PRO/VENIX™

UNIX™
for

Professional™

User Guide

digital

software

PRO/VENIX™
for the Professional

User Guide

Developed by:

VenturCom, Inc.
215 First Street
Cambridge, MA 02142

Digital Equipment Corporation
Maynard, MA 01754

First Printing

DEC	DECwriter	Professional	VAX
DECmate	DIBOL	Rainbow	VMS
DECnet	MASSBUS	RSTS	VT
DECsystem-10	PDP	RSX	Work Processor
DECSYSTEM-20	P/OS	UNIBUS	digital
DECUS			

The PRO/VENIX† Documentation Set

The PRO/VENIX documentation set consists of the following manuals:

PRO/VENIX Installation and System Manager's Guide

The set up and maintenance of PRO/VENIX are described in the installation sections. Other articles explain the UNIX-to-UNIX‡ communications systems. The ''System Maintenance Reference Manual'' contains reference pages for devices and system maintenance procedures (sections (7) and (8)).

PRO/VENIX User Guide

The *User Guide* contains tutorials for newcomers to PRO/VENIX, covering basic use of the system, the editor **vi** and use of the command language interpreters.

PRO/VENIX Document Processing Guide

The line and screen editors and **nroff**-related text formatting utilities are described in the Document Processing Guide. Topics include: line editor **ed**, and stream editor **sed**; the text formatter **nroff**; the **nroff**-preprocessors **tbl** and **neqn**.

PRO/VENIX Programming Guide

The chapters in the *Programming Guide* explicate the different programming languages for VENIX.

† VENIX is a trademark of VenturCom, Inc.

‡ UNIX is a trademark of Bell Laboratories.

PRO/VENIX Support Tools Guide

This guide includes tools for programming, such as the compiler-writing languages Yacc and Lex, the M4 Macro processor, the program development utility Make, and the desk calculator programs DC and BC.

PRO/VENIX User Reference Manual

This is a complete and concise reference for the PRO/VENIX system. This volume contains write-ups on all PRO/VENIX commands.

PRO/VENIX Progammer Reference Manual

The reference pages in this volume include system calls, library functions, file formats, miscellaneous functions and games.

Contents

INTRODUCTION

Chapter 1. VENIX FOR BEGINNERS

Chapter 2. DISPLAY EDITING WITH VI

Chapter 3. AN INTRODUCTION TO THE SHELL

Chapter 4. AN INTRODUCTION TO THE C SHELL

INTRODUCTION

The *PRO/VENIX User Guide* covers the following topics:

The first chapter, VENIX FOR BEGINNERS, is an introduction to VENIX and instructions on how to use the operating system.

AN INTRODUCTION TO DISPLAY EDITING WITH VI is a guide to the interactive screen editor, **vi**.

Chapter three, AN INTRODUCTION TO THE SHELL, is an overview of the capabilities of the shell command interpreter.

The chapter AN INTRODUCTION TO THE C SHELL describes the features of the C shell command command interpreter, including a glossary of commands.

Not all of the capabilities of the operating system are described or illustrated here, but enough are described so that a new user can become familiar with the use of the VENIX operating system.

Throughout this volume, references are made to pages in the *User Reference Manual*, *Programmer Reference Manual* and *Installation and System Manager's Guide*. These are often in the form of a name followed by a section number, e.g **fsck**(1). Entries with "name" (1) are contained in the *User Reference Manual*. References for sections (2)–(6) are included in the *Programmer Reference Manual*. Sections (7) and (8) are found in the *Installation and System Manager's Guide*.

Contents

1.1 INTRODUCTION 1-1

1.2 GETTING STARTED 1-3

1.3 DAY-TO-DAY USE 1-10

1.4 DOCUMENT PREPARATION 1-39

1.5 PROGRAMMING 1-42

1.6 VENIX READING LIST 1-47

Chapter 1

VENIX† For Beginners

This paper is meant to help new users get started on the VENIX operating system. It includes:

- basics needed for day-to-day use of the system — typing commands, correcting typing mistakes, logging in and out, mail, inter-terminal communication, the file system, printing files, redirecting I/O, pipes, and the shell.
- document preparation — a brief discussion of the major formatting programs and macro packages, hints on preparing documents, and capsule descriptions of some supporting software.
- VENIX programming — using the editor, programming the shell, programming in C, other languages and tools.
- An annotated VENIX bibliography.

1.1 INTRODUCTION

The VENIX Operating System is derived from Bell Labs' UNIX‡ System III. In many ways, VENIX is the state of the art in computer operating systems. From the user's point of view, it is easy to learn and use, and presents few of the usual impediments to getting the job done. It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. This paper will familiarize new users with the main ideas of the VENIX

† VENIX is a trademark of VenturCom, Inc.

‡ UNIX is a trademark of Bell Laboratories.

system so that they can start using it quickly and effectively. If you already have experience with UNIX, you will find that VENIX is quite similar.

In this article, you will find references to the *User Reference Manual*. In the manual you will find a more detailed explanation of the commands. In fact, the manual provides the best discussion of many topics so at several points in the text, we will refer you to it for a specific topic.

The five sections of this chapter are:

1. Getting Started: Instructions for logging in, typing, correcting typing mistakes, and logging out.

2. Day-to-day Use: Discussion of the file and directory systems, and commands to manipulate them effectively.

3. Document Preparation: Advice, but not extensive instructions, on using formatting tools to prepare manuscripts – one of the most common uses for VENIX.

4. Writing Programs: Discussion of some of the excellent tools for developing programs on VENIX; this is not a tutorial in any of the programming languages provided by the system.

5. A VENIX Reading List: An annotated bibliography of documents that should be helpful to new users.

In the following sections, the notation CTRL-X is generated by pressing down the CONTROL key while typing the letter X (either lower or upper case may be used). This is used to perform several special functions, as described in the following pages. (See also Appendix A.)

1.2 GETTING STARTED

1.2.1 Logging In

The system administrator will have to give you a valid user name before you can "login" on VENIX. Using a password is optional on this system; however, some administrators require all users to have one. If you are going to use a password, make sure it is easy to type and remember.

On your terminal you should see the word "login:". If not, it may mean that the previous user did not "log out" before leaving. To "log out" the previous user, type CTRL-D by holding down the CONTROL key while typing the letter D. Press the CR key. If you still cannot get the "login" message to appear, refer to "SETTING UP VENIX" in the *Installation and System Manager's Manual*.

If you are accessing the system, via a modem, on a line which may operate at different speeds (for example, a 300/1200 baud dial-in line), you may have to get VENIX to the correct speed before you receive the "login:". This can be done by pressing the BREAK key, which signals VENIX to try a different baud rate, and attempting to type. (If your terminal is at the wrong baud rate, keys you type will either not be shown or will appear as random letters.)

When the terminal displays "login:" type your user name in lower case letters and press CR. If you type upper case letters, VENIX will assume that your terminal cannot generate lower case letters and will display all subsequent output in upper case. VENIX is strongly oriented towards devices with lower case.

If you were assigned or chose a password, the system will ask for it after you have typed your user name. When the terminal displays "password:" type your password and CR. For security, the word will not appear as you type it. Getting a "login incorrect" message can mean you mistyped your user name or password. Press CR and start the "login" process again. Users who do not have a password should not get a "password:" request. If you are prompted for a password but you never chose nor were assigned one, then you probably mistyped your user name. Type a CR and start again from "login:".

Assuming that your login efforts are successful you will get a "Welcome to VENIX" message with the date, followed by a prompt (a word or character that means the system is waiting for you to type a command). Before the prompt appears, you may get some daily messages or a notice you have mail (see the following section on "mail" to learn how to send or read mail). In VENIX, the prompt will be a dollar sign "$" unless you take steps to personalize it.

1.2.2 Typing Commands

Once you've seen the prompt character, you can type commands, which are requests that the system do something. Try typing

```
date
```

followed by CR. You should get back something like

```
Tue Jan 12 14:17:10 EST 1983
```

Don't forget to press the CR key after the command, or nothing will happen. If you think you're being ignored, type CR; something should happen. 'CR' won't be mentioned again, but don't forget it. In VENIX, it has to be typed at the end of each line before the system gets the input.

Another command you might try is **pwd** ("print working directory") which gives your current directory location.

```
pwd
```

gives something like

/**usr**/*yourname*

(The name listed should be your own.)

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

```
pqd
```

you will be told

```
pqd: not found
```

Of course, if you inadvertently type the name of some other command, it will run, with more or less mysterious results.

All commands must be given in lower-case, so for example typing

```
PWD
```

will give you only

```
PWD: not found
```

A description of the kinds of error messages you may receive while attempting to run commands may be found in a later section of this document.

1.2.3 Mistakes in Typing

If you make a typing mistake, and see it before CR has been typed, there are two ways to recover. The DELETE key erases the last character typed. Successive uses of DELETE erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go.

A CTRL-U (typed by pressing the CONTROL key and typing 'u') erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type a CTRL-U and start it over.

The BACKSPACE key will *not* cause characters to be deleted. Since typing a back-space will cause a CRT terminal to overwrite a character, it may *appear* that VENIX is deleting a letter; however, this does not in fact cause the letter to be deleted from VENIX's input stream, and VENIX will act as if you made any other typing error. Newcomers should be careful to avoid this, as imbedding back-spaces in commands will cause confusing errors. If you try to use back-spaces (or any other control character) in a file name, VENIX will replace each occurrence in the name with a '#'.

1.2.4 Read-ahead

VENIX has full read-ahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So you can type several commands one after another without waiting for the first to finish or even begin.

If what you are typing gets garbled on the screen by other output, hitting CTRL-R will re-echo everything you have typed that has not yet been read by a program. To erase *all* of this, type CTRL-E; it is like CTRL-U, except that all previous unread lines, not just the current one, are erased.

1.2.5 Stopping a Program

You can stop most programs by typing a CTRL-C. There are some exceptions: in a few programs, like in the line-oriented text editor **ed**, CTRL-C stops whatever the program is doing but leaves you in that program. The BREAK or INTERRUPT key has the same effect. Screen editors act completely differently, and generally have their own unique control codes to abort or exit.

If your program is sending out information to your terminal screen faster than you can read it, typing CTRL-S will stop the output. The program itself isn't terminated; it just waits there until you tell it to continue typing to your terminal. This is accomplished by typing CTRL-Q, which starts things up again. Typing any other character will also start output, but that character will be entered and read.

You can also tell VENIX to *automatically* stop after every 20 lines of uninterrupted output, just as if you had typed CTRL-S. Output is continued by typing CTRL-Q. This mode of output is called "scroll" mode, and is enabled by typing

```
stty scroll
```

Once you do this, don't forget that all programs (except for screen editors) will stop every 20 lines unless you type the CTRL-Q.

If you want "scroll" mode to be in effect every time you log in, the above **stty** command can be placed in the file called **.profile**. The **.profile** file is read by the system every time you log in, and the commands within it automatically executed, just as if you typed them at the keyboard.

If you decide you don't like the "scroll" mode, you can turn it off by typing at any time

```
stty -scroll
```

1.2.6 Logging Out

The easiest way to log out is to type CTRL-D (EOT — End Of Transmission). VENIX will display a new "login:" message. You can also use the **login** command to log out by simply typing

```
login
```

or to log in to a new user by typing

login *new-user-name*

See Appendix A for a list of the terminal control codes that were discussed in the above paragraphs.

1.2.7 Strange Terminal Behavior

Sometimes you can get into a situation where your terminal acts strangely. For example, characters may not be echoed back to you as you type, or CR may cause the cursor to move down a line (i.e., do a line feed) but not return to the left margin. You can often fix this by logging out and logging back in. Or you can read the description of the command **stty** in the *User Reference Manual* and use it to reset the correct terminal mode. This can often be done by typing

```
stty -nl -raw echo
```

and ending the line with a line feed instead of a carriage return.

If your terminal echoes '#' signs when you delete characters, instead of erasing the characters from the screen, then you can type

```
stty crt
```

to fix it. If your terminal is typing everything in upper-case, you can bring it back to the normal upper- and lower-case state by typing

```
stty -lcase
```

VENIX will put you in upper-case only mode if you accidentally typed your login name in upper-case letters.

1.2.8 Mail

When you log in, you may sometimes get the message

```
You have mail.
```

VENIX provides a postal system so you can communicate with other users of the system. To read your mail, type the command

```
mail
```

Your mail will be printed, one message at a time, most recent message first. After each message, **mail** prompts you with a '?' and waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and CR, which does not (so it will still be there the next time you read your mailbox). Responding with a '?' yourself will get you a list of all the choices, which are of course also listed in the *User Reference Manual*.

How do you send mail to someone else? Suppose it is to go to "sam" (assuming "sam" is someone's login name). The easiest way is this:

```
mail sam
now type in the text of the letter
on as many lines as you like ...
After the last line of the letter
type CTRL-D.
```

And that's it. The CTRL-D sequence is used throughout VENIX to mark the end of input from a terminal, so you might as well get used to it. The CTRL-D must immediately follow the last newline. Be careful though: if you type a CTRL-D when there is no command to read it, your shell will find it and log you out.

For practice, send mail to yourself. (This isn't as strange as it might sound — mail to oneself is a handy reminder mechanism.)

There are other ways to send mail — you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail**(1). (The notation **mail**(1) means the command **mail** in section 1 of the *User Reference Manual*.

1.2.9 Writing to other users

At some point, out of the blue will come a message like

```
Message from joe tty07...
```

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won't be able to talk back. To respond, type the command

```
write joe
```

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to terminate or be terminated.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it's like this:

> **Joe types write smith and waits.**
>
> **Smith types write joe and waits.**
>
> **Joe now types his message (as many lines as he likes). When he's ready for a reply, he signals it by typing (o), which stands for "over".**
>
> **Now Smith types a reply, also terminated by (o).**
>
> **This cycle repeats until someone gets tired; he then signals his intent to quit with (oo), for "over and out".**
>
> **To terminate the conversation, each side must type a CTRL-D character alone on a line. (CTRL-C also works.) When the other person types his CTRL-D, you will get the message EOF on your terminal.**

If you write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, simply type CTRL-D.

1.3 DAY-TO-DAY USE

1.3.1 Creating Files — The Editor

If you have to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with one of the VENIX text editors. You will probably find that one of the screen editors, like **vi** or the **FinalWord**, is most convenient for practically all the editing work you have. The **FinalWord**, an optional editor with VENIX, is described in a separate manual. However, a basic knowledge of **ed** can be handy, and at times indispensable (its global substitution facilities do not exist in the screen editors); and since **ed** is the most widely known of the text editors, we'll assume its use throughout the rest of this paper. **ed** is thoroughly documented in **ed**(1).

All we want it for right now is to make some files. (A file is just a collection of information stored in the machine, a simplistic but adequate definition.) To create a file called "junk" with some text in it, do the following:

```
ed junk  (invokes the text editor)
a        (command to "ed", to add text)
now type in
whatever text you want ...
.        (signals the end of adding text)
```

The "**.**" that signals the end of adding text must be at the beginning of a line by itself. Don't forget it, for until it is typed, no other **ed** commands will be recognized — everything you type will be treated as text to be added.

At this point you can do various editing operations on the text you typed in, such as correcting spelling mistakes, rearranging paragraphs and the like. Finally, you must write the information you have typed into a file with the editor command **w**:

```
w
```

ed will respond with the number of characters it wrote into the file **junk**.

Until the **w** command, nothing is stored permanently, so if you quit and go home the information is lost. But after **w** the information is there permanently; you can re-access it any time by typing

```
ed junk
```

Type a **q** command to quit the editor. (If you try to quit without writing, **ed** will print a **?** to remind you. Typing a second **q** gets you out regardless.)

Now create a second file called **temp** in the same manner. You should now have two files, **junk** and **temp**.

1.3.2 What Files Are Out There?

The **ls** (for "list") command lists the names (not contents) of any of the files that VENIX knows about. If you type

```
ls
```

the response will be

```
junk      temp
```

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command (note the space before the argument **−t**):

```
ls -t
```

causes the files to be listed in the order in which they were last changed, most recent first. The **−l** option gives a "long" listing:

```
ls -l
```

will produce something like

```
-rw-rw-r--  1 fred  41 Jul 22 2:56 junk
-rw-rw-r--  1 fred  78 Jul 22 2:57 temp
```

(Since long listings are so frequently useful, a version of the **ls** command called simply **l** will give one to you automatically.) The date and time are of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from **ed**). **fred** is the owner of the file, that is, the person who created it. The **−rw−rw−r−−** tells who has permission to read and write the file; see the section on "Protection Modes" later for a full description.

Options can be combined: **ls −lt** gives the same thing as **ls −l**, but sorted into time order. You can also name the files you're interested in, and **ls** will list the information about them only. More details can be found in **ls**(1).

