather more complex.
You have the choice of saying which mode you want, or of saying how you want the existing permissions to be modified.

The following abbreviations are used:

u	user permissions
g	group permissions
o	other permissions
a	all permissions
=	assign a permission absolutely
+	add a permission
-	take away a permission

Permission types are *r*, *w* and *x*.

If you want to assign to *ictp1* read/write permissions for user, read for group and read for all others, you should enter

```
% chmod a=r,u+w ictp1
```

- cat

Beside being used to "con*cat*enate" files in order to create a bigger one, it can also be used to display a file to the terminal screen (standard output):

cat *filename*

cat works best for short files, which can be displayed in one screen.

If you cat a file that is too long, it may roll up the screen faster than you can read it. Press CTRL-S to freeze the output at a particular point, and CTRL-Q to resume listing the file.
You cannot go back to view the previous screens when you use **cat**.

If you accidentally type **cat** without a filename, press CTRL-D to get out of **cat**.

- more

If you want to read a "long" file on the screen, you can use the **more** command to display one screen of text at a time. The syntax is:

more *filename*

The **more** command displays the first part of the file which can be contained in the screen and the last line is a "percentage seen" message like:

(More) --- 47 %

This says that you are *47%* of your way through the file.
Press RETURN to view the next line.
Press SPACEBAR to view the next page.
Enter "**h**" to see what commands are available with **more**.

3.3 On-line Documentation

If you want some information about the correct syntax for entering a command or the particular features of a program, enter the command **man** and the name of the command. The format is:

man *command*

The output of **man** is implicitly "filtered" through the **more** command.

Using Linux, a nice way of getting on-line information for commands is to type:

command --help

3.4 Communicating with Other Users

We have seen that each file is marked with its owner's login id.

An even more visible sign that the system knows each of its users by name is that you can communicate with other users via electronic mail.

- mail

This facility allows you to compose a message at your terminal and send it to another user or list of users. It also enables you to read any messages that others may have sent to you.

To send mail to another user, type:

```
% mail username
Subject:
```

After that, you will not have any prompt. Instead mail is waiting for you to enter your message. Type in your message, press RETURN after every line, enter CTRL-D on a separate line to exit.

In order to read your mails, simply enter the command **mail** at the system prompt. After a mail status message, you are prompted by the mail program (&). Press RETURN to read the last mail. Type **help** and RETURN to see the mail program options.

```
% mail
Mail version 5.5 6/1/90. Type ? for help.
"/var/spool/mail/bulfone": 1 message 1new
>N 1 bulfone      date "Subjectname"
& help
```

4. Creating and Managing Files

4.1 Creating Files

- vi

Normally you create a text file with a text editor. An editor combines the power of a typewriter, an erasable pencil on paper, snips and paste.
The usual editor in the UNIX environment is **vi**. The syntax is:

vi *filename*

What happens is that a portion of the file you wish to modify is displayed in your terminal screen (a window on the file).

Within that window you can move the cursor around to control where changes are to be made, and then you can make changes by replacing text, adding text or deleting text.

vi has a wide range of commands for positioning the cursor, editing etc.: just about every key on the keyboard is some command to **vi**.

To learn it, you really need to get on a terminal and use it!

There are two basic modes of operation:

Command mode — Is the mode that allows you to move the cursor and enter commands (like delete, save, etc.). It is the first mode you enter after starting **vi**.

Insert mode. — Is the mode that allows you to enter new text into a file. To leave the insert mode, press ESC.

There are different commands to enter the insert mode: for example **i** will insert text before the cursor, **a** will append text after the cursor, etc.

Use the key arrows to move the cursor, or, if you don't have them on your keyboard, use **h** (left), **j** (down), **k** (up) and **l** (right).

You can, for example, delete *n* characters beginning with current using the command *n***x**, etc.

Getting out of **vi** is very important, especially at the beginning when it is easy to make mistakes.

Use the command **:wq** and press RETURN or **:x** and press RETURN to save and exit. Use the command **:q!** and press RETURN to exit without saving any changes.

See Appendix A for a detailed description of the **vi** commands.

4.2 Wildcards in File Names

A *filename* is the most important property of a file.

Filenames may contain any character except /, which is reserved as the separator between files and directories in a pathname. Filenames can be made up of upper and lower case letters, numbers, and the special characters "." (dot) and "_" (underscore).
When you have a number of files named in series (for example *ictp1* to *ictp12*) or filenames with common characters (like *set*, *senate* and *serial*), you can use *wildcards* to specify many files at once. These special characters are * and ?.

When used in a filename given as an argument to a command:

* is replaced by any number of characters in a filename. For example, *se** would match *set*, *senate* and *serial* if they were in the same working directory. You can use this also to save typing for a single filename.

? is replaced by any single character (so *h?p* matches *hop* and *hip*, but not *help*).

In addition, you can use brackets ([]) to surround a choice of characters you would like to match. Any one of the characters between the brackets will be matched. For example, *[Cc]hapter* would match either *Chapter* or *chapter* (but *[ch]apter* would match either *capter* or *hapter*).
Use a hyphen to separate a range of consecutive characters. For example, *chap[1-3]* would match either *chap1*, *chap2*, or *chap3*.

4.3 Managing Files

- cp

The **cp** command is used to copy a file in the same directory or to copy a file to and from directories. To make a copy of the file, use the command:

cp *old new*

where *old* is a pathname to the original file and *new* is the name you want to give to the copy of the file.
For example, to copy the */etc/passwd* file into a file called *password* in your working directory, you have to enter:

% **cp /etc/passwd password**

You can also use the form:

cp *old old_dir*

This puts a copy of the original file *old* into an existing directory *old_dir*. It gives the name of the original file to the copy of the file.

You can copy more than one file at a time to a single directory by listing the name of each file you want copied, with the destination directory at the end of the command line.

- mv

The **mv** command is used to rename a file. It is also used to actually move a file from one directory to another.
The **mv** command has exactly the same syntax as the **cp** command:

mv *old new*

where *old* is the old name of the file and *new* is the new name that you want to give to it.

It is always a good practice to make sure that the new name is unique; an existing file may be overwritten by the **mv** operation.

- mkdir

In order to create a new directory, use the **mkdir** command. The format is:

mkdir *dirname*

where *dirname* is the name of the new directory.

- rm

The **rm** command removes unwanted files so you can clean up your directory tree. The command:

rm *filename*

removes the file called *filename*.

Make sure you are deleting the correct files when you use wildcards in the **rm** command.
If you accidentally remove a file you need, you cannot recover it unless you have a backup copy in another directory, or on tape.

If you do not have a backup copy of a file you removed with the **rm** command, you will not be able to recover the file at all.

Do not type "**rm ***" heedlessly. If you do, you will delete all the files in your current working directory!

It is good practice to list the files with **ls *** before you remove them.

Or, you can use the *-i* option of **rm** which will prompt you interactively to confirm for each filename that you want it removed.

- rmdir

Just as you can create new directories, you can also remove them with the **rmdir** command.
As a precaution, the **rmdir** command only lets you delete directories that do not contain any files. The syntax is:

rmdir *dirname*

If a directory you try to remove does contain files, you will get a message like "**rmdir:***dirname* **not empty**".

The procedure for deleting a directory that contains some files is:

a. Enter **cd** *filename* to get into the directory you want to delete.

b. Enter **rm*** to remove all files in that directory.

c. Enter **cd ..** to go to the parent directory.

d. Enter **rmdir** *filename* to remove the unwanted directory.

d'. In case you still get the message "*dirname* **not empty**", use **ls -a** to check that there are no hidden files other than . and .. To remove all hidden files, type **rm .***.

e. Enter **ls -l** to check if the directory was actually removed.

5. Input/Output Redirection

5.1 Standard Input and Standard Output

A UNIX command usually requires some input (such as a file) and results in output.

In general, if no filename is specified in a command, the shell takes whatever you type in your keyboard as input to the command. Your terminal keyboard is the *standard input*.

When a command has finished running, the results are usually displayed on your terminal screen. The terminal screen is the *standard output*.

By default, each command takes its input from the standard input and sends the results to the standard output.

These two default cases of input/output can be varied. You can often use a given file as input to a command using the "<" operator. You can also write the results of a command to a named file or some other device instead of displaying it on the screen using the ">" operator. This is called *input/output redirection*. In general, redirecting input is not used as often as redirecting output, because most commands are designed to take their input from files anyway.

Input/output redirection is one of the nicest features of UNIX because of its tremendous power and flexibility.

5.2 Putting Text in a File

Directing the output of a command into a file is useful when you have a lot of output that would be difficult to read on the screen or when you put files together to create a bigger file.

When you append the notation "> *filename*" to the end of a command, the results of the command are diverted from the standard output to the named file. The > symbol is known as the *output redirection operator*.

Using **cat** with this operator, the contents of the file that are normally displayed on the standard output are diverted into a new file. For example:

```
% cat /etc/passwd > password
```

The command "**cat /etc/passwd**" simply displayed the file */etc/passwd* on the screen. In the example above, we use the > operator. Instead of printing the results of the command on the terminal screen, the contents are diverted to a file called *password* in the current directory. The effect is the same as the copy command "**cp /etc/passwd password**".

The > redirection operator can be used with any command, not just with **cat**. For example:

```
who > users
date > today
```


You can also create a small text file using the **cat** command and the > operator.

We said to press CTRL-D in case you type **cat** without a filename. This is because the **cat** command alone takes whatever you type on the keyboard as input. Thus, the command:

cat > *filename*

takes input from the keyboard and redirects it to a file. For example:

```
% cat > song
You can call on my name, and you know where ever I
am, I'll come running to see you again.
J. Taylor
^D
```

cat takes the text that you typed as input, and the > operator redirects it to a file called *song*. Type CTRL-D on a new line by itself to signal the end of the text. You get back the system prompt.

You can create a bigger file out of many smaller files using the **cat** command and the ">" operator. The form:

cat *file1 file2* > *newfile*

creates a file *newfile*, consisting of *file2* appended to *file1*.

You can also append information to the end of an existing file, instead of replacing its contents, by using the >> (*append redirection*) operator. The syntax is the following:

cat *file2* >> *file1*

appends the contents of *file2* to the end of *file1*.

If you are using the > operator, you should be careful not to accidentally overwrite the contents of a file. The system normally lets you redirect output to an existing file!

5.3 Pipes and Filters

In addition to redirecting input/output to a named file, you can connect two commands together so that **the output from one program becomes the input of the next command.**
Two or more commands connected in this way form a *pipe*.

A pipe is designated by a vertical bar (|) on the command line between two commands.

When a pipe is set up between two commands, the standard output of the command to the left of the pipe symbol becomes the standard input of the command to the right of the pipe symbol.

Any two programs can form a pipe as long as the first program writes to standard output and the second program reads from standard input.

When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output (or even "pipes" the result to another program), it is referred to as a *filter*.

One of the most common uses of filters is to modify output. The UNIX filters can be used to restructure output so that only the desired lines are displayed on the screen or sent to another file.

Almost all UNIX commands can be used to form pipes. Commands that are commonly used as filters are described below. Note, anyway, that they useful also as individual commands.

- grep

It searches a file or files for lines which contain strings of a certain pattern. The syntax is:

grep *"pattern" file*

The simplest use of **grep** is to look for a pattern consisting of a fixed character string. For example, it can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output:

% **ls -l | grep "Aug"**

The pipe above looks for all files in your directory that contain the string "Aug" (that is, they were last modified in August) and sends those lines to the terminal screen.

Some of the most useful **grep** options are given below:

-v Print all lines that do not match pattern
-n Print the matched line and its line number
-l Print only the names of files with matching lines
-c Print only the count of matching lines
-i Ignore upper and lower case

- sort

The **sort** command arranges the contents of a file alphabetically or numerically. The syntax is:

sort *filename*

Some of the most useful **sort** options are given below:

-n Sort by arithmetic value, ignoring blanks and tabs
-r Reverse the order of sort
-f Sort regardless of upper or lower case
+*x* Skip *x* fields before start sorting

More than two commands may be linked up into a pipe. Taking the previous pipe example using **grep**, we can further sort the files modified in August by order of size. The following pipe consists of the commands **ls**, **grep** and **sort**:

```
% ls -l | grep "Aug" | sort +4n
```

This pipe sorts all files in your directory modified in August by order of size, and prints them to the terminal screen. The **sort** option *+4n* tells the system to skip four fields (a field is preceded by a blank) then sort the lines in numeric order.