The use of optional arguments that begin with a minus sign, like **−t** and **−lt**, is a common convention for VENIX programs. In general, if a program accepts such optional arguments, they precede any filename arguments. It is also vital that you separate the various arguments with spaces: **ls−l** is not the same as **ls −l.**

1.3.3 Printing Files

Now that you've got a file of text, how do you print it so people can look at it? There are a host of programs to do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

```
ed junk
1,$p
```

ed will reply with the count of the characters in **junk** and then print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (several thousand lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after another. So here are a couple of alternatives.

First is **cat,** the simplest of all the printing programs. **cat** simply prints on the terminal the contents of all the files named in a list. Thus

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two. The files are simply concatenated (hence the name **cat**) onto the terminal.

pr produces formatted printouts of files. As with **cat, pr** prints all the files named in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will print **junk** neatly, then skip to the top of a new page and print **temp** neatly.

pr can also produce multi-column output:

```
pr -3 junk
```

prints **junk** in 3-column format. You can use any reasonable number in place of "3" and **pr** will do its best. **pr** has other capabilities as well; see **pr**(1).

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatter is **nroff**, which we will get to in the section on document preparation.

There is also a program called **lpr** that queues files and prints them (with page headers and numbers) on a printer. The command

```
lpr junk
```

will print out file **junk**.

If the information you want printed isn't in a file (for example, a directory listing obtained through the **ls** command), you can send it to the **lpr** (or most other commands) through a "pipe," described in a later section.

1.3.4 Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving it a new name), like this:

```
mv junk precious
```

This means that what used to be "junk" is now "precious". If you do an **ls** command now, you will get

```
precious
temp
```

Beware that if you move a file to another one that already exists, the already existing file's contents are lost forever.

If you want to make a *copy* of a file (that is, to have two versions of something), you can use the **cp** command:

```
cp precious temp1
```

makes a duplicate copy of **precious** in **temp1**.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

```
rm temp temp1
```

will remove both of the files named.

You will get a warning message if one of the named files wasn't there, but otherwise **rm**, like most VENIX commands, does its work silently. There is no prompting or chatter, and error messages are occasionally curt. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

1.3.5 What's in a Filename

So far we have used filenames without ever saying what's a legal name, so it's time for a couple of rules. First, filenames are limited to 14 characters, which is enough to be descriptive. Second, although you can use almost any character in a filename, common sense says you should avoid characters that might be used with other meanings. We have already seen, for example, that in the **ls** command, **ls −t** means to list in time order. So if you had a file whose name was **−t**, you would have a tough time listing it by name.

Besides the minus sign, there are other characters which have special meaning. In addition, all control characters (back-spaces, form-feeds, etc.) are absolutely forbidden. (VENIX enforces this rule by substituting any of those characters with pound signs ('#') each time they occur in file names.) To avoid pitfalls, you would do well to use only letters, numbers and the period until you're familiar with the situation.

On to some more positive suggestions. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for **ed** will not handle really big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

```
chap1
chap2
etc...
```

Or, if each chapter were broken into several files, you might have

```
chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...
```

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice VENIX user. What if you wanted to print the whole book? You could say

```
pr chap1.1 chap1.2 chap1.3 ......
```

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

```
pr chap*
```

The * means ''anything at all,'' so this translates into ''print all files whose names begin with **chap**, listed in alphabetical order.

This shorthand notation is not a property of the **pr** command, by the way. It is system-wide, a service of the program that interprets commands (the ''shell,'' see **sh**(1)). Using that fact, you can see how to list the names of the files in the book:

```
ls chap*
```

produces

```
chap1.1
chap1.2
chap1.3
...
```

The * is not limited to the last position in a filename — it can be anywhere and can occur several times. Thus

```
rm *junk* *temp*
```

removes all files that contain **junk** or **temp** as any part of their name. As a special case, * by itself matches every filename, so

```
pr *
```

prints all your files (alphabetical order), and

```
rm *
```

removes *all files*. (You had better be *very* sure that's what you wanted to say!)

The * is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9. Then you can say

```
pr chap[12349]*
```

The **[...]** means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated, so you can also do this with

```
pr chap[1-49]*
```

Letters can also be used within brackets: **[a−z]** matches any character in the range **a** through **z**.

The **?** pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls -l chap?.1
```

lists information about the first file of each chapter (**chap1.1**, **chap2.1**, etc.).

Of these niceties, * is certainly the most useful, and you should get used to it. The others are frills, but worth knowing.

If you should ever have to turn off the special meaning of *, ?, etc., enclose the entire argument in single quotes, as in

```
ls '?'
```

We'll see some more examples of this shortly.

1.3.6 What's in a Filename, continued

When you first made that file called **junk**, how did the system know that there wasn't another **junk** somewhere else, especially since the person in the next office is also reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to him. When you log in, you are "in" your directory. Unless you take special action, when you create a new file, it is made in the directory that you are currently in; this is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files is organized into a large tree (or "hierarchy") with your files located several branches into the tree. It is possible for you to "walk" around this tree, and to find any file in the system, by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start where you are and walk toward the root.

Let's try the latter first. The basic tool is the command **pwd** ("print working directory"), which prints the name of the directory you are currently in.

Although the details will vary according to the system you are on, if you give the command **pwd**, it will print something like

```
/usr/your-name
```

This says that you are currently in the directory **your-name**, which is in turn in the directory **/usr**, which is in turn in the root directory called by convention just /. (Even if it's not called **/usr** on your system, you will get something analogous. Make the corresponding changes and read on.)

If you now type (note the space after the command "ls"):

```
ls /usr/your-name
```

you should get exactly the same list of file names as you get from a plain **ls**: with no arguments, **ls** lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Next, try

```
ls /usr
```

This should print a series of names, among which is your own login name **your-name**.

The next step is to try

```
ls /
```

You should get a response something like this (although again the details may be different):

```
.profile   bin     dev     etc
lib        tmp     usr     venix
```

This is a collection of the basic directories that the system knows about (plus a few plain files); we are at the root of the tree.

Now try

cat /usr/your-name/junk

(if **junk** is still around in your directory). The name

/usr/your-name/junk

is called the **pathname** of the file that you normally think of as "junk". "Pathname" has an obvious meaning: it represents the full name of the path you have to follow from the root through the tree of directories to get to a particular file. It is a universal rule in the VENIX system that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:

```
              (root)
              / | \
             /  |  \
            /   |   \
           /    |    \
   etc    bin    usr    tmp    dev
  / | \  / | \  / | \  / | \  / | \
             /  |  \
            /   |   \
        adam   eve   mary
        /     /  \      \
             /    \    junk
         junk     temp
```

Notice that Mary's **junk** is unrelated to Eve's. For a look through an entire typical VENIX system, see "VENIX MAINTENANCE" in the *Installation and System Manager's Guide.*

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by saying

```
pr /usr/your-name/chap*
```

Similarly, you can find out what files your neighbor has by saying

```
ls /usr/neighbor-name
```

or make your own copy of one of her files by

```
cp /usr/your-neighbor/her-file yourfile
```

If your neighbor doesn't want you poking around in her files, or vice versa, privacy can be arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which can be set to control access. (See the section on "Protection Modes" later in this chapter for details.) As a matter of observed fact, for most users, openness is usually more beneficial than privacy.

As a final experiment with pathnames, try

```
ls /bin /usr/bin
```

Do some of the names look familiar? When you run a program, by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically doesn't find it), then in **/bin** and finally in **/usr/bin**. There is nothing magic about commands like **cat** or **ls**, except that they have been collected into a couple of places to be easy to find and administer.

What if you work regularly with someone else on common information in her directory? You could just log in as your friend each time you want to, but you can also say "I want to work on her files instead of my own". This is done by changing the directory that you are currently in (note the space after "cd"):

```
cd /usr/your-friend
```

Now when you use a filename in something like **cat** or **pr**, it refers to the file in your friend's directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact. Of course, if you forget what directory you're in, type

```
pwd
```

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when you write your book, you might want to keep all the text in a directory called **book**. So make one with

```
mkdir book
```

then go to it with

```
cd book
```

then start typing chapters. The book is now found in (presumably)

```
/usr/your-name/book
```

To get back to your original (often called "home") directory, just type

```
cd
```

From here, you can list the contents of directory **book** by typing

```
ls book
```

If a file **appendix** exists under **book**, you can look at it with the commands

```
cd book
cat appendix
cd
```

or you could (more concisely) type

```
cat book/appendix
```

The name "book/appendix" is simply the pathname to **appendix**, starting from your home directory. Remember: any place you can use a filename, you can use a pathname. We could have referred to **appendix** by its full pathname from the root, as in

```
cat /usr/your-name/book/appendix
```

but since we are already in directory **/usr/your-name**, there is no point in using this longer name.

VENIX understands that pathnames beginning with a "/" are being specified from the root of the tree downwards, while those *not* beginning with "/" are from the current directory.

You aren't limited to one level of directories. If you care to, you can make further directories under **book**. For example, if you are in directory **book** and type:

```
mkdir part1 part2 part3
```

you will have three directories in **book**. From here, typing

```
cd part1
```

moves you into directory **part1**. Now typing

```
pwd
```

should bring the response

```
/usr/your-name/book/part1
```

as you might have guessed. Your branch of the directory tree now looks like this

```
           |
       your-name
           |
           |
           |
         book
        /  |  \
       /   |   \
    part1 part2 part3
```

How do you move up a level in the tree, back up to directory **book**? There are several choices. The command

cd

by itself will move you back to your home directory; from there, the command

cd book

will bring you down to **book**. Alternatively, since you know the full pathname of **book**, you could type in the single command

cd /usr/your-name/book

These are both a bit awkward, however, and fortunately VENIX allows some convenient shorthand: the name ".." always refers to the directory above you. Thus the task of moving up to **book** could have been done more easily with the command

cd ..

This leads to further conveniences. Suppose, for example, that you are back in directory **part1**, and wish to move to directory **part2**. Using the ".." name, we could do this with

cd ..
cd part2

But there is no reason why you have to change directories one level at a time. Instead, these two commands can be condensed to the single

```
cd ../part2
```

The "/" is the standard directory name separator: ".." refers to the directory above you, and "part2" refers to the directory below that one, so "../part2" gets you right there.

The use of ".." is not limited to the **cd** command. For example, suppose your current directory was **part1**, and you wished to **cat** a file **chap** in directory **part2**. You could type

```
cd ..
cat part2/chap
cd part1
```

But why bother to change directories? These three commands can be condensed to

```
cat ../part2/chap
```

Remember: any place you use a file name, you can use a pathname. "../part2/chap" is simply the pathname to file **chap** from your current directory. ".." is just a directory name, and can be used in pathnames just like any other directory name.

To remove the directories **part1**, **part2**, **part3**, type

```
rm part1/* part2/* part3/*
rmdir part1 part2 part3
```

The first command removes all files from these directories; the second removes the empty directories. (You can't remove a directory unless it has been emptied first.) These commands could also be given, using the "[]" notation, as

```
rm part[1-3]/*
rmdir part[1-3]
```

The other special directory name, in addition to "..", is simply ".". This always refers to the directory you're in. This can be very useful with the **cp** (copy file) and **mv** (move file) commands. As shown before, the commands

```
cp file1 file2
```

or

```
mv file1 file2
```

can be used to copy or move, respectively, **file1** to **file2**. Using pathnames, you could also say

```
cp /usr/neighbor-name/his-file yourfile
```

Frequently, it is useful to copy or move files from one directory to another, without changing their name. If you have some directory "yourdir", the command

```
cp /usr/neighbor-name/his-file yourdir
```

will copy the file **his-file** to directory **yourdir**, leaving you with a file **yourdir/his-file**. The command

```
mv book/part1/chap book
```

moves the file **chap** from directory **book/part1** to directory **book**, leaving you with a file **book/chap**. Now, since "." refers to your current directory, you can say

```
cp /usr/neighbor-name/his-file .
```

or

```
mv /usr/neighbor-name/his-file .
```

and copy or move a file to your current directory, without changing its name.

If you use **cp** or **mv** to copy or move a directory, you can also specify more than one file at a time. For example,

```
cp /usr/neighbor-name/his-file /usr/neighbor-name/his-other-file yourdir
```

which copies two files to **yourdir**, or

```
mv /usr/neighbor-name/* .
```

which moves *all* files in **/usr/neighbor-name** to your current directory.

1.3.7 Using Files Instead of the Terminal

Most of the commands we have seen so far produce output on the terminal; some, like the editor, also take their input from the terminal. It is universal in VENIX systems that the terminal can be replaced by a file for either or both input and output. As one example,

```
ls
```

makes a list of files on your terminal. But if you say

```
ls >filelist
```

a list of your files will be placed in the file **filelist** (which will be created if it doesn't already exist, or overwritten if it does). The symbol > means "put the output on the following file, rather than on the terminal." Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of **cat** in a file:

```
cat f1 f2 f3 >temp
```

The symbol >> operates very much like > does, except that it means "add to the end of." That is,

```
cat f1 f2 f3 >>temp
```

means to concatenate **f1**, **f2** and **f3** to the end of whatever is already in **temp,** instead of overwriting the existing contents. As with >, if **temp** doesn't exist, it will be created for you.

In a similar way, the symbol < means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called **script.** Then you can run the script on a file by saying

```
ed file <script
```

As another example, you can use **ed** to prepare a letter in file **let**, then send it to several people with

```
mail adam eve mary joe <let
```

1.3.8 Pipes

One of the novel contributions of the VENIX system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipeline.

For example,

```
pr f g h
```

will print the files **f**, **g**, and **h**, beginning each on a new page. Suppose you want them run together instead. You could say

```
cat f g h >temp
pr <temp
rm temp
```

but this is more work than necessary. Clearly what we want is to take the output of **cat** and connect it to the input of **pr**. So let us use a pipe:

```
cat f g h | pr
```

The vertical bar | is the pipe symbol; it means take the output from **cat**, which would normally have gone to the terminal, and put it into **pr** to be neatly formatted.

There are many other examples of pipes. For example,

```
ls -m | sort -r
```

prints a single-column list of your files, sorted in reverse alphabetical order.

```
ls -m | sort -r | lpr
```

sends that list to the printer via the **lpr** command.

The program **wc** counts the number of lines, words and characters in its input, and as we saw earlier, **who** prints a list of currently-logged-on people, one per line. Thus

```
who | wc
```

tells how many people are logged on (the first number shown is the line count). And of course

```
ls | wc
```

counts your files.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many VENIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines. **pr** is one example:

```
pr -3 a b c
```

prints files **a, b** and **c** in order in three columns. But in

```
cat a b c | pr -3
```

pr prints the information coming down the pipeline, still in three columns.

1.3.9 The Shell

We have already mentioned once or twice the mysterious "shell" (see **shell(1)**) **sh**(1). The shell is the program that interprets what you type as commands and arguments. It also looks after translating *, etc., into lists of filenames, and <, >, and | into changes of input and output streams.

The shell has other capabilities too. For example, you can run two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a prompt character.

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don't want to wait around for the results before starting something else, you can say

```
ed file <script &
```

The ampersand at the end of a command line says "start this command running, then take further commands from the terminal immediately," that is, don't wait for it to complete. Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you're doing on the terminal, it would be better to say

```
ed file <script >script.out &
```

which saves the output lines in a file called **script.out**.

When you initiate a command with **&**, the system replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

```
kill process-number
```

If you forget the process number, the command **ps** will tell you about all processes running. (If you are desperate, **kill 0** will kill all your processes. The last process number you ran with an '&' is also known as "$!", so **kill $!** will stop just that one.)

You can say

```
(command-1; command-2; command-3) &
```

to start three commands in the background, or you can start a background pipeline with

```
command-1 | command-2 &
```

Just as you can tell the editor or some similar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell, after all, is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (**tabs**, **date**, **who**) into a file, let's call it **startup**, and then run it with

```
sh startup
```

This says to run the shell with the file **startup** as input. The effect is as if you had typed the contents of **startup** on the terminal.

If this is to be a regular thing, you can eliminate the need to type **sh**: simply type, once only, the command

```
chmod +x startup
```

and thereafter you need only say

```
startup
```

to run the sequence of commands. The **chmod**(1) command marks the file executable; the shell recognizes this and runs it as a sequence of commands.

If you want **startup** to run automatically every time you log in, create a file in your login directory called **.profile**, and place in it the line