- wc

The wc command counts the number of lines, words, and characters in a file. The syntax is:

wc *filename*

Options resctrict the counting process to the indicated object:

-l Counts only line
-w Counts only words
-c Counts only characters

6. Managing UNIX Multi-Tasking

On a single-tasking system like MS-DOS, you would enter the command and then wait for the system prompt telling you that you could enter a new command.

In UNIX, there is a way to enter new commands in the "foreground" while one or more commands are still running in the "background". This is called *background processing*.

Background processing is useful, for example, for running slow programs whose output you do not need immediately.

When you enter a command as a background process, the shell assigns a process ID to the background process and prints out the process ID. The system prompt reappears immediately so that you can enter a new command.

The command is still running in the "background" but you can use the system to do other things during that time. You can also log off at this time since the background process runs to completion even when you are not using the system.

To run a command as a background process, add the "&" (ampersand) character at the end of the command line before you press RETURN. For example:

```
% who > users &
[1] 18702
```

The process ID (PID) for the command is 18702.
The process ID is useful when you want to check the status of a background process or to cancel it when you have to.

Using the shell, you can put an entire sequence of commands separated by semicolons into the background by putting an & at the end of the entire command line:

command1; command2 &

The shell supports an additional feature called job control. You can use the *suspend character* CTRL-Z to suspend a program running in the foreground. The program will pause, and the shell will give you a new system prompt. You can then do anything else you like, including putting the suspended program into the background using the **bg** command.

The **jobs** command lists all background processes. For example:

```
% jobs
JOB NUMBER      STAT        COMMAND
[1]             Running     sleep 60
```

The **fg** *jobnumber* command will bring a background process to the foreground.

- ps

ps allows you to see how long a process has been running. The output of **ps** also tells you the process ID of the background processes and the terminal from which it was launched. For example:

```
% ps
 PID   TTY  STAT TIME   COMMAND
8070   020    S  0:12   csh
8231   020    R  0:02   ps
  |     |          |
  | terminal       |
  |   line         |
  |                |
process            run
  id               time
```

process ID — A unique number assigned by UNIX to the background process.

terminal line — The terminal number from which the background process was launched.

run time — The amount of computer time that the process has used.

command — The name of the process.

- kill

The **kill** command is used to stop a background process from being executed further. The syntax is:

kill *PID(s)*

kill terminates the designated process IDs (shown under the PID heading in the **ps** listing).
If you do not know the process ID, do a **ps** first to display the status of background processes.

Try the following example, where the **sleep** command simply causes a process to "go to sleep" for *n* number of seconds. We enter two commands, **sleep** and **who**, on the same line as a background process:

```
% (sleep 60; who) &
% ps
% kill sleep PID
```

Suppose we decide that 60 seconds is too long a time to wait for the output of **who**. The **ps** listing shows the process ID associated to the **sleep** command, so we use this PID to kill the **sleep** process. You will see a "termination message" from the system. The **who** command is executed immediately, since it is no longer waiting on **sleep**, and proceeds to list the users logged into the system.
Some processes may be hard to kill. If a normal **kill** of these processes is not working, enter "**kill -9** *PID*". This is a sure kill and can destroy anything, including the shell which is interpreting it!

7. Customizing the C Shell

7.1 Aliases

An alias is an abbrevation for a frequently used command or series of commands. The syntax is:

alias *name definition*

You define an alias by giving its name followed by a definition. If the definition contains special symbols, put single quotation marks around it. For example:

```
% alias list ls -l
```

makes **list** display a long listing of your directory.

```
% alias blocks 'pwd; du -s'
/home/bulfone
18    .
```

uses **pwd** to display the directory name and **du** to report how many blocks it contains (note that the ; is a special character used to separate commands).

The **alias** command with no arguments displays all defined aliases.

7.2 Shell variables

The shell gives you the capability to define *variables* that can hold strings or set of strings. The main reason for using these variables is to save long strings such as pathnames. The syntax is:

set *varname* = *string*

This creates a shell variable called *varname* and initializes it to a string of characters. If the string includes spaces, you must put quotation marks around it to include them. For example:

```
% set cf = /home/bulfone/games/chess
```

We assign the name *cf* (current file) to the filename */home/bulfone/games/chess*.

```
% set title = "ICTP VLSI Course"
```

Here we use quotation marks to preserve the spaces in the variable *title*.

You can use shell variables in commands by preceding their names with **$**.
Therefore, if you wanr to display variables, use the command

echo *$varname*

To get a long listing of the file */home/bulfone/games/chess*, all you have to type is:

```
% ls -l $cf
```

You can release a variable when you no longer need it with the **unset** command:

unset *varname*

For example, to release the shell variable **cf**, type

```
% unset cf
```

The **set** command with no arguments displays the shell variables and their current values.

Many UNIX commands display information only. We can replace a command with its output by using *command substitution*: substituting the output of the command for the command itself. For example:

```
% set d = `date`
% echo $d
Fri Nov 18 13:01:41 MET 1994
```

This sets a shell variable **d** to the string output from the **date** command.

We call **d** a *wordlist* (it is a variable that contains a list of strings divided into separate words; surround the list with parentheses to define a wordlist). Now you can reference individual words as *$d[1]*, *$d[2]*, and so on. The command

```
% echo Today is $d[1]. The time is $d[4]
Today is Fri. The time is 13:01:41
```

demonstrates how this technique lets you control the format of the **date** output.

Some of these variables belong to a special list, and are called the shell's predefined variables. They can have default settings (being initially the same for every user), while others get their values from UNIX user information files (such as */etc/passwd*). The most widely used are described below.

path

It contains a list of directory names. The shell uses it to search for commands. For example:

% set path = (~/bin /bin /usr/bin /usr/local .)

The shell searches the directories in the *path* variable to find and execute the commands you enter (note that it is a wordlist variable). The shell searches *path* from left to right, so you should order its entries carefully.

home

It contains the absolute pathname of your home directory. The shell uses it when you type **cd** without arguments and when you use ~ in a command.

shell

By default, the shell initializes *shell* to */bin/csh*. The shell executes this file when you run a script or issue a command that creates a subshell. Normally you don't change *shell*.

mail

It is a wordlist variable which defines your mailbox file. For example:

% set mail = (/var/spool/mail/bulfone)

The shell checks this file regularly to inform you when you have mail.

prompt

It is the shell prompt.

noclobber

The *noclobber* variable keeps you from overwriting existing files with output redirection. The default value is unset, use the statement

% set noclobber

to put this protection in effect.

7.3 Environment Variables

Special information like your login directory, the location of your mailbox, your terminal type are all part of your user environment.

A set of special variables, called *environment variables*, maintains this data. UNIX can pass ("export") these to programs executed from the shell.

Unlike shell variables that are accessible only within the shell, environment variables are available to both your current shell and subsequent programs (such as scripts, editors, Alliance). When you execute a program, UNIX gives it the values of all environment variables.

Although environment variables are not built into the shell, you can define them from it. The syntax is:

setenv *VARNAME string*

It defines *VARNAME* to be an environment variable and initializes it to *string*.

You can release an environment variable when you no longer need it with the **unsetenv** command:

unset *VARNAME*

The **env** command with no arguments displays the environment variables and their current values.

The shell maintains certain environment variables, and you may also create your own. The list of the most widely used *predefined environment variables* is described below.

TERM

The variable *TERM* is set to a code that defines your terminal type. The **vi** editor and the **more** command use your terminal's cursor-positioning commands: the *TERM* variable makes these terminal-independent. For example the command

% setenv TERM vt100

sets *TERM* to *vt100*.

HOME

The *HOME* variable contains the absolute pathname of your home directory. It is initialized at login time from the home directory field in the UNIX password file */etc/passwd*. Your working directory is initially *$HOME* when you login.

The predefined shell variable *home* takes its value from the *HOME* environment variable.

PATH

It contains the same information as the shell *path* variable, but in a different format, By default, the login program sets *PATH* to

```
:/bin:/usr/bin
```

The shell maintains both *path* and *PATH*. When you change *path*, the shell updates *PATH* automatically. This permits the information stored in *path* to be exported to other programs.

PATH's notation is different from *path*, and it is not a wordlist variable. By changing *path* with:

% set path = (~/bin /bin /usr/bin /usr/local .)

gives the following value for *PATH* (displayed by **echo $PATH**):

```
/home/bulfone/bin:/bin:/usr/bin:/usr/local:.
```

USER

Both hold the user's login name. The login program initializes *USER* from the */etc/passwd* file. It is useful to personalize commands and shell scripts.

SHELL

It contains the absolute pathname of your shell. By default this variable is undefined, therefore you should initialize *SHELL* to the shell variable *shell*, that is */bin/csh*.

MAIL

The *MAIL* variable contains the pathname of your mail file. This is the value used by the **mail** command; the shell uses the *mail* variable to see if you have any new mail.

7.4 Shell Scripts

Shell scripts are a powerful tool for invoking and organizing UNIX commands. They can even substitute for system programs written in a general-purpose language such as "C".

At the simplest level, they can invoke one command after another. This is comparable to the batch files provided in many operating systems (e.g. MS-DOS).

However, the shell's more advanced programming options can handle an even wider range of problems. Decision sequences and loops allow a shell script to perform almost any programming task. Typical applications include file management, format conversion, searches, etc.

One key factor is that shell scripts are easy to write. All you need is an editor. You don't need to compile a script, link it, or load it: just make it executable. Debugging is straightforward, since the shell has options that display intermediate output as a script executes.

Let's look at the following example. Suppose you want to keep a record of your disk usage. You plan to use the command line

% (date; du ~) >> ~/disk_storage&

This results in the background execution of **date** followed by **du** (disk usage) apllied to your home directory. We add the output to the file *~/disk_storage*. Running this command updates *~/disk_storage*.

The problem is that the latest information goes at the end of the file instead of at the beginning. Let's make a script called **ppd** to append at the beginning. It will have a filename as its argument and will read from the standard input. This allows us to use it in a pipe as follows:

% (date; du ~) | ppd ~/disk_storage&

```
1   #csh script to prepend std input to file argument
2   #Version 1.0
3
4   set tf = /tmp/ppd.$$    #name temp file
5   set dest = $argv[1]     #get argument name
6
7   cat - $dest > $tf       #cp std input,$dest to $tf
8   mv $tf $dest
```


The first character in a script must be the symbol #. # is also the comment symbol; the shell ignores everyting after it on a line.

We use a shell variable to hold the pathname of the temporary file.

The idea here is also to make **ppd** a tool anyone can use at any time. Line 4 defines a temporary filename using the current process number *$$*. Since this number is unique systemwide, two users calling **ppd** at the same time will not try to use the same temporary file.

Line 4 sets variable *tf* to the temporary filename.

Line 5 sets *dest* to *$argv[1]*, a shell variable that lets scritpts refer to command line arguments. *$srgv[1]* is the first argument, *$argv[2]* the second, and so forth.

Line 7 uses **cat** to combine the standard input and the argument file into a new file called *$tf*. The - option reads the standard input first. Line 8 renames *$tf* to the original filename *~/disk_storage*. *~/disk_storage* now has the latest information at the beginning.

To make **ppd** executable, use the **chmod** command:

chmod +x ppd

Use the **echo** command to display messages from a script. Consider the example:

```
1    #csh script that gives formatted display of
2    #current date and time
3    #Version 1.0
4
5    set d = `date`
6    echo "Today's date: $d[2-3] $d[6]"
7    echo "Current time: $d[4]"
```

Line 5 uses command substitution to assign the current date to variable *d*.
Lines 6 and 7 print the date and time by accessing different words in *d*.