```
startup
```

When the shell first gains control when you log in, it looks for the **.profile** file and does whatever commands it finds in it. We'll get back to the shell in the section on programming. See "An Introduction to the Shell" in this manual for more details on the shell.

1.3.10 Error Messages

There are many kinds of errors which can arise in day-to-day use of VENIX, but they can be divided into three classes: errors given by commands you run; errors given by the shell itself; and errors given by VENIX.

Command error messages vary widely. They are intended to be self-explanatory, but you may have to check the command write-up in the first section of the *User Reference Manual*, in particular the sub-heading "DIAGNOSTICS." A common error message, "Usage: ...", is given if a command is called with the wrong number of file names, flags, or other arguments; the "usage" message tries to give you an idea of how the command is supposed to be used. Errors are most frequently due to attempts to reference files which do not exist or which you have no permission to tamper with. The **access**(1) command can be used to check your permission regarding a file or directory. See the section in this chapter on "Permission Modes" for a description of its use.

The shell itself has a number of error messages which it will give you if you try to run a command that can not be found, redirect I/O in illegal ways, or commit one of several other felonies. Some of them refer to "signals", numbered 1 through 16. Signals are sent to programs which attempt, for example, to access non-existent memory, execute illegal machine instructions. Some signals result in a "core dump", which is the creation of a file called **core** containing an image of the running program. These core files may be used for debugging by sophisticated users; beginners can safely remove them.

Here is a list of the more common shell error messages, along with possible causes. A complete list can be found in **sh**(1). Some of them are given only when executing shell programs; others can be found in normal interactive use.

arg list too long

Too many arguments were passed to a command, usually due to use of "*" or other wildcards matching many file names. May be solved by not using long pathnames in wildcards. For example, instead of typing

```
pr /usr/fred/src/*
```

type

```
cd /usr/fred/src
pr *
```

Maximum number of characters passed as arguments is roughly 1500.

cannot create
File can not be created — you may not have write permission in directory, or file may already exist and not have write permission for you.

cannot execute
File can not be executed — no execute permission for you.

cannot make pipe
See system administrator — pipe device may be incorrectly setup.

cannot open
File can not be opened — does not exist, or no permission for you.

core dumped
Memory image of program dumped in file called "core". Occurs if program dies under certain conditions — commonly due to illegal memory access caused by a pointer bug.

Floating exception
Floating point error: due either to use or floating point instruction without hardware or simulator, or floating point error condition (e.g. divide by zero). If no floating point hardware exists, C programs must be compiled with −f flag.

not found
Given program can not be found. The shell looks in all directories given in the PATH variable, generally the current directory, /bin and /usr/bin.

you have mail
Good news, we hope. Run the "mail" command to read your mail.

Finally, there are low-level error messages given by VENIX itself. These result from problems in dealing with devices, or overflow conditions inside the operating system. These messages appear on the console terminal only. A list of these messages may be found in "VENIX MAINTENANCE" in the *Installation and System Manager's Manual*. Often these messages will be combined with error messages from running programs; for example, if you try to copy a file to a write-protected floppy diskette, the **cp** program will complain of errors in writing, while the device driver responsible for handling that disk unit will report problems with messages like "bn XXX ... err ....."

The most serious kind of system error message is that beginning with the word "PANIC ...". This indicates a situation from which VENIX can not recover, and precedes a total system halt.

1.3.11 File Protection Modes

When you do a long directory listing above, as in **ls −l** or just **l**, you get something like

```
-rw-rw-r--    1  joe    41 Sep 7     8:49   test
-rw-rw-r--    1  joe    78 Sep 9     9:23   temp
-rwxrwxr-x    1  joe  3050 Sep 9    13:23   a.out
```

The mysterious **−rw−rw−r−−** is the file's protection setting. All files under VENIX can be given three kinds of permission: read, write, and execute. "Read" permission allows one to read the contents of the file; "write" permission allows one to write on the file (either append to the file or overwrite it, even clear it to zero); and "execute" permission allows one to execute the file (which should be either a compiled program or a shell script). For each of these things, the file is protected for three classes of people: the owner, the group owner of the file, and everybody else in all other groups.

When a user tries to access a file, VENIX compares the user's ID and group ID with that of the file: if the user ID matches (that is, the user is the owner of the file), then the "owner" protection setting is used; otherwise, if the group ID matches (the file belongs to someone in your group) then the "group" protection setting is used; finally, if neither user or group ID matches, then the "other" setting is used. The **−rw−rw−r−−** shown above by the **l** command indicates read, write and execute permission for each of these three classes respectively: an '**r**' indicates read permission, a '**w**' write permission, and '**x**' execute permission. A '−' instead of a letter indicates that that permission has been denied.

If you can't access a file you think you should be able to, try the **access**(1) command, which is of the form

```
access file-name
```

Access tries to find the file, and checks its permission setting. It reports your permission for the file. If it can't find the file, it checks the pathname (if any) to it, and reports on how far it could get, or if the file exists at all.

When files are created, they are given a default permission setting. Normally this allows for reading and writing by the user and group, and reading only by others. Execute permission is only given if the file is really executable (such as a compiled program).

For file **temp** above, the owner can read and write to it, other people in the owner's group can read and write to it, and everyone else can read it, but *not* write to it. Nobody at all has execute permission (probably because this is not a compiled program). The file **a.out**, on the other hand, does have execute permission by everyone.

To change the file's mode, use **chmod**(1) (**ch**ange **mod**e). **chmod** allows you to specify the file permission symbolically or numerically. Symbolically, you can assign permission by saying

```
chmod g=r file1
```

which sets "group" permission for **file1** to "read" only. To specify the mode numerically, the permission must be put into an octal number. This is done by assigning each possible permission a bit value according to the following table:

4	**2**	**1**	**4**	**2**	**1**	**4**	**2**	**1**
read	**write**	**execute**	**read**	**write**	**execute**	**read**	**write**	**execute**
	OWNER			**GROUP**			**OTHER**	

The value of each possible setting is found by taking the numbers above each permission, and adding them up for the owner, group, and other categories. Full permission is 777; no permission at all is 000. The default permission (read/write for owner and group, read for others) is 664.

To set the file permission to "read/write" for the owner only, and no permission for anyone else, the command

```
chmod 600
```

can be used. For obvious reasons, you can only change the protection mode of files which you yourself own.

These same permissions are also used with directories, although they have slightly different meanings. For directories, read permission means "permission to see what files are in the directory." Write permission means, "permission to write or delete an entry in the directory," i.e. create or remove a file within that directory. Execute permission really has nothing to do with execution (how can you execute a directory?), but instead indicates "permission to access a particular file within the directory."

Modes can be assigned to directories just as they are to files. To protect all files within a directory, it is usually easier simply to remove read/write/execute permission from the directory itself. If a user has no permission at all to access a directory, he can't touch the files in it.

You might be wondering what point there is to denying *yourself* permission to read, write or execute a file. In general, protecting a file from yourself is not very useful. However, there are cases when it comes in handy: for instance, if you have important information in a particular file, and want to avoid accidentally overwriting it, you might deny yourself write permission to it. Of course, since you remain the file's owner, you can always re-enable your own write permission later and write to it when you really want to.

The default permission setting is read/write for owner and group, and read only for others. If you want to change this default, you can set the "umask" value in the ".profile" file in your home directory. "Umask" indicates the value of those permissions which are to be turned *off* whenever a file is created.

Normally, this consists of write permission for others; as seen in the table above, this has value 2. The line

```
umask N
```

in your .profile sets the mode to **N**. For example,

```
umask 44
```

means that read/write permission for group and other will always be denied whenever a file is created. Of course, after a file is created, the owner can always change its permission with **chmod**.

1.3.12 Identifying Files

If an unknown file appears in your directory (dropped there by someone else, or created accidentally), you can use the **file** command to try to identify it, as in

```
file grog
```

file makes a good guess at what its contents are (executable program, library, C source ...), although it isn't always accurate.

Conventions exist under VENIX for filename "extensions" of the form ".X" at the end of file names, to give some clue as to the contents of the file. These conventions are enforced by various programs: for example, the C compiler assumes that all files with ".c" names are C source files, while files with ".o" are object files. The following is a list of common extensions and their meaning:

.a	Library archive (created by **ar**(1))
.c	C source file
.h	C '#include' file

.l	Lex source file
.o	Object file
.s	Assembler source file
.y	Yacc source file

Some programs, like the text processor **nroff**, don't care at all about extensions. When using these programs, the user is free to choose whatever extension is convenient and helpful (or not to use any extensions at all).

Finally, there are a few files which appear commonly, created or used automatically by various programs:

.profile	User profile, executed whenever a user logs in, found in the home directory. Commonly contains commands to set the prompt string and do other start-up activities.
a.out	Executable program, as produced by one of the compilers and the linker/loader.
dead.letter	Where mail is placed if the addressee is unknown.
core	Core dump - usually created when a program does an illegal memory reference or contains an illegal instruction. The **adb** debugger may be used to analyze this by experienced users; beginners should simply delete **core** files when they appear, and seek help elsewhere.
mbox	Mail box, used to hold letters you receive and want saved.

1.4 DOCUMENT PREPARATION

VENIX systems are used extensively for document preparation. **nroff** (pronounced ''en-roff'') is a program that produces a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like.

1.4.1 Formatting Packages

The basic idea of **nroff** is that the text to be formatted contains within it ''formatting commands'' that indicate in detail how the formatted text is to look.

nroff is very flexible, but it can be complicated and difficult to use from scratch. For this reason, a ''package'' of canned formatting requests (known as ''macros'') is available to let you specify paragraphs, running titles, footnotes, multi-column output, and so on, with little effort and without having to learn too much about **nroff**. The macro package called −**ms**, for ''manuscript,'' takes a modest effort to learn, but the rewards for using it are so great that it is time well spent. In this section we will provide a hasty look at its main functions.

Formatting requests typically consist of a period and two upper-case letters, such as **.TL**, which is used to introduce a title, or **.PP** to begin a new paragraph. A document is typed so it looks something like this:

```
.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
etc.
```

The lines that begin with a period are the formatting requests. For example, **.PP** calls for starting a new paragraph. The precise meaning of **.PP** depends on what publication you may have specified (with a previous formatting request) the document will appear in. The rules can be changed if you like, but they are changed by changing the interpretation of **.PP**, not by re-typing the document.

To actually produce a document in standard format using **−ms**, use the command

```
nroff -ms files ...
```

The **−ms** argument tells **nroff** to use the manuscript package of formatting requests. For more information, see the *Document Processing Guide* and **nroff**(1) in the *User Reference Manual*.

1.4.2 Supporting Tools

In addition to the basic formatters, there are a lot of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the manual.

neqn lets you integrate mathematics into the text of a document. The program **tbl** provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths. See the *User Reference Manual* for descriptions of these programs.

spell detects possible spelling mistakes in a document. It works by comparing the words in your document to a dictionary, and printing those that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a very good job.

grep looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

```
grep 'ing$' chap*
```

will find all lines that end with the letters **ing** in the files **chap***. (It is almost always a good practice to put single quotes around the pattern you're searching for, in case it contains characters like * or $ that have a special meaning to the shell.) **grep** is often useful for finding out in which of a set of files the misspelled words detected by **spell** are actually located.

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading.)

wc counts the words, lines and characters in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

```
tr A-Z a-z <input >output
```

sort sorts files in a variety of ways; **ptx** makes a permuted index (keyword-in-context listing). **sed** provides many of the editing facilities of **ed**, but can apply them to arbitrarily long inputs. **awk** provides the ability to do both pattern matching and numeric computations, and to conveniently process fields within lines. These programs are for more advanced users, and they are not limited to document preparation. Put them on your list of things to learn about.

Most of these programs are either independently documented in the *Support Tools Guide*, or the *Document Processing Guide* (such as **neqn** and **tbl**), or are sufficiently simple that the description in the *User Reference Manual* is adequate explanation.

1.4.3 Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

Keep the individual files of a document down to modest size, perhaps ten to fifteen thousand characters. Larger files edit more slowly, and of course if you make a dumb mistake it's better to have lost a small file than a big one. Split into files at natural boundaries in the document, for the same reasons that you start each sentence on a new line.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of requests like **.PP**, and then define them appropriately, either by using one of the canned packages (the better way) or by defining your own **nroff** commands. As long as you have entered the text in some systematic way, it can always be cleaned up and re-formatted by a judicious combination of editing commands and request definitions.

1.5 PROGRAMMING

The VENIX system is a productive programming environment. There is already available a rich set of tools and facilities like pipes, I/O redirection and the capabilities of the shell that often make it possible to do a job by pasting together existing programs instead of writing from scratch.

In conjunction with reading this section you should consult the *Programming Guide*.

1.5.1 The Shell

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

```
cat ...     collect the files
| tr ...    put each word on a new line
| tr ...    delete punctuation, etc.
| sort      into dictionary order
| uniq      discard duplicates
| comm      print words in text
            but not in dictionary
```

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, you could laboriously type

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

But you can do the job much more easily. One way is to type

```
ls chap* >temp
```

to get the list of filenames into a file. Then edit this file to make the necessary series of editing commands (using the global commands of **ed**), and write it into **script**. Now the command

```
ed <script
```

will produce the same output as the laborious hand typing. Alternately (and more easily), you can use the fact that the shell will perform loops, repeating a set of commands over and over again for a set of arguments:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable *i* to each file name in turn, then does the command. You can type this command at the terminal, or put it in a file for later execution.

Shell variables can be extremely useful, especially for repeatedly referencing long pathnames. For instance, if you want to access the file **/u2/fred/books/autobiog/chap2**, you could say

```
cat /u2/fred/books/autobiog/chap2
```

or

```
cd /u2/fred/books/autobiog
cat chap2
```

But if you have to reference several files in different directories, this approach is rough on the fingers. With shell variables, you can say

```
X="/u2/fred/books/autobiog/chap2"
Y="/usr/robin/progs/crunch.c"
```

and thereafter use **$X** and **$Y** to access the files, as in

```
cat $X
cc -c $Y
```

1.5.2 Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, with variables, control flow (**if-else**, **while**, **for**, **case**), subroutines, and interrupt handling. Since there are many building-block programs, you can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in "AN INTRODUCTION TO THE SHELL" in this volume.

1.5.3 Programming in C

If you are undertaking anything substantial, C is the only reasonable choice of programming language; everything in the VENIX system is attuned to it. The system itself is written in C, as are most of the programs that run on it. It is also an easy language to use once you get started. C is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the book describe the system interfaces, that is, how to do I/O and similar functions. The document "VENIX PROGRAMMING" in the *Programming Guide* is mandatory reading for all but the most trivial C programming under VENIX.

Here is a simple program, **hello.c**, taken from page 6 of the Kernighan and Ritchie book:

```
main()
{
        printf("hello, world\n");
}
```

If you compile the program with the C compiler **cc**, as in

```
cc hello.c
```

you will get a file called **a.out** created in your directory. Run the program by typing

```
a.out
```

and you will get

```
hello, world
```

on your terminal. You can try some of the things you learned above. Typing

```
a.out >temp
```

will create a file called **temp**, containing the words "hello, world."

```
a.out | wc
```

will get you a count of the lines, words and characters, i.e.