In the following example, we use the shell *if* statement. The **greet** script displays a greetings message on Friday.

```
1    #csh script to get weekend greetings
2
3    set d = `date`
4    echo "Today is $d[1]"
5    if ($d[1] == Fri) then
6    echo ====================
7    echo Have a nice weekend!
8    echo ====================
9    endif
```

Lines 5 to 9 contain the *if* statement.

Line 5 uses parentheses to create an expression ($d[1] == Fri). The == is a relational operator meaning "is equal to". If the expresion is true, the shell executes statements 6, 7 and 8.

Line 9 marks the end of the *if* statement.

The shell has many other programming features, that make it extremely powerful. Among the others:

Conditional Statements

```
if (expression) then
    command(s)

else
    command(s)

endif
```

Foreach Construct

```
foreach name ( wordlist )
    command(s)
end
```

While Construct

```
while (expression)
    command(s)
end
```


Switch statement (replaces a long sequence of if-then-else statements)

```
switch (string)
    case pattern1:
        command(s)
        breaksw

    case pattern2:
        command(s)
        breaksw

    .    .    .    .

    default:
        command(s)
        breaksw

endsw
```

7.5 The C shell Special Files

The shell lets you customize your UNIX interface by means of three special files: **.cshrc, .login, .logout.**

The .cshrc File

After you login, UNIX calls the shell to execute your commands. But before displaying the prompt, the shell must di some work. It first searches for a file **.cshrc** in your home directory. If it is there, the shell executes the commands stored in it.

The **.cshrc** file is used to define shell characteristics. These includes primarily aliases and shell variables.

Since the shell executes **.cshrc** every time it executes a script (or a new shell), this makes aliases and default settings available to scripts (tus the shell may execute our **.cshrc** file many times during a session.

Let's consider the following example:

```
1   #.cshrc file - Version 1.0
2
3   # Set up shell variables
4   set noclobber          #redirection protection
5   set path = (~/bin /bin /usr/bin /usr/local .)
6
7   # Set up aliases
8   alias bye logout       #bye for logout
9   alias nusers \
10  '(set a = `who|wc -l` ;echo "There are $a users")'
```

Line 4 sets *noclobber* shell variable for extra protection.

Line 8 defines an alias for **logout** and line 9\10 define the alias **nusers** to report the number of users currently logged in.

Line 5 makes the search path include */usr/local and ~/bin.*

The .login File

Next the shell executes file **.login** in your home directory if it is available.

.login is executed only once each session.

Environment variable definitions are placed in **.login**.
UNIX makes these variables available to other programs, including a shell that executes a script.

Place commands that you want executed once at login in **.login**.
Using **.cshrc** the shell would execute these commands each time you execute a script file.

Place commands that set up terminal characteristics in **.login**.

Let's consider the following example:

```
1   # .login - Version 1.0
2
3   # Greetings and Salutations
4   echo Hello, $USER !
5   echo Welcome to UNIX
6   echo -n "Today is "; date
7
8   #Environment Variables
9   setenv SHELL /bin/csh  #C shell
10  setenv TERM vt100
11
12  stty intr ^C          #set inerrupt to cntrl-C
13  nusers
```

Lines 4 to 6 display the login messages using the UNIX commands **echo, date** and the environment variable *$USER*.

Line 9 defines our *SHELL* as */bin/csh* (C shell) and line 10 defines the *TERM* variable.

The **stty** command in line 12 sets the interrupt key to control-C.

Line 13 executes the alias **nusers** to display the numebr of users logged in. We can use aliases in our **.login** file because **.cshrc** has already been executed.

The .logout File

When you want to end your session, you execute **logout**. The shell executes **.logout** in your home directory if available.

Let's consider the following example:

```
1    # .logout - Version 1
2
3    set d = `date`
4    if ($d[1] == Fri) then
5         echo Thanks God It is Friday
6    endif
7
8    echo -n "Logging out at: "
9    date
```

Word $d[1] of wordlist d is compared with the string Fri in order to **echo** a TGIF message.

Lines 8 and 9 display the exit message.

7.6 UNIX Database Files

UNIX maintains several database files to initialize shell predefined variables and UNIX environment variables. Let's have a quick look to some of these files.

/etc/passwd

It contains a single line of information for each user. Each line is divided into fields separated by colons according to the following format:

user:password:userid:groupid:comment:home:program

User is your login name.

Password contains your login password (encrypted).

Userid is a unique number that UNIX assigns to your username, and it is used by the system for storing file and directory ownership.

Likewise, groupid is an other unique number assigned to your default group. The corresponding name is stored in /etc/group.

The comment filed contains personal or administrative information. home contains your home directory.

Program is the default shell (or any program) that the login precess executes when you login.

/etc/cshrc

This is a systemwide startup file for all users who run the shell.

The shell executes commands placed in this file before executing the user's **.cshrc** file.

This eliminates the need to include these statemenmts in each user's **.cshrc** or **.login** file.

8. The Internet Network

During the last years, computer networks have grown at an impressive rate, driven by

- technical feasibility
- the need to share resources (information, news distribution, services)

The most popular and widely used computer network is the **Internet**.

What is the Internet?

It is a **global network made of hundreds of subnetworks** connected together, where more than **two million computers** of different types (from big mainframes, to microprocessors) can communicate each other.

Beginning of '94: 100 countries and more than **20 million users**!

How can so many computers, differing in their hardware and software, communicate each other?

They need **protocols**, a set of common rules to follow, a sort of common "language" to understand each other.

The **TCP/IP** (Transmission Control Protocol/Internet Protocol) protocols are used in the Internet, and mainly specify:

- how data packets can be routed through the different subnetworks
- how computers, from any subnetwork in the global network, can reliably exchange data packets

Of course, this is not enough, as we are human beings and we can interact each other, for example, by sending some written message through the mail and not by electrical signals like data packets.

Therefore, on top of the TCP/IP common set of protocols, a series of application services has been developed:

elm (electronic mail)

ftp (file transfer protocol)

telnet (remote login)

- elm

Allows to send e-mail through the Internet.

- ftp

Allows to transfer files through the Internet. The synatx is:

ftp *hostname* or *IPaddress*
`username`
`password`

Once you enter your ftp session, basic commands are:
ls or **dir**
to obtain the list of files in the current directory

cd *pathname*
to change directory (like UNIX)

lcd *pathname*
to change your local directory (the directory in your system, where you want to store the files to be downloaded)

bin
to set binary transfer mode

ascii
to set text tranfer mode

get *remfile locfile*
to download a file in your computer

put *locfile remfile*
to transfer a file in the remote site

help
on-line help

quit
to exit the ftp session

On the Internet there are different public archive sites that you can access by using `anonymous` as username (give you Internet e-mail address as password) where you can find public domain software, information, documentation, etc. For example Linux can be taken from `sunsite.unc.edu`

- telnet

Allows to open a remote session on an other computer. The synatx is:

telnet *hostname* or *IPaddress*
`username`
`password`

Type **logout** to exit the telnet session.

Historical Notes and Remarks:

Note that the Internet and its protocols are just an evolution of the so called **ARPANET**, a research project from DARPA (Defence Advanced Research Projects Agency of the U.S. Department of Defence).

The **Berkeley UNIX** adopted TCP/IP from the beginning and greatly contributed to its *de facto* standardisation. Moreover, the Berkeley UNIX specified a series of primitives, called *socket system calls*, which are used to access the TCP protocol in an easy way.

Shell Escape Commands

`:!` *cmd*	Execute shell command *cmd*; you can add these special characters to indicate: `%` name of the current file `#` name of last file edited
`:!!`	Execute last shell command
`:r!` *cmd*	Read and insert output from *cmd*
`:f` *file*	Rename current file to *file*
`:w !`*cmd*	Send currently edited file to *cmd* as standard input and execute *cmd*
`:cd` *dir*	Change the current working directory to *dir* (`$HOME` is default)
`:sh`	Start a sub-shell (CTRL-d returns to editor)
`:so` *file*	Read and execute commands in *file* (*file* is a shell script)

Shell Filters

`!`*cursor_cmd cmd*	Send text from current position to *cursor_cmd* to shell command *cmd*. Replace original text in file with output from *cmd*
`!}sort` Return	Example: Sort from current position to end of paragraph and replace text with sorted text

Macros and Abbreviations

`:map` *key cmd_seq*	Define *key* to run *cmd_seq* when pressed
`:map`	Display all created macros on status line
`:unmap` *key*	Remove macro definition for *key*
`:ab` *str string*	When *str* is inserted, replace with *string*
`:ab`	Display all abbreviations
`:una` *str*	Unabbreviate *str*

`Map` allows you to define strings of `vi` commands. Place in `.exrc` to run each time you enter `vi`. For long macros, set the `notimeout` option. If you embed control characters (e.g., keys like ESC) in the macro, you need to precede them with CTRL-v. If you need to include quotes (`"`), precede them with \ (backslash). Unused keys in `vi` are: `K V g q v * =` and the function keys.

Example:

`:map v /I` CTRL-v ESC `dwiYou` CTRL-v ESC ESC

When `v` is pressed, search for "`I`" (`/I` ESC), delete word (`dw`) and insert "`You`" (`iYou` ESC). CTRL-v allows ESC to be inserted.

Setting Options

Options shown here are default. To change them, either set them (`:set` *option*) or unset them (`:set no`*option*). To run options each time you enter `vi`, place in `.exrc` file in home directory and omit preceding colons (`:`).

`:set all`	Print all options
`:set no`*option*	Turn off *option*
`:set noai`	Set automatic indentation
`:set ap`	Print line after `d c J m :s t u` command
`:set bf`	Discard control characters from input
`:set eb`	Precede error messages with bell
`:set noic`	Ignore case when searching
`:set dir=tmp`	Set directory of buffer file
`:set lisp`	Modify brackets for Lisp compatibility
`:set magic`	Pattern match with special characters
`:set mesg`	Allow other users to send messages
`:set nolist`	Show tabs (`^I`) and end of line (`$`)
`:set nonu`	Prefix lines with line number
`:set opt`	Speed output: eliminate automatic Return
`:set prompt`	Prompt for command mode input with :
`:set nore`	Simulate smart terminal on dumb terminal
`:set remap`	Allow macros within macros
`:set noreport`	Indicate largest size of changes reported on status line
`:set ro`	Change file to read only
`:set scroll=`*n*	Set *n* lines for CTRL-d and `z`
`:set sh=`*shell_path*	Set shell escape (default is `/bin/sh`)
`:set showmode`	Indicate input or replace mode
`:set sw=8`	Set the shift width
`:set term`	Print terminal type
`:set noterse`	Shorten error messages with `terse`
`:set notimeout`	Eliminate one second time limit for macros
`:set tl=0`	Set significance of tags beyond this many characters (0 means all)
`:set ts=8`	Set tab stops for text input
`:set nowa`	Inhibit normal checks before write commands
`:set warn`	Warn "`No write since last change`"
`:set window=`*n*	Set number of lines in a text window

APPENDIX A

Quick Reference for vi

This quick reference lists commands you can use in the `vi` editor on Hewlett-Packard's UNIX™ System, HP-UX.

Notations

- Commands beginning with : (colon) must end with Return. These commands represent escape commands to the `ex` editor. You can reference the `ex` tutorial for more details.
- *file* is the name of file
- *cursor_cmd* is a cursor movement command (e.g., `G j w b`)
- *char* is a single character
- *str* is a character string (can contain pattern matching characters)
- CTRL-x means you press CTRL, hold it down, and press the x key.
- *n,m* can be two line numbers (e.g., `4,50`), line marker (e.g., `.,$`), or search expression (e.g., `/string1/,/string2/`).
- *(a-z)* means you choose a letter from a through z

Modes

Command Mode	When you are not inserting or changing text, you can move the cursor and run commands (e.g., searching, deleting, saving). Pay attention to the case of the commands; check the Caps lock key if `vi` behaves strangely.
Insert Mode	When you insert or change more than one character of text, you cannot use command mode commands. To leave the insert mode, press ESC.