```
1  2   13
```

or one line, two words, and thirteen characters (the newline character at the end is counted).

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines with C compilers. It is automatically called with every **cc**, and described in pages marked **3S** in the *User's Guide*. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library.

Most C programs can be moved to other UNIX or UNIX-derived operating systems, which exist on a wide variety of processors, including at least: the PDP-11, VAX-11, all Intel 16 and 32 bit processors, Zilog Z8000, Motorola 68000,IBM 370 and Series 1, and National 16000. C compilers exist for other machines in addition, and will support most C programs unless they require special features of UNIX (such as pipes). Calls to the standard I/O library will work on all of these machines.

There are a number of supporting programs that go with C. **lint** checks C programs for potential portability problems, and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file) **make** allows you to specify the dependencies among the source files and the processing steps needed to make a new version; it then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger **adb** is useful for digging through the dead bodies of C programs, but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited instrumentation service, so you can find out where programs spend their time and what parts are worth optimizing. Compile the routines with the **−p** option; after the test run, use **prof** to print an execution profile. The command **time** will give you the gross run-time statistics of a program, but they are not super accurate or reproducible.

1.5.4 Other Languages

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the **yacc** compiler-compiler, which helps you develop a compiler quickly. The **lex** lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself, or as a front end to recognize inputs for a **yacc**-based program. Both **yacc** and **lex** require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later on.

1.6 VENIX READING LIST

Many of the documents listed below can be found either in this volume, the *Programming Guide*, *Support Tools Guide*, or the *Document Processing Guide*.

General:

User Reference Manual Lists VENIX commands.

Programmer Reference Manual Lists system routines and interfaces, file formats, and some of the maintenance procedures.

Special*The Bell System Technical Journal*(BSTJ) Issue on UNIX, July/August, 1978, contains many papers describing UNIX, and some retrospective material.

The 2nd International Conference on Software Engineering (October, 1976) contains several papers describing the use of the Programmer's Workbench (PWB) version of UNIX.

H. McGilton and R. Morgan. *Introducing the UNIX System*. McGraw-Hill, 1983. A primer for the UNIX novice and an easy-to-use reference guide for experienced users.

Document Preparation:

The *FinalWord for VENIX* gives a simple but complete explanation of how to use The FinalWord, a full screen editor and formatter.

"NROFF USER'S MANUAL" **nroff** is the basic formatter used with −**ms**, **neqn** and **tbl**. The reference manual is indispensable if you are going to write your own macro sets, but start with these papers which are in the *Document Processing Guide*.

"AN NROFF TUTORIAL" It is an attempt to unravel the intricacies of **nroff**.

"USING THE -MS MACROS" Describes the −**ms** macro package, which isolates the novice from the vagaries of **nroff** and takes care of most formatting situations. (See the *Document Processing Guide.*)

"MATHEMATICS TYPESETTING PROGRAM" (See the *Document Processing Guide.*)

"TABLE FORMATTING PROGRAM" (See the *Document Processing Guide.*)

Programming:

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978. Contains a tutorial introduction, complete discussions of all language features, and the reference manual.

"VENIX PROGRAMMING" Describes how to interface with the system from C programs: I/O calls, signals, processes. (See the *Programming Guide*.)

"AN INTRODUCTION TO THE SHELL" An introduction and reference manual for the shell. Mandatory reading if you intend to make effective use of the programming power of this shell. (See this volume.)

"YET ANOTHER COMPILER-COMPILER (yacc)" (See the *Support Tools Guide*.)

"A LEXICAL ANALYZER GENERATOR (lex)" (See the *Support Tools Guide*.)

"A C PROGRAM CHECKER - lint" (See the *Programming Guide*.)

"A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make)" (See the *Support Tools Guide*.)

"A TUTORIAL INTRODUCTION TO ADB" An introduction to a powerful but complex debugging tool. (See the *Programming Guide*.)

"FORTRAN 77" A full Fortran 77 for the VENIX system. (See the *Programming Guide*.)

APPENDIX A

TERMINAL CONTROL CODES

CTRL-C	Stops a program.
CTRL-D	Indicates end of terminal input to a program and a "log out" to the shell.
CTRL-E	Erases all previous unread lines, not just the current input line.
CTRL-Q	Restarts output to terminal.
CTRL-R	Re-echoes everything typed that has not yet been read by a program.
CTRL-S	Stops output to terminal.
CTRL-U	Erases all characters typed to that point on the current input line.
CTRL-Z	Stops a program and causes a "core dump" for debugging.

NOTE: These characters do not have the same meaning in a screen editor.

Contents

2.1 GETTING STARTED 2-1

2.2 MOVING AROUND IN THE FILE 2-4

2.3 MAKING SIMPLE CHANGES 2-9

2.4 MOVING ABOUT;

REARRANGING AND DUPLICATING TEXT 2-13

2.5 HIGH LEVEL COMMANDS 2-17

2.6 SPECIAL TOPICS 2-19

2.7 NITTY-GRITTY DETAILS 2-27

Chapter 2

AN INTRODUCTION TO DISPLAY EDITING WITH VI

2.1 GETTING STARTED

This document provides an introduction to the screen editor **vi** (pronounced *vee-eye*). Sections 2.1 through 2.5 of this document describe the basics of using **vi**. Some topics of special interest are presented in section 2.6, and the nitty-gritty details about how the editor functions are saved for section 2.8.

2.1.1 Specifying Terminal Type

Before you can start **vi** you must tell the system what kind of terminal you are using. A list of the more common terminal types and their codes are given in Appendix A. If your terminal does not appear in this listing, ask your system administrator for your terminal code. If your terminal does not have a code, one can be assigned and a description for the terminal can be created.

To specify your terminal type use the following command:

```
% setenv TERM terminal
```

where *terminal* is the terminal code. So, for example, to set the terminal type for a DEC Professional you would enter:

```
% setenv TERM pro
```

The **setenv** command works with the C shell **csh**. If you are using the Bourne shell then you should give the commands:

```
$ TERM=pro
$ export TERM
```

If you want to have your terminal type set up automatically when you log in, these statements can be placed in your **.login** file (for csh) or **.profile** (for Bourne shell). Terminal types can also be set inside **vi** with the **set term** command.

2.1.2 Editing a File

After telling the system which kind of terminal you have, make a scratch copy of a file you are familiar with, and run **vi** on this file. Give the command

```
% vi scratchfile
```

The screen should clear and the text of your file should appear on the screen.

If you gave the system an incorrect terminal type code then you may have a mess on your screen now. This happens when the editor sends control codes for one kind of terminal to another kind of terminal. In this case hit the keys **:q** (colon and the **q** key) and then hit the CR key. This should get you back to the command level interpreter. Figure out what you did wrong and try again.

Another thing which can go wrong is that you typed the wrong file name and the editor just printed an error diagnostic. In this case you should follow the above procedure for getting out of the editor, and try again this time spelling the file name correctly.

If the editor doesn't seem to respond to the commands which you type here, try sending an interrupt to it by typing CTRL-C, and then hitting the **:q** command again followed by the CR key.

2.1.3 The Editor's Copy: The Buffer

The editor does not directly modify the file you are editing. Rather, the editor makes a copy of this file, in a place called the **buffer,** and remembers the file's name. You do not affect the contents of the file unless and until you write into the original file the changes you have made.

2.1.4 Arrow Keys

The arrow keys, located to the right of the keyboard, are used to move the cursor up, down, left, and right. In addition to the arrow keys, you may also use: **h** to move the cursor to the left, **j** to move the cursor down, **k** to move the cursor up, and **l** to move the cursor to the right.

2.1.5 Special Keys: ESC, CR and DEL

These special keys are very important, so be sure to find them right now. Look on your keyboard for the key labeled ESC or ALT.

Try hitting this key a few times. The editor will ring the bell or flash the screen to indicate that it is in a quiescent state. Partially formed commands are canceled by ESC, and when you insert text in the file you end the text insertion with ESC. This key is a fairly harmless one to hit if you don't know what is happening.

The CR key is important because it is used to terminate certain commands. It is on the right side of the keyboard, and is the same command used at the end of each shell command.

Another very useful command is CTRL-C (which is typed by holding down the CTRL key down and pressing C at the same time).

CTRL-C generates an interrupt, telling the editor to stop what it is doing. It is a forceful way of making the editor listen to you, or to return it to the quiescent state if you don't know or don't like what is going on. Try hitting the '/' key on your terminal. (This key is used when you want to specify a search string, and is discussed more in section 2.2.) The cursor should now be positioned on the bottom line of the terminal after a '/' printed as a prompt. You can get the cursor back to its previous position in the text by typing CTRL-C; try this now.* From now on we will simply refer to CTRL-C as "sending an interrupt."

* Backspacing over the '/' will also cancel the search.

The editor often echoes, on the last line of the terminal, your commands. If the cursor is resting on the first position of this last line, then the editor is performing a computation, such as computing a new position in the file after a search or running a command to reformat part of the buffer. When this is happening you can stop the editor by sending an interrupt.

2.1.6 Getting Out of the Editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **:wq** to the editor. This will write the contents of the editor's buffer back into the file you are editing, if you made any changes, and then end the editing session. You can also end an edit session by giving the command **:q!**CR; This is a dangerous, but occasionally essential, command that ends the edit session and *discards all your changes.*** Be very careful not to give this command when you really want to incorporate the changes you have made!

2.2 MOVING AROUND IN THE FILE

2.2.1 Scrolling and Paging

The editor has a number of commands for moving around in the file. The most useful of these is generated by hitting the CTRL and D keys at the same time, CTRL-D. This two-character notation will be used throughout this document for referring to a control character.

As you know if you tried hitting CTRL-D, this command scrolls down in the file. The D stands for down. Many editor commands are mnemonic making them fairly easy to remember. For instance, the command to scroll up is CTRL-U. For dumb terminals that can't scroll up, hitting CTRL-U clears the screen and refreshes it with a line which is farther back in the file.

** All commands which read from the last display line can also be terminated with an ESC as well as a CR.

There are other ways to move around in the file: the keys CTRL-F and CTRL-B move forward and backward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these rather than CTRL-D and CTRL-U if you wish.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting CTRL-F to move forward a page will leave you only a little of the previous text to look at. Scrolling, on the other hand, leaves more context and happens more smoothly. You can continue to read the text as it scrolls.

2.2.2 Searching, Goto, and Previous Context

By giving the editor a string of characters or words to search for, you can locate a specific position in the file. Type the character /, then a string of characters, and finally a CR. The editor will position the cursor at the next occurrence of this string. Try hitting **n** to go on to the next occurrence of this string. The character **?** will search in reverse from where you give the command, but otherwise it is like /.*

If the search string you give the editor is not present in the file, the editor will print a message on the last line of the screen, and the cursor will return to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with the caret symbol (^). To match only at the end of a line, end the

* These searches will normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction of your search. You can disable this wraparound in scans by giving the command **:se nowrapscan** CR, or more briefly **:se nows** CR.

search string with a **$**. Thus **/^search** CR will search for the word 'search' at the beginning of a line, and **/last$** CR searches for the word 'last' at the end of a line.**

The command **G**, when preceded by a number will position the cursor at that line in the file. Thus **1G** will move the cursor to the first line of the file. If you do not give **G** a count, then it moves to the end of the file.

If you are near the end of the file, and the last line of text is not at the bottom of the screen, the editor will place only the character '~' on each remaining line. This indicates that the last line in the file is on the screen, that is, the '~' lines are past the end of the file.

You can find out the state of the file you are editing by typing a CTRL-G. The editor will show you: the file name, the number of the current line, the number of lines in the buffer, and the percentage of how far through the buffer you are. Try doing this now, and remember the number of the line you are on. Give a **G** command to get to the end and then another **G** command to get back where you were.

You can also get back to a previous position by using the command `` (two back quotes). This is often more convenient than **G** because it requires no advance preparation. Try giving a **G** or a search with / or **?** and then a `` to get back to where you were. If you accidentally hit **n** or any command which moves you far away from a context of interest, you can quickly get back by hitting ``.

** Actually, the search string you give here can be a **regular expression** in the sense of the editors **ex**(1) and **ed**(1). If you don't wish to learn about this yet, you can disable this more general facility with **:se nomagic** CR. By putting this command in **EXINIT** in your environment, it will always be in effect (more about **EXINIT** later).

2.2.3 Moving Around on the Screen

Experiment with moving the cursor around on the screen. First try the arrow keys and then use the optional commands **h**, **j**, **k**, and **l** as described in section 1.4. Experienced users of **vi** prefer these keys to arrow keys because they are usually right underneath their fingers.

Hit the + key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-white position on the line. The – key is like + but goes the other way.

These are very common keys for moving up and down lines in the file. If you move beyond the text on the screen while using these keys, then the screen will scroll down (or up if possible) to bring a line at a time into view. The CR key has the same effect as the + key.

vi also has commands to take you to the top, middle, and bottom of the screen. **H** will take you to the top (home) line on the screen. Preceding it with a number will take you to that line on the screen. For example, **3H** will position the cursor on the third line on the screen. The command **M** will take you to the middle line on the screen, and **L** will take you to the last line on the screen. **L** also takes counts, thus **5L** will take you to the fifth line from the bottom.

2.2.4 Moving Within a Line

Pick a word on the screen, but not the first word on a line. Move the cursor using CR and – to the line where the word is. Now use the **w** key to advance the cursor word by word on the line or hit the **b** key to back up by words. You may also want to use the **e** key which advances the cursor to the end of the current word rather than the beginning of the next. Also try SPACE (the space bar) which moves right one character and the BACKSPACE key (or CTRL-H) which moves left one character. The key **h** works as CTRL-H does and is useful if you don't have a BACKSPACE key. (Also, as noted just above, **l** will move to the right.)

The **w** and **b** keys stop at each group of punctuation. To go backward and forward by word without stopping at punctuation use **W** and **B** rather than the lower case equivalents. Try these on a few lines with punctuation to see how they differ from the lower case **w** and **b**.

The word keys wrap around the end of a line. Try moving to a word on the next line by repeatedly hitting **w**.

2.2.5 Summary

The following table summarizes the commands that have been introduced in this section:

SPACE	advance the cursor one position
CTRL-B	backwards to previous page
CTRL-D	scrolls down in the file
CTRL-G	tell what is going on
CTRL-H	backspace the cursor
CTRL-N	next line, same column
CTRL-P	previous line, same column
CTRL-U	scrolls up in the file
+	next line, at the beginning
–	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of / or **?** pattern
w	word after this word

2.2.6 View

To use the editor to look at a file, rather than to make changes, invoke **view** instead of **vi**. This will set the **readonly** option and will prevent you from accidently overwriting the file.

2.3 MAKING SIMPLE CHANGES

2.3.1 Inserting

One of the most useful commands is the **i** (insert) command. After you type **i**, everything you type is inserted into the file until you hit ESC Try this now; position your cursor to some word in the file and insert text before this word. On a dumb terminal it will seem, for a minute, that some of the characters on your line have been overwritten, but they will reappear when you hit ESC .P Now try finding a word which can, but does not now, end in an 's'. Move your cursor to this word and type **e** (move to end of word), then **a** for append and then 's ESC' to terminate the textual insert. This easy sequence of commands can be used to pluralize a word.

Try inserting and appending a few times to make sure you understand how this works. Remember with **i** you place text to the left of the cursor, with **a** to the right.

Often when you are editing, you will need to add new lines of text before or after some specific line in the file. Find a line where this makes sense and give the command **o** to create a new line after the line you are on, or the command **O** to create a new line before the line you are on. After you create a new line in this way, the text you type is inserted on the new line until you type an ESC.

Many related editor commands are invoked by the same letter key and differ only in that one is a lower case key and the other is an upper case key. In these cases, the upper case key often differs from the lower case key in its sense of direction, with the upper case key working backward and/or up, while the lower case key moves forward and/or down.

When typing in more than one line of text, hit a CR at the middle of your input. A new line will be created for text, and you can continue to type. On a slow or dumb terminal, the editor may choose to let you type over the existing screen lines. This avoids the lengthy delay that would occur if the editor attempted to keep the screen always up to date. The screen will be fixed up and the missing lines will reappear when you hit ESC.

While you are inserting new text, you can use some of the characters you normally use at the system command level. To backspace over the last character that you typed you can use CTRL-H or the BACKSPACE key. To erase the input you have typed on the current line you can use CTRL-U for killing input lines. The character CTRL-W will erase a whole word and leave your cursor at the space after the previous word; it is useful for quickly backing up in an insert.

Notice that when you backspace during an insertion the characters you backspace over are not erased; the cursor moves backwards, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case the characters disappear when when you hit ESC; if you want to get rid of them immediately, hit ESC and then **a** again.

Notice also that you can only erase characters that you have just inserted, and that you can't backspace around the end of a line. If you need to back up to the previous line to make a correction, just hit ESC and move the cursor back to the previous line. After making the correction you can return to where you were and use the insert or append command again.

2.3.2 Making Small Corrections

Small corrections in existing text are quite easy to make. If there is a single character that you'd like to get rid of just position the cursor on the wrong character and hit the **x** key; this deletes the character from the file.

If there is a character that is incorrect, you can replace it with the correct character by the command **r***c*, where *c* is the correct character. Finally, if the incorrect character should be replaced by a string of characters, give the **s** command followed by the replacement string and end with ESC. If there are a small number of characters that are wrong you can precede **s** with a count of the number of characters to be replaced. Counts with **x** are also useful to specify the number of characters to be deleted.

2.3.3 More Corrections: Operators

To make changes at a higher level, you should use the **d** key as a delete operator. Try the command **dw** to delete a word. Now type **.** a few times. Notice that this repeats the effect of **dw**. The command **.** repeats the last command which made a change. The command **db** deletes a word backwards, namely the preceding word. For deleting single characters use **d**SPACE. This is equivalent to the **x** command.

Another very useful operator is **c** for change. To change the text of a single word, use **cw** followed by the replacement text and ESC. Find a word which you can change to another, and try this now. You'll see that the end of the text to be changed was marked with the character '**$**'.

2.3.4 Operating on Lines

Variations of the operators discussed above can be used on entire lines of text. For example, to delete a line of text type the **d** operator twice, **dd**. If you are on a dumb terminal, the editor may erase the line on the screen, replacing it with only an @. This line does not correspond to any line in your file, but only acts as a place holder. By repeating the **c** operator twice **cc**, you can change a whole line, erasing its previous content and replacing it with text you type up to an ESC. †

You can delete or change more than one line by preceding the **dd** or **cc** with a count, i.e. **5dd** deletes 5 lines. You can also give a command like **dL** to delete all the lines up to and including the last line on the screen, or **d3L** to delete through the third from the bottom line. Try some commands like this now.* Notice that the editor lets you know when you change a large number of lines so that you can see the extent of the change. The editor will also always tell you when a change affects text which you cannot see.

† The command **S** is a synonym for **cc**. **S** is a substitute on lines, while s is a substitute on characters.

* One subtle point is that using the / search after a **d** will normally delete characters from the current position to the point of the match. To delete whole lines including the two points, give the pattern as **/pat/+0**, a line address.

2.3.5 Undoing

Now suppose that the last change which you made was incorrect; you could use the insert, delete and append commands to put the correct material back. However, since we often regret a change or make a change incorrectly, the editor provides a **u** (undo) command to reverse the last change. Try this a few times, and give it twice in a row to notice that a **u** also undoes a **u**.

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have your original line back. The **U** command restores the current line to the state before you started changing it. You can recover text which you delete, even if **u** undo will not bring it back. (See the section below on recovering lost text.)

2.3.6 Summary

SPACE	advance the cursor one position
CTRL-H	backspace the cursor
CTRL-W	erase a word during an insert
DEL	erase a character during an insert
CTRL-U	kill the insert on this line
.	repeat the changing command
O	open and inputs new lines, above the current
U	undo the changes you made to the current line
a	append text after the cursor
c	change the object you specify to the following text
d	delete the object you specify
i	insert text before the cursor
o	open and inputs new lines, below the current
u	undo the last change

2.4 MOVING ABOUT; REARRANGING AND DUPLICATING TEXT

2.4.1 Low Level Character Motions

Move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis, comma or period. Try the command **fx** where **x** is this character. This command finds the next **x** character to the right of the cursor in the current line. Then hit a ; which finds the next instance of the same character. By using the **f** command and then a sequence of ;'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACEs. There is also a **F** command, which is like **f**, but searches backward. The ; command repeats **F** also.

When you are operating on the text in a line it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try for**df***x* **x** now and notice that the **x** character is deleted. Undo this with **u** and then try the**dt***x*; **t** here stands for "to", i.e. delete up "to" the next **x**, but not the **x**. The command **T** is the reverse of and**t**, **x**.

When working with the text of a single line, an ^ moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Thus **$a** will append new text at the end of the current line.

Your file may have tab (^I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every 8 columns.* When the cursor is at a tab, it sits on the last of the several spaces which represent the width of that tab. Experiment with moving the cursor back and forth over tabs so you understand how this works.

* This is set by a command of the form **:set tabstop**=*x* CR, to set tabstops every **x** columns. This has an effect only on the screen representation within the editor.

On rare occasions, your file may have nonprinting characters in it. These characters are displayed with a two character code, the first character of which is a caret (^). On the screen, non-printing characters resemble a '^' character adjacent to another, but spacing or backspacing over the character will reveal that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes discards control characters that you are attempting to insert in your file. This depends on the character and the setting of the **beautify** option. You can get a control character in the file by beginning an insert and then typing a CTRL-V before the control character. The CTRL-V quotes the following character, causing it to be inserted directly into the file.

2.4.2 Higher Level Text Objects

In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections. The operations **(** and **)** move to the beginning of the previous and next sentences respectively. Thus the command **d)** will delete the rest of the current sentence; likewise **d(** will delete the previous sentence if you are at the beginning of the current sentence, or the current sentence up to where you are if you are not at the beginning.

A sentence is defined to end with a '**.**', '**!**' or '**?**' which is followed by either the end of a line, or by two spaces. Any number of closing '**)**', '**]**', '**"**' and '**'**' characters may appear after the '**.**', '**!**' or '**?**' before the spaces or end of line.

The operations { and } move over paragraphs and the operations **[[** and **]]** move over sections.†

† The **[[** and **]]** operations require the operation character to be doubled because they can move the cursor far from where it currently is. While it is easy to get back with the command ``, these commands would still be frustrating if they were easy to hit accidentally. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string valued option **paragraphs**. The default setting for this option defines the paragraph macros of the **−ms** and **−mm** macro packages, used with the nroff text formatter i.e., the '**.BP**', '**.LP**', '**.PP**' and '**.QP**', '**.P**' and '**.LI**' macros.‡ Each paragraph boundary is also a sentence boundary.

Sections in the editor begin after each macro in the **sections** option, normally '**.NH**', '**.SH**', '**.H**' and '**.HU**', and each line with a formfeed **^L** in the first column. Section boundaries are always line and paragraph boundaries also.

Experiment with the sentence and paragraph commands until you are sure how they work. If you have a large document, look through it using the section commands. The section commands interpret a preceding count as a different window size in which to redraw the screen at the new location, and this window size is the base size for newly drawn windows until another size is specified. This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command a small count to then see each successive section heading in a small window.

2.4.3 Rearranging and Duplicating Text

The editor has a single unnamed buffer where the last deleted or changed text is saved. It also has a set of named buffers **a−z** that you can use for saving copies of text and for moving text around in a file or between files.

The operator **y** "yanks" a copy of the object which follows into the unnamed buffer. If preceded by a buffer name, "*x***y**, where *x* here is replaced by a letter **a−z**, it places the text in the named buffer. The text can then be put back in the file with the commands **p** or **P**; **p** puts the text after or below the cursor, while **P** puts the text before or above the cursor.

‡ You can easily change or extend this set of macros by assigning a different string to the **paragraphs** option in your EXINIT. The '**.bp**' directive is also considered to start a paragraph.

If the text that you yank forms part of a line, or is a sentence which partially spans more than one line, then the text will be placed after the cursor (or before if you use **P**) when you put it back. If the yanked text forms whole lines they will be put back as whole lines, without changing the current line. In this case, the put command acts much like an **o** or **O** command.

The command **YP** makes a copy of the current line and leaves the cursor on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also make a copy of the current line, and place it after the current line. You can give **Y** a count of lines to yank (such as **3YP**), and thus duplicate several lines.

To move text within the buffer, delete it from one place and put it back in another. The delete operation can be preceded by the name of a buffer where the text is to be stored. For example, **"a5dd** deletes 5 lines into the named buffer **a**. You can move the cursor to the eventual resting place of the these lines and do a **"ap** or **"aP** to put them back. In fact, you can switch and edit another file before you put the lines back, by giving the command **:e** *name* CR where *name* is the name of the other file you want to edit. You will have to write back or discard the contents of the current editor buffer, if you have made changes, before the editor will let you switch to the other file. An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files, so to move text from one file to another you should use an unnamed buffer.

2.4.4 Summary

^	first non-white on line
$	end of line
)	forward sentence
}	forward paragraph
]]	forward section
(	backward sentence
{	backward paragraph
[[	backward section
f*x*	find *x* forward in line
p	put text back, after cursor or below current line

y	yank operator, for copies and moves
t*x*	up to *x* forward, for operators
F*x*	**f** backward in line
P	put text back, before cursor or above current line
T*x*	**t** backward in line

2.5 HIGH LEVEL COMMANDS

2.5.1 Writing, Quitting, Editing New Files

So far we have seen how to enter **vi** and write out a file using either **:wq** or .**:w**CR The first writes changes and exits from the editor; the second writes and stays in the editor.

If you have changed the editor's copy of the file but do not wish to save your changes, then you can give the command **:q!**CR to quit the editor without writing the changes. You can also reedit the same file (start over) by giving the command .**:e!**CR Note that these commands should be used only rarely, and with caution, as it is not possible to recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving the editor by giving the command **:e name** CR. The editor will tell you if you have not written out your current file before you try to do this, and it will delay editing the other file. You can then give the command **:w**CR to save your work before giving the **:e name** CR command again. Or you can carefully give the command **:e! name** CR, which edits the other file and discards the changes you have made to the current file. To have the editor automatically save changes, include **set autowrite** in your EXINIT, and use **:n** instead of **:e**.

2.5.2 Escaping to a Shell

You can get to a shell to execute a single command by giving the **vi** command .:!cmdCR The system will run the single command **cmd** and when the command finishes, the editor will ask you to hit a CR to continue. When you have finished looking at the output on the screen, hit CR to get back to editing. You can also give another : command when it asks you for a CR. If you wish to execute more than one command in the shell, then give the command ;**:sh**CR this will give you a new shell. When you finish with the shell type a CTRL-D and the editor will clear the screen and continue editing.

2.5.3 Marking and Returning

The command `` returns the cursor to the place it was before being moved by a command such as /, **?** or **G**. You can also mark lines in the file with single letter name tags and return to these marks later by invoking the names. Try marking the current line with the command **m**_x_, where you pick some letter for _x_, say '**a**'. Then move the cursor to a different line (anywhere you like) and hit `**a**. The cursor will return to the place you marked. Marks last only until you edit another file.

When using operators such as **d** and referring to marked lines, it is often desirable to delete whole lines rather than to the exact position in the line marked by **m**. In this case you can use the form '**x** rather than `**x**. Used without an operator, '**x** will move to the first non-white character of the marked line; similarly '' moves to the first non-white character of the line containing the previous context mark ``.

2.5.4 Adjusting the Screen

If the screen image is messed up because of a transmission error to your terminal, or because some program other than the editor wrote output to your terminal, you can hit a CTRL-L, the ASCII form-feed character, to refresh the screen.

On a dumb terminal, if there are lines of @ symbols on the screen as a result of line deletions, you may delete these lines by typing CTRL-R.

Finally, if you wish to place a certain line of text at the top, middle or bottom of the screen, you can position the cursor to that text line, and then give a **z** command. You should follow the **z** command with a CR if you want the line to appear at the top of the window, a **.** if you want it at the center, or a **–** if you want it at the bottom.

2.6 SPECIAL TOPICS

2.6.1 Editing on Slow Terminals

When you are on a slow terminal, it is important to limit the amount of output which is generated to your screen so that you will not suffer long delays, waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how the editor replaces a deleted line with an @ on dumb terminals.

The use of the slow terminal insertion mode is controlled by the **slowopen** option. You can force the editor to use this mode even on faster terminals by giving the command **.:se slow**CR If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by **.:se noslow**CR

The editor can simulate an intelligent terminal on a dumb one. Try giving the command **.:se redraw**CR This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command **.:se noredraw**CR

Editing at low speed can be made more pleasant by starting to edit in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window which is redrawn after the screen is cleared by giving window sizes as argument to the commands which cause large screen motions:

```
: / ? [[ ]] ` '
```

If you are searching in a file for a particular instance of a common string, you can precede the first search command by a small number, say 3, and the editor will draw three line windows around each instance of the string.

You can easily expand or contract the window, placing the current line as you choose, by giving a number on a **z** command, after the **z** and before the following CR, **.** or **−**. Thus the command **z5.** redraws the screen with the current line in the center of a five line window.†

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by hitting CTRL-C. If you do this you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by hitting a CTRL-L; or move or search again, ignoring the current state of the display.

2.6.2 Options, Set, and Editor Startup Files

The editor has a set of options, some of which have been mentioned above. The most useful options are given in the following table.

Name	**Default**	**Description**
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before **:n**, **:ta**, ^, **!**
ignorecase	noic	Ignore case in searching
list	nolist	Tabs print as**^I**; end of lines as **$**
magic	nomagic	Characters **.** **[** and * are special in scans
number	nonu	Lines are displayed with line numbers
paragraphs	para = IPLPPPQPbpP LI	Macro names which start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
sections	sect = NHSHH HU	Macro names which start new sections
shiftwidth	sw = 8	Shift distance for <, > and input **^D** and **^T**
showmatch	nosm	Show matching (or { as) or } is typed
slowopen	slow	Postpone display updates during inserts
term	dumb	The kind of terminal you are using.

† Note that the command **5z.** has an entirely different effect, placing line 5 in the center of a new window.

There are three kinds of options: numeric, string and toggle. You can set numeric and string options with the statement:

set *opt*=*val*

and toggle options can be set or unset respectively by these statements:

set *opt*
set no*opt*

Option statements can be placed in EXINIT in your environment, or given while you are running **vi** by preceding them with a **:** and following them with a CR.

You can get a list of all options which you have changed by the command ,**:set**CR or the value of a single option by the command **:set ?**CR *opt*. A list of all possible options and their values is generated by .**:set all**CR Set can be abbreviated **se**. Multiple options can be placed on one line, for example:

:se ai aw nuCR**.**

Options set by the **set** command only last while you stay in the editor. To have certain options set whenever you use the editor, create a list of **ex** commands† which are to be run every time you start up **ex**, or **vi**. A typical list includes a **set** command. Since it is advisable to get these commands on one line, they can be separated with the | character, for example:

set | ai| aw| terse

which sets the options **autoindent**, **autowrite**, and **terse**. This string should be placed in the variable EXINIT in your environment. If you use **csh**, put this line in the file **.login** in your home directory:

setenv EXINIT 'set ai aw terse'

† All commands that start with **:** are **ex** commands.

If you use the Bourne shell, put these lines in the file **.profile** in your home directory:

```
EXINIT='set ai aw terse'
export EXINIT
```

Of course, the particulars of the line depend on which options you want to set.

2.6.3 Recovering Lost Lines

If you delete a number of lines and then regret that they were deleted, despair not! The editor saves the last 9 deleted blocks of text in a set of numbered registers 1−9. You can get the *n*'th previously deleted text back in your file by the command **"*n*p**. The **"** here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this and then **.** (period) to repeat the put command. In general the **.** command will repeat the last change you made. As a special case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. So a sequence of the form

```
"1pu.u.u.
```

will, if repeated long enough, show you all the deleted text which has been saved for you. You can omit the **u** commands here to gather all this text in the buffer, or stop after any **.** command to keep just the then recovered text. The command **P** can also be used rather than **p** to put the recovered text before rather than after the cursor.

2.6.4 Recovering Lost Files

If the system crashes, you can recover to within a few changes the work you were doing. When you login, you will normally receive mail with the name of the file which has been saved for you. Change to the directory where you were when the system crashed and give the command:

```
% vi −r name
```

replacing *name* with the name of the file you were editing. This will recover most of the work you did before the crash.†

You can get a listing of the files which are saved for you by giving the command:

```
vi -r
```

If more than one version of a particular file is saved, the editor gives you the newest one each time you run the recover. You can get an old saved copy back by first recovering the new copies. The invocation "**vi -r**" will not always list all saved files, but they can be recovered even if they are not listed.

2.6.5 Continuous Text Input

When you are typing in large amounts of text, it is convenient to have lines broken automatically near the right margin. You can cause this to happen by giving the command **.:se wrapmargin = 10**CR This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.

If the editor breaks an input line where you do not want a break, you can rejoin the lines with **J**. **J** can also take a count of the number of lines to be joined, as in **3J** to join 3 lines. The editor supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. You can kill the white space with **x** if you don't want it.

2.6.6 Features for Editing Programs

The editor has a number of commands for editing programs. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure in the body of the program. The editor has an **autoindent** facility for helping you generate correctly indented programs.

† In rare cases, some lines of the file may be lost. The editor will give you the numbers of these lines and their text will be replaced by the string '**LOST**'. These lines will almost always be among the last ones you changed.

To use this facility give the command **:se ai**CR. Now try opening a new line with **o** and type some characters on the line after a few tabs. If you start another line, notice that the editor supplies space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use CTRL-D to backtab over the supplied indentation.

Each time you type CTRL-D you back up one position, normally to an 8 column boundary. This amount is settable; the editor has an option called **shiftwidth** which you can use to change this value. Try giving the command **:se sw=4**CR and then experimenting with **autoindent** again.

For shifting lines left or right in the program, there are operators < and >. These shift the lines right or left by one **shiftwidth**. Try << and >> which shift one line left or right, and **<L** and **>L** which shift the rest of the display left or right.

In a complicated expression where you wish to see how the parentheses match, put the cursor at a left or right parenthesis and hit **%**. This will show you the matching parenthesis. This works also for braces { and }, and brackets **[** and **]**.

If you are editing C programs, you can use the **[[** and **]]** keys to advance or retreat to a line starting with a {, i.e. a function declaration at a time. When **]]** is used with an operator it stops after a line which starts with }; this is sometimes useful with **y]]**.

2.6.7 Filtering Portions of the Buffer

System commands can run over portions of the buffer by using the operator **!**. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty-printer. As an exercise, type a list of random words, one per line, ending with a blank line. Back up to the beginning of the list, and give the command **.!}sort**CR This says to sort the next paragraph of material, and the blank line ends a paragraph.

2.6.8 Macros

vi has a parameterless macro facility, which lets you set it up so that when you hit a single keystroke, the editor will act as though you had hit some longer sequence of keys. You can set this up if you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two flavors of macros:

a. Ones where you put the macro body in a buffer register, say *x*. You can then type **@x** to invoke the macro. The @ may be followed by another @ to repeat the last macro.

b. You can use the **map** command from **vi** (typically in your **EXINIT**) with a command of the form:

> **:map** *lhs rhs* **CR**

mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either 1 character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and **vi** will wait for you to finish it before it echoes anything). The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs* you should escape them with a CTRL-V. (It may be necessary to double the CTRL-V if the map command is given inside **vi**, rather than in *ex*.) Spaces and tabs inside the *rhs* need not be escaped.

Thus to make the **q** key write and exit the editor, you can give the command

> **:map q :wqCTRL-V CTRL-V CR CR**

which means that whenever you type **q**, it will be as though you had typed the four characters **:wq**CR. A CTRL-V is needed because without it the CR would end the **:** command, rather than becoming part of the *map* definition. There are two CTRL-V's because from within **vi**, two CTRL-V's must be typed to get one. The first CR is part of the *rhs*, the second terminates the **:** command.

Macros can be deleted with