Start a vi Session

`vi` *file*	Edit *file*
`vi -r` *file*	Edit last saved version of *file* after system or editor crash
`vi + n` *file*	Edit *file* and place cursor at line *n*
`vi +` *file*	Edit *file* and place cursor on last line
`vi` *file1 file2 ...filen*	Edit *file1* through *filen*; After saving changes in *file1*, enter `:n` for next file
`vi +/`*str file*	Edit *file* and place cursor at line containing *str*

Save Text and Exit vi

`ZZ` or `:wq` or `:x`	Save file and exit `vi`
`:w` *file*	Save *file* but do not exit; omitting *file* saves current file
`:w!` *file*	Save *file* overriding normal checking
`:`*n,mw file*	Write lines *n* through *m* to *file*
`:`*n,mw>>file*	Append lines *n* through *m* to end of *file*
`:q`	Leave `vi`, saving changes before last write (you may be prompted to save first)
`:q!`	Leave `vi` without saving any changes since last write
`Q`	Escape `vi` into `ex` editor with same file; `:vi` returns.
`:e!`	Re-edit current file, disregarding changes since last write

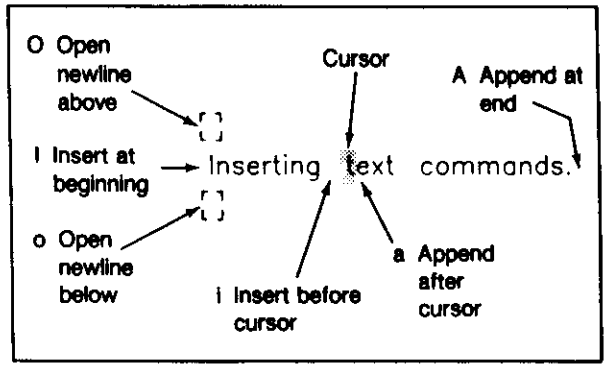
Status Commands

`:.=`	Print current line number
`:=`	Print number of lines in file
CTRL g	Show file name, current line number, total lines in file, and percent of file location
`:l` (letter "l")	Display tab (`^I`) backslash (\) backspace (`^H`) newline ($) bell (`^G`) formfeed (`^L`) of current line in status line

Inserting Text

To leave the insert mode, press ESC.

`a`	Append after cursor
`A`	Append after end of current line
`i`	Insert before cursor
`I`	Insert before beginning of current line
`o`	Open new line below current line and insert
`O`	Open new line above current line and insert
CTRL v *char*	While inserting, ignore special meaning of *char* (e.g., for inserting characters like ESC and control characters)
`:r` *file*	Read *file*, and insert after current line
`:`*nr file*	Read *file*, and insert after line *n*

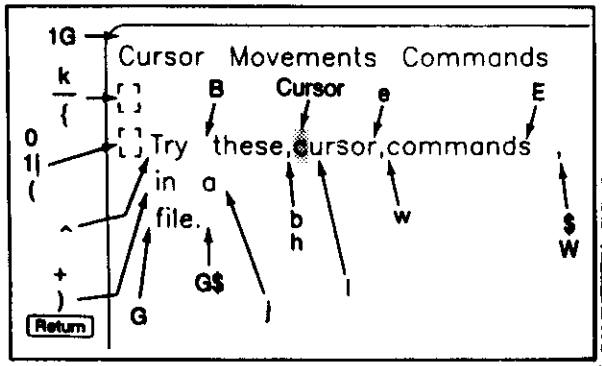


Undoing and Repeating Commands

`u`	Undo last command
`U`	Restore current line to original state
`"`*n*`p`	Retrieve last *n*th delete (last 9 deletes are in a buffer)
`"1pu.u.`	Scroll through the delete buffer until you retrieve desired delete (repeat `u.`)
`n`	Repeat last / or ? search command
`N`	Repeat, in reverse direction, last / or ? search command
; (semi-colon)	Repeat last `f F t` or `T` search command
, (comma)	Repeat, in reverse direction, last `f F t` or `T` search command
. (period)	Repeat last text change command

Moving the Cursor

`k` or CTRL p	Up
`j` or CTRL j or CTRL n	Down
`h` or CTRL h or Back space	Left
`l` or **Space**	Right
`w` or `W`	Start of next word; `W` ignores punctuation
`b` or `B`	Start of previous word; `B` ignores punctuation
`e` or `E`	End of next word; `E` ignores punctuation
`0` (zero) or |	First column in current line
n|	Column *n* in current line
`^` (caret)	First non-blank character in current line
`$`	Last character in current line
+ or Return	First character in next line
`-`	First non-blank character in previous line
`1G`	First line in file
`G`	Last line in file
`G$`	Last character in file
n`G`	Line *n* in file
`(`	Back to beginning of sentence
`)`	Forward to beginning of next sentence
`{`	Back to beginning of paragraph
`}`	Forward to beginning of next paragraph



Section Positioning

Mark sections by placing { in first column.

Searching

%	Search to beginning of balancing ( ) [ ] or { }
*f*char	Search forward in current line to *char*
F*char*	Search backward in current line to *char*
t*char*	Search forward in current line to character before *char*
T*char*	Search backward in current line to character after *char*
/*str* Return	Find *str*
?*str* Return	Search in reverse for *str*
:set ic	Ignore case when searching
:set noic	Pay attention to case when searching (default)

Global Search and Replace

:*n*,*m*s/*str1*/*str2*/*opt*	Search from *n* to *m* for *str1*. Replace *str1* with *str2*, using *opt*. *opt* can be g for global change, c to confirm change (press y to acknowledge, Return to suppress), and p to print changed lines.
&	Repeat last :s command
:g/*str*/*cmd*	Run *cmd* on all lines that contain *str*
:g/*str1*/s/*str2*/*str3*/	Find line containing *str1*, replace *str2* with *str3*
:v/*str*/*cmd*	Execute *cmd* on all lines that do not match *str*

Copying and Placing Text

*n*yy or *n*Y	Yank *n* lines (place in buffer); omitting *n* yanks current line
y*cursor_cmd*	Yank from cursor to *cursor_cmd* (e.g., yG yanks current line to last line in file)
"*(a-z)n*yy or "*(a-z)n*dd	Copy or delete *n* lines into named buffer *a* through *z*; omit *n* for current line
p (lower-case)	Put yanked text after cursor (print buffer); also prints last deleted text
P	Put yanked text before cursor; also prints last deleted text
"*(a-z)*p or "*(a-z)*P	Put lines from named buffer *a* through *z* after or before current line

Changing Text

Preceding these commands with *n* (a number) repeats the command *n* times.

r*char*	Replace current character with *char*
R*text* ESC	Replace current character(s) with *text*
s*text* ESC	Substitute *text* for current character
S or cc *text* ESC	Substitute *text* for entire line
cw*text* ESC	Change current word to *text*
C*text* ESC	Change rest of current line to *text*
c*cursor_cmd text* ESC	Change to *text* from current position to *cursor_cmd*

Joining Lines

J	Join next line to end of current line
*n*J	Join next *n* lines

Cursor Placement and Adjusting the Screen

H	Move cursor to top line of screen
*n*H	Move cursor to line *n* from top of screen
M	Move cursor to middle of screen
L	Move cursor to bottom line of screen
*n*L	Move cursor to Line *n* from bottom of screen
CTRL-e	Move screen up one line
CTRL-y	Move screen down one line
CTRL-u	Move screen up ½ page
CTRL-d	Move screen down ½ page
CTRL-b	Move screen up one page
CTRL-f	Move screen down one page
CTRL-l (letter "l")	Redraw screen
z Return	Make current line top line on screen
*n*z Return	Make line *n* top line on screen
z.	Make current line middle line
*n*z.	Make line *n* middle line on screen
z-	Make current line bottom line
*n*z-	Make line *n* bottom line on screen

Shell Escape

:! *cmd*	Execute sh these specia % name o # name o
:!!	Execute last
:r! *cmd*	Read and in
:f *file*	Rename cur
:w !*cmd*	Send curren input and ex
:cd *dir*	Change the ($HOME is d
:sh	Start a sub-
:so *file*	Read and e shell script)

Shell

!*cursor_cmd cmd*	Send text fr to shell con in file with c
!}sort Return	Example: S paragraph a

Macros and

:map *key cmd_seq*	Define ke
:map	Display a
:unmap *key*	Remove
:ab *str string*	When str
:ab	Display a
:una *str*	Unabbre

Map allows you to define stri
.exrc to run each time you e
notimeout option. If you em
like ESC in the macro, you nee
If you need to include quotes ("
Unused keys in vi are: K V g

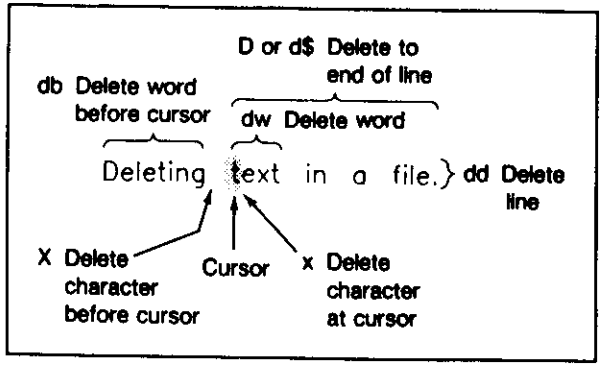
Example:

:map v /I CTRL-v ESC

When v is pressed, search for "
insert "You" (iYou ESC). CTRL

Deleting Text

CTRL-h or Back space	While inserting, delete previous character
CTRL-w	While inserting, delete previous word
CTRL-x	While inserting, delete to start of inserted text
*n*x	Delete *n* characters beginning with current; omitting *n* deletes current character
*n*X	Delete previous *n* characters; omitting *n* deletes previous character
xp	Switch character at cursor with following character
*n*dw	Delete next *n* words beginning with current; omitting *n* deletes current word
*n*db	Delete previous *n* words; omitting *n* deletes previous word
*n*dd	Delete *n* lines beginning with current; omitting *n* deletes current line
:*n*,*m*d	Delete lines *n* through *m*
D or d$	Delete from cursor to end of current line
d*cursor_cmd*	Delete text to *cursor_cmd* (e.g., dG deletes from current line to end of file)



Placing Marks in the Text

m*(a-z)*	Mark current position with a letter a through z (e.g., ma)
'*(a-z)*	Move cursor to position *(a-z)* (e.g., 'a)
'' or `` (single quotes or grave accents)	Move cursor to location before last / ? or G command

Pattern Matching

Pattern matching characters help find strings with similar characteristics.

:set magic	Allow pattern matching with special characters (default)
:set nomagic	Allow only ^ and $ as special characters
^ (caret)	Match beginning of line
$	Match end of line
.	Match any single character
\<	Match beginning of word
\>	Match end of word
[*str*]	Match any single character in *str*
[^*str*]	Match any character not in *str*
[*a-n*]	Match any character between *a* and *n*
*	Match zero or more occurrences of previous character in expression
\	Escape meaning of next character (e.g., \$ lets you search for $)
\\	Escape the \ character

Indenting Text

CTRL-i or Tab	While inserting, insert one shift width
:set ai	Turn on auto-indentation
:set sw=*n*	Set shift width to *n* characters
n<< or *n*>>	Shift *n* lines left or right (respectively) by one shift width; omitting *n* shifts one line
< or >	Use with cursor command to shift multiple lines left or right

HEWLETT PACKARD

HP Part Number
98597-90000
Microfiche No. 98597-99000
Printed in U.S.A. 9/87

Quick Refer

This quick reference lists commar
on Hewlett-Packard's UNIX™ Syst

Notat

- Commands beginning with
  These commands represer
  editor. You can reference t
- *file* is the name of file
- *cursor_cmd* is a cursor mov
  (e.g., G j w b)
- *char* is a single character
- *str* is a character string (car
  acters)
- CTRL-x means you pres
  the x key.
- *n*,*m* can be two line numbe
  . , $), or search expression
- *(a-z)* means you choose a l

Mod

Command Mode	When you are you can move (e.g., searchir tion to the ca Caps lock ke
Insert Mode	When you in character of mode comma press ESC.