```
:unmap lhs
```

If the *lhs* of a macro is "**#0**" through "**#9**", this maps the particular function key instead of the 2 character "**#**" sequence. So that terminals without function keys can access such definitions, the form "**#x**" will mean function key *x* on all terminals (and need not be typed within one second.) The character "**#**" can be changed by using a macro in the usual way:

```
:map CTRL-V CTRL-V CTRL-I #
```

to use tab, for example. (This won't affect the *map* command, which still uses **#**, but just the invocation from visual mode.)

The undo command reverses an entire macro call as a unit, if it made any changes.

Placing a '**!**' after the word **map** causes the mapping to apply to input mode, rather than command mode. Thus, to arrange for **^T** to be the same as 4 spaces in input mode, you can type:

```
:map CTRL-T CTRL-V/'/'/'/'
```

where /' is a blank. The CTRL-V is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

2.6.9 WORD ABBREVIATIONS

A feature similar to macros in input mode is word abbreviation. This allows you to type a short word and have it expanded into a longer word or words. The commands are **:abbreviate** and **:unabbreviate** (**:ab** and **:una**) and have the same syntax as **:map**. For example:

```
:ab eecs Electrical Engineering and Computer Sciences
```

causes the word '**eecs**' to always be changed into the phrase '**Electrical Engineering and Computer Sciences**'. Word abbreviation is different from macros in that only whole words are affected. If '**eecs**' were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke, as it should be with a macro.

2.6.10 Abbreviations

The editor has a number of short commands which abbreviate longer commands which we have introduced here. They often save a bit of typing and you can learn them as convenient. See **vi**(1) in the *User Reference Manual* for abbreviations.

2.7 NITTY-GRITTY DETAILS

2.7.1 Line Representation in the Display

The editor folds long logical lines onto many physical lines in the display. Commands which advance lines advance logical lines and will skip over all the segments of a line in one motion. The command | moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try **80**| on a line which is more than 80 columns long.†

The editor only puts full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty, placing only an @ on the line as a place holder. When you delete lines on a dumb terminal, the editor will often just clear the lines to @ to save time (rather than rewriting the rest of the screen.) You can always maximize the information on the screen by giving the CTRL-R command.

If you wish, the editor will place line numbers before each line on the display. Give the command **:se nu**CR to enable this, and the command **:se nonu**CR to turn it off. You can have tabs represented as **^I** and the ends of lines indicated with '**$**' by giving the command ;**:se list**CR **:se nolist**CR turns this off.

† Uses **J** to join together short lines.

Finally, lines consisting of only the character '~' are displayed when the last line in the file is in the middle of the screen. These represent physical lines which are past the logical end of a file.

2.7.2 Counts

Most **vi** commands will use a preceding count to affect their behavior in some way. The following table gives the common ways in which the counts are used:

new window size	**: / ? [[ ]] ` '**
scroll amount	**^D ^U**
line/column number	**z G |**
repeat effect	most of the rest

The editor maintains a notion of the current default window size. On microcomputer console terminals and terminals which run at speeds greater than 1200 baud the editor uses the full terminal screen. On terminals which are slower than 1200 baud (most dialup lines are in this group) the editor uses 8 lines as the default window size. At 1200 baud the default is 16 lines.

This size is used when the editor clears and refills the screen after a motion which moves far from the edge of the current window. Commands which take a new window size as count often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with an **R** or similar command or off the bottom with a command such as CR or CTRL-D. The window will revert to the last specified size the next time it is cleared and refilled.†

The scroll commands CTRL-D and CTRL-U likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus

† But not by a CTRL-L which just redraws the screen as it is.

10a+ – – – – –ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands which ignore any counts (such as CTRL-R), the rest of the editor commands use a count to indicate a simple repetition of their effect. Thus **5w** advances five words on the current line, while **5**CR advances five lines. A very useful instance of a count as a repetition is a count given to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you will delete first one and then three words. You can then delete two more words with **2.**.

2.7.3 More File Manipulation Commands

The following table lists the file manipulation commands which you can use when you are in **vi.**

:w	write back changes
:wq	write and quit
:x	write (if necessary) and quit (same as ZZ).
:e *name*	**edit file** *name*
:e!	**reedit, discarding changes**
:e + *name*	**edit, starting at end**
:e +*n*	**edit, starting at line** *n*
:e #	**edit alternate file**
:w *name*	**write file** *name*
:w! *name*	**overwrite file** *name*
:*x,y***w** *name*	**write lines** *x* **through** *y* **to** *name*
:r *name*	**read file** *name* **into buffer**
:r !*cmd*	**read output of** *cmd* **into buffer**
:n	**edit next file in argument list**
:n!	**edit next file, discarding changes to current**
:n *args*	**specify new argument list**

All of these commands are followed by a CR or ESC. The most basic commands are **:w** and **:e**. A normal editing session on a single file will end with a **ZZ** command. If you are editing for a long period of time you can give **:w** commands occasionally after major amounts of editing, and then finish with a **ZZ**. When you edit more than one file, you can finish with a **:w** and start editing a new file by giving a **:e** command, or set **autowrite** and use **:n** <file>.

If you make changes to the editor's copy of a file, but do not wish to write them back, then you must give an **!** after the command you would otherwise use; this forces the editor to discard any changes you have made. Use this carefully.

The **:e** command can be given a + argument to start at the end of the file, or a +*n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usually a scan like +*/pat* or +*?pat*. In adding new names to the **e** command, you can use the character % which is replaced by the current file name, or the character # which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file. Thus if you try to do a **:e** and get a message that you haven't written the file, you can give a **:w** command and then a **:e** # command to redo the previous **:e**.

You can write part of the buffer to a file by giving the starting and ending line numbers (of the text to be written) after the : and before the **w**, separated by ,'s. To find out the line numbers that delimit your text use CTRL-G. You can also mark these lines with **m** and then use an address of the form *'x,'y* on the **w** command here.

You can read another file into the buffer after the current line by using the **:r** command. You can similarly read in the output from a command, just use !*cmd* instead of a file name.

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command **:n**. It is also possible to restate the files to be edited by giving the **:n** command a list of file names, or a pattern to be expanded.

The **:ta** command is very useful for editing large programs. It utilizes a data base of function names and their locations, which can be created by programs such as *ctags,* to quickly find a function whose name you give. If the **:ta** command will require the editor to switch files, then you must **:w** or abandon any changes before switching. You can repeat the **:ta** command without any arguments to look for the same tag again.

2.7.4 More About Searching for Strings

When you are searching for strings in the file with / and **?**, the editor normally places you at the next or previous occurrence of the string. If you are using an operator such as **d**, **c** or **y**, you may wish to stop the action a line before the one containing the search pattern. You can give a search of the form */pat/−n* to refer to the *n*'th line before the next line containing *pat*, or you can use + instead of − to refer to the lines after the one containing *pat*. If you don't give a line offset, then the editor will act on characters up to the match place, rather than whole lines; thus use "**+0**" to affect to the line which matches.

You can have the editor ignore the case of words in searches by giving the command **.:se ic**CR The command **:se noic**CR turns this off.

Search strings may actually be regular expressions. If you do not want this facility, you should

:set nomagic

In this case, only the characters ^ and **$** are special in patterns. The character \ is also then special (as it is almost everywhere in the system), and may be used to get at an extended pattern matching facility. It is also necessary to use a \ before a / in a forward scan or a **?** in a backward scan, in any case. The following table gives the extended forms when **magic** is set.

^	at beginning of pattern, matches beginning of line
$	at end of pattern, matches end of line
.	**matches any character**
\<	**matches the beginning of a word**
\>	**matches the end of a word**
[*str*]	**matches any single character in** *str*
[*str*]	**matches any single character not in** *str*
[*x*−*y*]	**matches any character between** *x* **and** *y*
*	**matches any number of the preceding pattern**

If you use **nomagic** mode, then the **.** **[** and ***** primitives are given with a preceding \.

2.7.5 More About Input Mode

There are a number of characters that you can use to make corrections during input mode. These are summarized in the following table.

CTRL-H	deletes the last input character
CTRL-W	deletes the last input word, defined as by **b**
DEL	**erases a character**
CTRL-U	**deletes the input on this line**
\	**escapes a following ^H and your DEL and CTRL-U**
ESC	**ends an insertion**
CTRL-C	**interrupts an insertion, terminating it abnormally**
RETURN	**starts a new line**
CTRL-D	**backtabs over** *autoindent*
0CTRL-D	**kills all the** *autoindent*
CTRL CTRL-D	**same as 0CTRL-D, but restores indent next line**
CTRL-V	**quotes the next non-printing character into the file**

The usual way to correct input is to type CTRL-H for a single character, and to type one or more CTRL-W's to back over incorrect words. Your system kill character, CTRL-U, will erase all input on the current line. In general, you can not erase a line which was entered previously, but you can erase characters which were just entered using the insertion command. To make corrections on the previous line after a new line has been started, you can hit ESC to end the insertion, move over and make the correction, and then return to where you were to continue. The command **A** which appends at the end of the current line is often useful for continuing.

If you wish to type the erase or kill character (DEL or CTRL-U) then you must precede it with a \. A more general way of entering non-printing characters is to precede them with a CTRL-V. The CTRL-V echoes as a ^ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type almost any character and it will be inserted into the file at that point.*

If you are using **autoindent** you can backtab over the indent which it supplies by typing a CTRL-D. This backs up to a *shiftwidth* boundary. This only works immediately after the supplied **autoindent**.

When you are using **autoindent** you may place a label at the left margin of a line by typing ↑ and then CTRL-D. The editor will move the cursor to the left margin for one line, and restore the previous indent on the next. If you wish to kill the indent completely and not have it resume on the next line, type a **0** (zero) followed immediately by a CTRL-D.

2.7.6 Vi and Ex

vi is an editing mode within the editor **ex**. When you are running **vi** you can switch to the line oriented editor **ex** by typing **Q**. All of the **:** commands in **vi** are available in **ex**. Likewise, most **ex** commands can be invoked from **vi** using **:**. Just give them without the **:** and follow them with a CR.

In those rare instances when an internal error occurs in **vi**, you will automatically be switched to the command mode of **ex**. You can either save your work and quit the editor by giving the command **x** after the **ex** prompt **:**, or you can reenter **vi** by giving **ex** a **vi** command.

---

* Some exceptions are that the editor does not allow the **NULL** (^@) character to appear in files. Also the **LF** (linefeed or **^J**) character is used by the editor to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ^ before you type the character. In fact, the editor will treat a following letter as a request for the corresponding control character. This is the only way to type in a CTRL-S or CTRL-Q, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

There are several things that are easier to do with **ex** than **vi**, such as systematic changes in line oriented material. The advanced editing documents for the editor **ed** contain a lot more information about this style of editing. Experienced users often selectively use both **ex** command mode and **vi** command mode to speed the work they are doing.

Appendix A
TERMINAL TYPE CODES

The following is a listing of some terminal types and their codes. If your terminal does not appear in this listing, ask your system adminstrator for your terminal code.

Code	**Full name**
PC	IBM PC Console
pro	DEC Professional Console
hp2621a	Hewlett-Packard 2621A/P
hp2645	Hewlett-Packard 264x
adm31	Lear Siegler ADM-31
c100	Human Design Concept 100
bantam	Perkin-Elmer
h1500	Hazeltine 1500
h19	Heathkit h19
i100	Infoton 100
mime	Imitating a smart act4
t1061	Teleray 1061
vt52	Dec VT-52
vt100	Dec VT-100
aaadb	Annarbor Ambassador
vi50	Visual 50
vi200	Visual 200
tvi950	Televideo

Contents

3.1 INTRODUCTION 3-1

3.2 SIMPLE COMMANDS 3-2

3.3 SHELL PROCEDURES 3-8

3.4 KEYWORD PARAMETERS 3-23

Chapter 3

AN INTRODUCTION TO THE SHELL

3.1 INTRODUCTION

The **shell** is a command programming language that provides an interface to the VENIX operating system. Its features include control-flow primitives, parameter passing, variables, and string substitution. Constructs such as **while, if then else**, **case**, and **for** are available. Two-way communication is possible between the **shell** and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as **shell** input.

The **shell** can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through **pipes** can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file which allows command procedures to be stored for later use.

The **shell** is both a command language and a programming language that provides an interface to the VENIX operating system. This chapter describes, with examples, the VENIX operating system **shell**. The "Simple Commands" part of this section covers most of the everyday requirements of terminal users. Some familiarity with the VENIX operating system is an advantage when reading this section; refer to the chapter "VENIX For Beginners". The "Shell Procedures" part of this section describes those features of the **shell** primarily intended for use within **shell** commands or procedures. These include the control-flow primitives and string-valued variables provided by the **shell**. A knowledge of a programming language would be helpful when reading this section. The last part, "Keyword Parameters", describes the more advanced features of the **shell**. See Table 3.A for a defined listing of grammar words used in this section.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *User Reference Manual* for (1M) and the *Installation and System Manager's Guide*, for (7) and (8). Other references to entries of the form **name**(N), where "N" is the number (1), possibly followed by a letter, refer to entry **name** in section **N** of the *User Reference Manual*. Entries where "N" is a number (2 through 6) possibly followed by a letter, refer to entry **name** in section **N** of the *Programmer Reference Manual*.

3.2 SIMPLE COMMANDS

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls −l
```

prints a list of files in the current directory. The argument **−l** tells **ls**(1) to print status information, size, and the creation date for each file.

3.2.1 Background Commands

To execute a command, the **shell** normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file **pgm.c.** The trailing "&" is an operator that instructs the **shell** not to wait for the command to finish. To help keep track of such a process, the **shell** reports its process number following its creation. A list of currently active processes may be obtained using the **ps**(1) command.

3.2.2 Input/Output Redirection

Most commands produce output to the standard output that is initially connected to the terminal. This output may be directed to a file by the notation ">" thus:

```
ls −l >file
```

The notation >*file* is interpreted by the **shell** and is not passed as an argument to **ls**(1). If *file* does not exist, the **shell** creates it; otherwise, the original contents of *file* are replaced with the output from **ls**(1). Output may be appended to a file using the notation ">>" as follows:

```
ls −l >>file
```

In this case, *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by the notation "<" thus:

```
wc <file
```

The command **wc**(1) reads its standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then

```
wc −l <file
```

can be used.

3.2.3 Pipelines and Filters

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by |, between commands as in

```
ls −l | wc
```

Two or more commands connected in this way constitute a pipeline, and the overall effect is the same as:

```
ls -l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe [see **pipe**(2)] and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc**(1) when there is nothing to read and halting **ls**(1) when the pipe is full.

A filter is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**(1) selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from **ls** that contain the string "**old**". Another useful filter is **sort**(1). For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints only the number of file names in the current directory containing the string "**old**".

3.2.4 File Name Generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints only information relating to the file **main.c**. The "**ls -l**" command alone prints the same information about all files in the current directory.

The **shell** provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates as arguments to **ls**(1) all file names in the current directory that end in .c. The character "*" is a pattern that will match any string including the null string. In general, patterns are specified as follows:

*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters a through z.

The input

```
/usr/fred/test/?
```

matches all names in the directory **/usr/fred/test** that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all **core** files in subdirectories of **/usr/fred.** [The **echo**(1) command is a standard VENIX operating system command that prints its arguments, separated by blanks.] This last feature can be expensive, requiring a scan of all subdirectories of **/usr/fred**.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched. The input

```
echo *
```

will therefore echo all file names in the current directory not beginning with ".". The input

```
echo .*
```

will echo all those file names that begin with ".". This avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory", respectively. [Notice that **ls**(1) suppresses information for the files "." and "..".]

3.2.5 Quoting

Characters that have a special meaning to the **shell**, such as

```
< > * ? | &
```

are called metacharacters . A complete list of metacharacters is given in Table 3.B. Any character preceded by a \ is quoted and loses its special meaning, if any. The \ is elided so that

```
echo \?
```

will echo a single ?, and

```
echo \\
```

will echo a single \. To allow long strings to be continued over more than one line, the sequence **new line** (or CR) is ignored. The \ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

will echo

xx**xx**

The quoted string may not contain a single quote but may contain new lines which are preserved. This quoting mechanism is the simplest and is recommended for casual use. A third quoting mechanism using double quotes is also available and prevents interpretation of some but not all metacharacters. Details of quoting are described under "Evaluation and Quoting" in section "Keyword Parameters".

3.2.6 Prompting by the Shell

When the **shell** is used from a terminal, it will issue a prompt to the terminal user indicating it is ready to read a command from the terminal. By default, this prompt is "**$** ". The prompt may be changed by entering

PS1 = *newprompt*

This sets the prompt to be the string *newprompt*. If a new line is typed and further input is needed, the **shell** will issue the prompt "> ". Sometimes this can be caused by mistyping a quote mark. If it is unexpected, then an interrupt (DEL) will return the **shell** to read another command. The other prompt (">") may be changed by entering:

PS2 = more

3.2.7 The Shell and Login

Following the user's **login**(1), the **shell** is called to read and execute commands typed at the terminal. If the user's login directory contains the file **.profile**, then it is assumed to contain commands and is read immediately by the **shell** before reading any commands from the terminal.

3.2.8 Summary

ls	Prints the names of files in the current directory.
ls >file	Puts the output from **ls** into *file*.

ls | wc −l	Prints the number of files in the current directory.
ls | grep old	Prints those file names containing the string *old.*
ls | grep old | wc −l	Prints the number of files whose name contains the string *old.*
cc pgm.c &	Runs **cc** in the background.

3.3 SHELL PROCEDURES

The **shell** may be used to read and execute commands contained in a file. For example, the following call

```
sh file [ args ... ]
```

calls the **shell** to read commands from *file.* Such a file call is called a "command procedure" or "shell procedure". Arguments may be supplied with the call and are referred to in *file* using the positional parameters **$1, $2, ...** For example, if the file wg contains

```
who | grep $1
```

then the call

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

All VENIX operating system files have three independent attributes (often called "permissions"), **read, write**, and **execute** (rwx). The VENIX operating system command **chmod**(1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file wg has execute status (permission). Following this, the command

```
wg fred
```

is equivalent to the call

```
sh wg fred
```

This allows **shell** procedures and programs to be used interchangeably. In either case, a new process is created to execute the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#**. The name of the file being executed is available as **$0**.

A special **shell** parameter **$*** is used to substitute for all positional parameters except **$0**. A typical use of this is to provide some default arguments, as in,

```
nroff -T450 -cm $*
```

which simply prepends some arguments to those already given.

3.3.1 Control Flow—for

A frequent use of **shell** procedures is to loop through the arguments **($1, $2, ...)** executing commands once for each argument. An example of such a procedure is **tel** that searches the file **/usr/lib/telnos** that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of **tel** is

```
for i
do
     grep $i /usr/lib/telnos
done
```

The command

```
tel fred
```

prints those lines in **/usr/lib/telnos** that contain the string "fred".

The command

```
tel fred bert
```

prints those lines containing "fred " followed by those for "bert ".

The **for** loop notation is recognized by the **shell** and has the general form

```
for name in w1 w2
do
    command-list
done
```

A *command-list* is a sequence of one or more simple commands separated or terminated by a new line or a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a new line or semicolon. A *name* is a **shell** variable that is set to the words w1 w2 ... in turn each time the *command-list* following **do** is executed. If **in w1 w2 ..." is omitted, then the loop is executed once for each positional parameter; that is, in** *$** is assumed.

Another example of the use of the **for** loop is the **create** command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation >*file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or new line) is required before **done**.

3.3.2 Control Flow—case

A multiple way (choice) branch is provided for by the **case** notation. For example,

```
case $# in
     1) cat >>$1 ;;
     2) cat >>$2 <$1 ;;
     *) echo 'usage: append [ from ] to' ;;
esac
```

is an append command. (Note the use of semicolons to delimit the cases.) When called with one argument as in

append *file*

$# is the string "**1**", and the standard input is appended (copied) onto the end of *file* using the **cat**(1) command.

append *file1 file2*

appends the contents of *file1* onto *file2*. If the number of arguments supplied to append is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

case *word* **in**
pattern) *command-list*;;
...
esac

The **shell** attempts to match *word* with each *pattern* in the order in which the patterns appear. If a match is found, the associated *command-list* is executed and execution of the **case** is complete. Since * is the pattern that matches any string, it can be used for the default case.

Caution: No check is made to ensure that only one pattern matches the case argument.

The first match found defines the set of commands to be executed. In the example below, the commands following the second "*" will never be executed since the first "*" executes everything it receives.

```
case $# in
     *) ... ;;
     *) ... ;;
esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc**(1) command.

```
for i
do
     case $i in
          -[ocs]) ... ;;
          -*)     echo 'unknown flag $i' ;;
          *.c)    /lib/c0 $i ... ;;
          *)      echo 'unexpected argument $i' ;;
     esac
done
```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by a |. For example,

```
case $i in
     -x|-y)...
esac
```

is equivalent to

```
case $i in
     -[xy])...
esac
```

The usual quoting conventions apply so that

?

```
case $i in
     \?)...
```

will match the character ?.

3.3.3 Here Documents

The **shell** procedure **tel** described under "Control Flow—for" in this chapter uses the file **/usr/lib/telnos** to supply the data for **grep**(1). An alternative is to include this data within the **shell** procedure as a here document, as in,

```
for i
do
     grep $i <<!
     ...
     fred mh0123
     bert mh0789
     ...
!
done
```

In this example, the **shell** takes the lines between <<! and ! as the standard input for **grep**(1). The string "!" is arbitrary. The document is being terminated by a line that consists of the string following <<.

Parameters are substituted in the document before it is made available to **grep**(1) as illustrated by the following procedure called **edg**.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

edg *string1 string2 file*

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented using \ to quote the special character **$** as in

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

[This version of **edg** is equivalent to the first except that **ed**(1) will print a **?** if there are no occurrences of the string **$1.**]

Substitution within a here document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<#
...
#
```

The document is presented without modification to **grep**. If parameter substitution is not required in a here document, this latter form is more efficient.

3.3.4 Shell Variables

The **shell** provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by writing

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables user, box, and acct. A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

will echo fred.

Variables may be used interactively to provide abbreviations for frequently used strings.

For example,

```
b=/usr/fred/bin
mv file $b
```

will move the *file* from the current directory to the directory **/usr/fred/bin.** A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of **ps**(1) to the file **/tmp/psa,** whereas,

```
ps a >$tmpa
```

would cause the value of the variable tmpa to be substituted.

Except for **$?**, the following are set initially by the **shell.**

$?	The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a non-

zero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

$#	The number of positional parameters in decimal. Used, for example, in the **append** command to check the number of parameters.
$$	The process number of this **shell** in decimal. Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$!	The process number of the last process run in the background (in decimal).
$−	The current **shell** flags, such as **−x** and **−v.**

Some variables have a special meaning to the **shell** and should be avoided for general use.

$MAIL When used interactively, the **shell** looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the **shell** prints the message "you have mail" before prompting for the next command. This variable is typically set in the file **.profile** in the user's login directory. For example:

```
MAIL=/usr/mail/fred
```

$HOME The default argument for the **cd**(1) command. The current directory is used to resolve file name references that do not begin with a / and is changed using the **cd** command.

For example,

```
cd /usr/fred/bin
```

makes the current directory **/usr/fred/bin.** Then

```
cat wn
```

will print on the terminal the file **wn** in this directory. The command **cd**(1) with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the user's login profile.

$PATH A list of directories containing commands (the search path). Each time a command is executed by the **shell**, a list of directories is searched for an executable file. If **$PATH** is not set, the current directory, **/bin**, and **/usr/bin** are searched by default. Otherwise, *$PATH* consists of directory names separated by **:**. For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first **:**), **/usr/fred/bin, /bin,** and **/usr/bin** are to be searched in that order. In this way, individual users can have their own 'private' commands that are accessible independently of the current directory. If the command name contains a /, this directory search is not used; a single attempt is made to execute the command.

$PS1	The primary **shell** prompt string, by default, "$ ".
$PS2	The **shell** prompt when further input is needed, by default, "> ".
$IFS	The set of characters used by blank interpretation. (See "3.4.4 Evaluation and Quoting" in section "Keyword Parameters".)

3.3.5 Test Command

The **test** command is intended for use by **shell** programs. For example,

```
test −f file
```

returns zero exit status if *file* exists and nonzero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given below [see **test**(1) for a complete specification].

test s	true if the argument *s* is not the null string
test −f file	true if *file* exists
test −r file	true if *file* is readable
test −w file	true if *file* is writable
test −d file	true if *file* is a directory

3.3.6 Control Flow—while

The actions of the **for** loop and the **case** branch are determined by data available to the **shell**. A **while** or **until** loop and an **if then else** branch are also provided, whose actions are determined by the exit status returned by commands.

A **while** loop has the general form

```
while command-list1
do
    command-list2
done
```

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time round the loop *command-list1* is executed; if a zero exit status is returned, then *command-list2* is executed; otherwise, the loop terminates. For example,

```
while test $1
do
    ...
    shift
done
```

is equivalent to

```
for i
do
    ...
done
```

The **shift** command is a **shell** command that renames the positional parameters **$2, $3, ...** as **$1, $2, ...** and loses **$1**.

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop, the termination condition is reversed. For example,

```
until test −f file
do
    sleep 300
done
commands
```

will loop until *file* exists. Each time round the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

3.3.7 Control Flow—if

Also available is a general conditional branch of the form,

```
if command-list
then
   command-list
else
   command-list
fi
```

that tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the **test** command to test for the existence of a file as in

```
if test -f file
then
        process file
else
        do something else
fi
```

A multiple test **if** command of the form

```
if ...
then
        ...
else
        if ...
        then
                ...
        else
                if ...
                ...
                fi
        fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then
        ...
elif ...
then
        ...
elif ...
...
fi
```

The **touch** command changes the "last modified" time for a list of files. The command may be used in conjunction with **make**(1) to force recompilation of a list of files.

The following example is the **touch** command:

```
flag=
for i
do
        case $i in
                -c)     flag=N ;;
                *)      if test -f $i
                        then
                                ln $i junk$$
                                rm junk$$
                        elif test $flag
                        then
                                echo file \'$i\' does not exist
                        else
                                >$i
                        fi ;;
        esac
done
```

The **−c** flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The **shell** variable *flag* is set to some non-null string if the **−c** argument is encountered. The commands

ln ...; rm ...

make a link to the file and then remove it.

The sequence

```
if command1
then
        command2
fi
```

may be written

command1 **&&** *command2*

Conversely,

command1 || *command2*

executes *command2* only if *command1* fails. In each case, the value returned is that of the last simple command executed.

3.3.8 Command Grouping

Commands may be grouped in two ways,

{ *command-list* ; }

and

(*command-list* **)**

The first form, *command-list*, is simply executed. The second form executes *command-list* as a separate process. For example,

(cd *x*; **rm** *junk* **)**

executes **rm** *junk* in the directory *x* without changing the current directory of the invoking **shell**.

The commands

cd *x*; **rm** *junk*

have the same effect but leave the invoking **shell** in the directory *x*.

3.3.9 Debugging Shell Procedures

The **shell** provides two tracing mechanisms to help when debugging **shell** procedures. The first is invoked within the procedure as

set −v

(*v* for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by entering

sh −v *proc ...*

where *proc* is the name of the **shell** procedure. This flag may be used in conjunction with the **−n** flag which prevents execution of subsequent commands. (Note that typing "**set −n**" at a terminal will render the terminal useless until an end-of-file is typed.)

The command

set −x

will produce an execution trace with flag **−x**. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see the effect it has.) Both flags may be turned off by typing

set −

and the current setting of the **shell** flags is available as **$−**.

3.4 KEYWORD PARAMETERS

Shell variables may be given values by assignment or when a **shell** procedure is invoked. An argument to a **shell** procedure of the form *name*=*value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking **shell** is not affected. For example,

user = fred *command*

will execute *command* with user set to fred. The **−k** flag causes arguments of the form *name = value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as positional parameters **$1**, **$2**, **...** .

The **set** command may also be used to set positional parameters from within a procedure.

For example,

set − *

will set **$1** to the first file name in the current directory, **$2** to the next, etc. Note that the first argument, −, ensures correct treatment when the first file name begins with a −.

3.4.1 Parameter Transmission

When a **shell** procedure is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a **shell** procedure by specifying in advance that such parameters are to be exported. For example,

export *user box*

marks the variables *user* and *box* for export. When a **shell** procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking **shell**. It is generally true of a **shell** procedure that it may not modify the state of its caller without an explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared readonly. The form of this command is the same as that of the **export** command,

readonly *name ...*

Subsequent attempts to set readonly variables are illegal.

3.4.2 Parameter Substitution

If a **shell** parameter is not set, then the null string is substituted for it. For example, if the variable d is not set,

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d-.}
```

which will echo the value of the variable d if it is set and "." otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d- '*'}
```

will echo * if the variable d is not set. Similarly,

```
echo ${d-$1}
```

will echo the value of d if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.}
```

and if d were not previously set, it will be set to the string ".". (The notation ${...=...} is not available for positional parameters.)

If there is no sensible default, the notation

```
echo ${d?message}
```

will echo the value of the variable d if it has one; otherwise, *message* is printed by the **shell** and execution of the **shell** procedure is abandoned. If *message* is absent, a standard message is printed. A **shell** procedure that requires some parameters to be set might start as follows:

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command built into the **shell** and does nothing once its arguments have been evaluated. If any of the variables user, acct, or bin are not set, the **shell** will abandon execution of the procedure.

3.4.3 Command Substitution

The standard output from a command can be substituted in a similar way to parameters. The command **pwd**(1) prints on its standard output the name of the current directory. For example, if the current directory is /usr/fred/bin, the command

```
d=‘pwd‘
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between single left quotes (‘...‘) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ‘ must be escaped using a \.

For example,

```
ls ‘echo "$1"‘
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including here documents), and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within **shell** procedures. An example of such a command is **basename**, which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string "main". Its use is illustrated by the following fragment from a **cc**(1) command.

```
case $A in
        ...
        *.c)      B=`basename $A .c`
        ...
esac
```

that sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

- **for *i* in `ls −t`; do ...**

  The variable *i* is set to the names of files in time order, most recent first.

- **set `date`; echo $6 $2 $3, $4**

  will print, e.g., 1977 Nov 1, 23:59:59

3.4.4 Evaluation and Quoting

The **shell** is a macro processor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in Table 5.A. Before a command is executed, the following substitutions occur:

1. Parameter substitution, e.g., **$user**

2. Command substitution, e.g., **'pwd'**

   Only one evaluation occurs so that if, for example, the value of the variable *X* is the string "**$y**" then

   ```
   echo $X
   ```

   will echo "**$y**".

3. Blank interpretation

   Following the above substitutions, the resulting characters are broken into nonblank words (blank interpretation). For this purpose, blanks are the characters of the string "*$IFS*". By default, this string consists of blank, tab, and newline. The null string is not regarded as a word unless it is quoted. For example,

   ```
   echo ″
   ```

   will pass on the null string as the first argument to **echo**, whereas

   ```
   echo $null
   ```

   will call **echo** with no arguments if the variable *null* is not set or set to the null string.

4. File name generation

   Each word is then scanned for the file pattern characters *****, **?**, and **[...]**; and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the word used for a **case** branch.

As well as the quoting mechanisms described earlier using \ and **`...`**, a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occurs; but file name generation and the interpretation of blanks does not.

The following characters have a special meaning within double quotes and may be quoted using \.

$	**parameter substitution**
‘	**command substitution**
″	**ends the quoted string**
\	**quotes the special characters $ ‘ ″ **

For example,

```
echo "$x"
```

will pass the value of the variable *x* as a single argument to **echo.** Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation **$@** is the same as **$*** except when it is quoted. Inputting

```
echo "$@"
```

will pass the positional parameters, unevaluated, to **echo** and is equivalent to

```
echo "$1" "$2" ...
```

The following illustration gives, for each quoting mechanism, the **shell** metacharacters that are evaluated.

metacharacter

Quoting mechanism	\	$	*	‘	″	′
′	n	n	n	n	n	t
‘	y	y	y	t	y	y
″	y	y	n	y	t	n

In cases where more than one evaluation of a string is required, the built-in command **eval** may be used. For example, if the variable *X* has the value "**$y**" and if *y* has the value "**pqr**", then

```
eval echo $X
```

will echo the string "pqr".

In general, the **eval** command evaluates its arguments (as do all commands) and treats the result as input to the **shell.** The input is read and the resulting command(s) executed. For example,

```
wg=′eval who|grep′
$wg fred
```

is equivalent to

```
who|grep fred
```

In this example, **eval** is required since there is no interpretation of metacharacters, such as |, following substitution.

3.4.5 Error Handling

The treatment of errors detected by the **shell** depends on the type of error and on whether the **shell** is being used interactively. An interactive **shell** is one whose input and output are connected to a terminal [as determined by **ioctl**(2)]. A **shell** invoked with the **−i** flag is also interactive.

Execution of a command (see also "Command Execution") may fail for any of the following reasons:

- Input/output redirection may fail. For example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault" signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the **shell** will go on to execute the next command. Except for the last case, an error message will be printed by the **shell**. All remaining errors cause the **shell** to exit from a command procedure. An interactive **shell** will return to read another command from the terminal. Such errors include the following:

- Syntax errors, e.g., if ...then... done
- A signal such as interrupt. The **shell** waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as **cd**(1).

The **shell** flag **−e** causes the **shell** to terminate if any error is detected. The following is a list of the VENIX operating system signals:

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	Kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination [from **kill**(1)]

The VENIX operating system signals marked with an asterisk "*" as shown in the list produce a core dump if not caught. However, the **shell** itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to **shell** programs are 1, 2, 3, 14, and 15.

3.4.6 Fault Handling

Shell procedures normally terminate when an interrupt is received from the terminal. The **trap** command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt); and if this signal is received, it will execute the following commands:

```
rm /tmp/ps$$; exit
```

The **exit** is another built-in command that terminates execution of a **shell** procedure. The **exit** is required; otherwise, after the trap has been taken, the **shell** will resume executing the procedure at the place where it was interrupted.

VENIX operating system signals can be handled in one of three ways.

1. They can be ignored, in which case the signal is never sent to the process.
2. They can be caught, in which case the process must decide what action to take when the signal is received.
3. They can be left to cause termination of the process without it having to take any further action.

If a signal is being ignored on entry to the **shell** procedure, for example, by invoking it in the background (see "Command Execution"), **trap** commands (and the signal) are ignored.

The use of **trap** is illustrated by this modified version of the **touch** command illustrated below:

```
flag=
trap ′rm −f junk$$; exit′ 1 2 3 15
for i
do
     case $i in
     −c)  flag=N ;;
     *)   if test −f $i
          then
                  ln $i junk$$; rm junk$$
          elif test $flag
          then
                  echo file \′$i\′ does not exist
          else
                  >$i
          fi ;;
     esac
done
```

The cleanup action is to remove the file *junk$$*. The **trap** command appears before the creation of the temporary file; otherwise, it would be possible for the process to die without removing the file.

Since there is no signal 0 in the VENIX operating system, it is used by the **shell** to indicate the commands to be executed on exit from the **shell** procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following:

```
trap ′′ 1 2 3 15
```

is a fragment taken from the **nohup**(1) command which causes the VENIX operating system HANGUP, INTERRUPT, QUIT, and SOFTWARE TERMINATION signals to be ignored both by the procedure and by invoked commands.

Traps may be reset by entering

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The **scan** procedure is an example of the use of **trap** where there is no exit in the trap command. The **scan** takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when **scan** is waiting for input. The **scan** procedure follows:

```
d=`pwd`
for i in *
do
        if test -d $d/$i
        then
              cd $d/$i
              while echo "$i:" && trap exit 2 && read x
              do
                      trap : 2
                      eval $x
              done
        fi
done
```

The **read x** is a built-in command that reads one line from the standard input and places the result in the variable *x*. It returns a nonzero exit status if either an end-of-file is read or an interrupt is received.

3.4.7 Command Execution

To run a command (other than a built-in), the **shell** first creates a new process using the system call **fork**(2). The execution environment for the command includes input, output, and the states of signals and is established in the child process before the command is executed. The built-in command **exec** is used in rare cases when no fork is required and simply replaces the **shell** with a new command. For example, a simple version of the **nohup** command looks like

```
trap ″ 1 2 3 15
exec $*
```

The **trap** turns off the signals specified so that they are ignored by subsequently created commands, and **exec** replaces the **shell** by the command specified.

Most forms of input/output redirection have already been described. In the following, *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is **.c*. Input/output specifications are evaluated left to right as they appear in the command. Some input/output specifications are as follows:

> **word**	The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
>> **word**	The standard output is sent to file *word*. If the file exists, then output is appended (by seeking to the end); otherwise, the file is created.
< **word**	The standard input (file parameter 0) is taken from the file *word*.
<< **word**	The standard input is taken from the lines of **shell** input that follow up to but not including a line consisting only of *word*. If *word* is quoted, no interpretation of the document occurs. If *word* is not quoted, parameter and

	command substitution occur and \ is used to quote the characters \, **$**, ‘, and the first character of *word*. In the latter case, **newline** is ignored (e.g., quoted strings).
>& digit	The file descriptor *digit* is duplicated using the system call **dup**(2), and the result is used as the standard output.
<& digit	The standard input is duplicated from file descriptor *digit*.
<&–	The standard input is closed.
>&–	The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*. Another example,

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking, file descriptor 2 is created by duplicating file descriptor 1; but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file **/dev/null.** This prevents two processes (the **shell** and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the **shell** to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the VENIX operating system convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the **shell** command **trap** has no effect for an ignored signal.

3.4.8 Invoking the Shell

The following flags are interpreted by the **shell** when it is invoked. If the first character of argument zero is a minus, commands are read from the file **.profile.**

−c string	If the **−c** flag is present, then commands are read from *string*.
−s	If the **−s** flag is present or if no arguments remain, commands are read from the standard input. **Shell** output is written to file descriptor 2.
−i	If the **−i** flag is present or if the **shell** input and output are attached to a terminal [as told by **getty**(8)], this **shell** is *interactive*. In this case, TERMINATE is ignored (so that **kill 0** does not kill an interactive **shell**, and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the **shell**.

TABLE 3.A

GRAMMAR

item	*word*
	input-output
	name = *value*
simple-command:	*item*
	simple-command item
command:	*simple-command*
	(*command-list* **)**
	{ *command-list* }
	for *name* **do** *command-list* **done**
	for *name* **in** *word ...* **do** *command-list* **done**
	while *command-list* **do** *command-list* **done**
	until *command-list* **do** *command-list* **done**
	case *word* **in** *case-part ...* **esac**
	if *command-list* **then** *command-list else-part* **fi**
pipeline:	*command*
	pipeline | *command*
andor:	*pipeline*
	andor **&&** *pipeline*
	andor || *pipeline*
command-list:	*andor*
	command-list **;**
	command-list **&**
	command-list ; *andor*
	command-list **&** *andor*

input-output:	> *word* < *word* >> *word* << *word*
file	*word* **&** *digit* **&** –
case-part:	*pattern*) *command-list* ;;
pattern:	*word* *pattern* | *word*

else-part:	**elif** *command-list* **then** *command-list else-part*
	else *command-list*

empty:	*empty*
word:	sequence of nonblank characters
name	sequence of letters, digits, or underscores starting with a letter
digit:	**0 1 2 3 4 5 6 7 8 9**

TABLE 3.B

METACHARACTERS AND RESERVED WORDS

(a)	*syntactic:*	
	|	pipe symbol
	&&	'andf' symbol
	||	'orf' symbol
	;	command separator
	;;	case delimiter
	&	background commands
	( )	command grouping
	<	input redirection
	<<	input from a here document
	>	output creation
	>>	output append
(b)	*patterns:*	
	*	match any character(s) including none
	?	match any single character
	[...]	match any of the enclosed characters

(c) *substitution:*

${...}	substitute **shell** variable
‘...‘	substitute command output

(d) *quoting:*

\	quote the next character
′...′	quote the enclosed characters except for the ′
″...″	quote the enclosed characters except for the **$**, ‘ ,\, and ″

(e) *reserved words:*

if then else elif fi
case in esac
for while until do done
{ } **[ ] test**

Contents

4.1 INTRODUCTION 4-1

4.2 TERMINAL USAGE OF THE SHELL 4-1

4.3 DETAILS ON THE SHELL FOR TERMINAL USERS 4-12

4.4 SHELL CONTROL STRUCTURES
AND COMMAND SCRIPTS 4-26

4.5 MISCELLANEOUS SHELL MECHANISMS 4-38

Chapter 4

AN INTRODUCTION TO THE C SHELL

4.1 INTRODUCTION

A **shell** is a command language interpreter. **csh** is the name of one particular command interpreter on VENIX. The primary purpose of **csh** is to translate command lines typed at a terminal into system actions, such as invocation of other programs. **csh** is a user program just like any you might write.

In addition to this document, you will want to refer to a copy of the **csh** *User Reference Manual*. The **csh** documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

There are important words; names of commands, and words which have special meaning in discussing the shell and VENIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

4.2 TERMINAL USAGE OF THE SHELL

4.2.1 The Basic Notion of Commands

A **shell** in VENIX acts mostly as a medium through which other **commands** are invoked. While it has a set of **builtin** commands which it performs directly, most useful commands are, in fact, external to the shell. The shell is distinguished from the command interpreters of other systems by the fact that it is a user program, and it is used almost exclusively as a mechanism for invoking other programs.

Commands in the VENIX system expect a list of strings or **words** as arguments. The command

```
mail bill
```

consists of two words. The first word **mail** names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to run it for you. It will look in a number of **directories** for a file with the name **mail** which is expected to contain the mail program.

The rest of the words of the command are given to the command itself to execute. In this case we specified also the word **bill** which is interpreted by the **mail** program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the **mail** command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
        Bill
%
```

Here we typed a message to send to **bill** and ended this message with a control-d which sent an end-of-file to the mail program. The mail program then transmitted our message. The characters '%' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the '%' prompt the shell was reading command input from our terminal. We typed a complete command 'mail bill'. The shell then executed the **mail** program with argument **bill** and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file after which the shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another '%' prompt.

This is the essential pattern of all interaction with VENIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

4.2.2 Flag Arguments

A useful notion in VENIX is that of a **flag** argument. While many arguments to commands specify file names or user names some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '−'. Thus the command

```
ls
```

will produce a list of the files in the current directory. The option **−s** is the size option, and

```
ls −s
```

causes **ls** to also give, for each file the size of the file in blocks of 512 characters. The manual page for each command in the *User Reference Manual* gives the available options for each command. The **ls** command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most VENIX utilities perform only one or two functions rather than having a large number of hard to remember options.

4.2.3 Output to Files

Many commands may read input or write output to files rather than simply taking input and output from the terminal. Each such command could take special words as arguments indicating where the output is to go. It is simpler, and usually sufficient, to connect these commands to files to which they wish to write, within the shell itself, and just before they are executed.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default **standard output** for the date command and the date command prints the date on its standard output. The shell lets us redirect the **standard output** of a command through a notation using the **metacharacter** '>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the **date** command in an environment where its standard output is the file 'now' rather than our terminal. Thus this command places the current date and time in the file 'now'. It is important to know that the **date** command was unaware that its output was going to a file rather than to our terminal. The shell performed this **redirection** before the command began executing.

The file 'now' need not have existed before the **date** command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option **noclobber** exists to prevent this from happening accidentally; it is discussed in section 2.2.

4.2.4 Metacharacters in the Shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have **syntactic** and **semantic** meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of **quotation** which allows us to create words which contain **metacharacters** and to thus work without constantly worrying about whether certain characters are metacharacters.

Note that the shell is only reading input when it has prompted with '% '. Thus metacharacters will normally have effect only then. We need not worry about placing shell metacharacters in a letter we are sending via **mail.**

4.2.5 Input from Files; Pipelines

We learned above how to route the standard output of a command to a file. It is also possible to route the standard input of a command from a file. This is not often necessary since most commands will read from a file name given as argument. We can give the command

```
sort < data
```

to run the **sort** command with standard input, where the command normally reads, from the file 'data'. We would more likely say

```
sort data
```

letting the **sort** command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the sort program would sort lines from its **standard input.** Since we did not **redirect** the standard input, it would sort lines as we typed them on the terminal until we typed a control-d to generate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of the next, i.e. to run the commands in a sequence known as a **pipeline.** For instance the command

```
ls −s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which **ls** sorts. We could look at the many options of **ls** to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the **sort** command, combining it with **ls** to get what we want.

The **−n** option of sort specifies a numeric sort rather than an alphabetic sort. Thus

```
ls −s | sort −n
```

specifies that the output of the **ls** command run with the option −s is to be **piped** to the command **sort** run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the **−r** reverse sort option and the **head** command in combination with the previous command doing

```
ls −s | sort −n −r | head −5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the **sort** command asking it to sort numerically in reverse order (largest first). This output has then been run into the command **head** which gives us the first few lines out. In this case we have asked **head** for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The metanotation introduced above is called the **pipe** mechanism. Commands separated by '|' characters are connected together by the shell and the output of each is run into the input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

4.2.6 Filenames

Many commands to be executed will need the names of files as arguments. VENIX pathnames consist of a number of components separated by '/'. Each component except the last names a directory in which the next component resides. Thus the pathname

```
/etc/motd
```

specifies a file in the directory 'etc' which is a subdirectory of the **root** directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. Filenames which do not begin with '/' are interpreted starting at the current **working** directory. This directory is, by default, your **home** directory and can be changed dynamically by the **chdir** change directory command.

Most filenames consist of a number of alphanumeric characters and '.'s. In fact, all printing characters except '/' may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' is not a shell-metacharacter and is often used as the prefix with an **extension** of a base name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. They share a **root** portion of a name (a root portion being that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file 'prog.c' might be the source for a C program, the file 'prog.o' the corresponding object file, the file 'prog.errs' the errors resulting from a compilation of the program and the file 'prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the metanotation

```
prog.*
```

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are sorted into the argument list to the command alphabetically. Thus the command

```
echo prog.*
```

will echo the names

```
prog.c prog.errs prog.o prog.output
```

Note that the names are in lexicographic order here, and a different order than we listed them above. The **echo** command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by filename expansion of the metasyntax in the one input word.

Other metanotations for **filename expansion** are also available. The character '?' matches any single character in a filename. Thus

```
echo ? ?? ???
```

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently lexicographically sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

prog.[co]

will match

prog.c prog.o

in the example above. We can also place two characters astride a '−' in this notation to denote a range. Thus

chap.[1−5]

might match files

chap.1 chap.2 chap.3 chap.4 chap.5

if they existed. This is shorthand for

chap.[12345]

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an **argument list)** contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

No match.

Another very important point is that the character '.' at the beginning of a filename is treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the current directory which have special meaning to the system, as well as other files such as **.cshrc** which are not normally visible. We will discuss the special role of the file **.cshrc** later.

Another filename expansion mechanism gives access to the pathname of the **home** directory of other users. This notation consists of the character '~' followed by another users login name. For instance the word '~bill' would map to the pathname '/mnt/bill' if the home directory for 'bill' was in the directory

'/mnt/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~/mbox'. This notation is expanded by the shell into the file 'mbox' in your **home** directory. This can be very useful if you have used **chdir** to change to another users directory and have found a file you wish to copy using **cp.** You can do

```
cp thatfile ~
```

which will be expanded by the shell to

```
cp thatfile /mnt/bill
```

e.g., which the copy command will interpret as a request to make a copy of 'thatfile' in the directory '/mnt/bill'. The '~' notation doesn't, by itself, force named files to exist. This is useful, for example, when using the **cp** command, e.g.

```
cp thatfile ~/saveit
```

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of word which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.5.2, as it is used much less frequently.

4.2.7 Quotation

We have already seen a number of metacharacters used by the shell. These metacharacter pose a problem in that we cannot use them directly as parts of words. Thus the command

```
echo *
```

will not echo the character '*'. It will either echo an sorted list of filenames in the current directory, or print the message 'No match' if there are no files in the current directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.' or '−' in an argument word to a command is to enclose it with single quotation characters '′', i.e.

```
echo ′*′
```

There is one special character '!' which is used by the **history** mechanism of the shell and which cannot be **escaped** in this way. It and the character '′' itself can be preceded by a single '\' to prevent their special meaning. These two mechanisms suffice to place any printing character into a word which is an argument to a shell command.

4.2.8 Terminating Commands

When you are running a command from the shell and the shell is dormant waiting for it to complete there are a couple of ways in which you can force such a command to complete. For instance if you type the command

```
cat /etc/passwd
```

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT signal to the **cat** command by hitting the CTRL-C key on your terminal. Actually, hitting this key sends this INTERRUPT signal to all programs running on your terminal, including your shell. The shell normally ignores such signals however, so that the only program affected by the INTERRUPT will be **cat.** Since **cat** does not take any precautions to catch this signal the INTERRUPT will cause it to terminate. The shell notices that **cat** has died and prompts you again with '% '. If you hit INTERRUPT again, the shell will just repeat its prompt since it catches INTERRUPT signals and chooses to continue to execute commands rather than going away like **cat** did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the **mail** program in the first example above was terminated when we hit a control-d which generates and end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; VENIX then logs you off the system. Since this means that typing too many control-d's can accidentally log us off, the shell has a mechanism for preventing this. This **ignoreeof** option will be discussed in section 4.3.2.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a control-d. This is because it read to the end-of-file of our file 'prepared.text' in which we placed a message for 'bill' with an editor. We could also have done

```
cat prepared.text | mail bill
```

since the **cat** command would then have written the text through the pipe to the standard input of the mail command. When the **cat** command completed it would have terminated, closing down the pipeline and the **mail** command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, generated by a control-\. This will usually provoke the shell to produce a message like:

```
a.out: Quit -- Core dumped
```

indicating that a file 'core' has been created containing information about the program 'a.out's state when it ran amuck. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the **core file** is.

If you run background commands (as explained in section 4.3.6) then these commands will ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the **kill** program. See section 4.3.6 for an example.

4.2.9 What Now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to VENIX and type the following command to the system:

```
chsh myname /bin/csh
```

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. **You only have to do this once; it takes effect at next login.** You are now ready to try using **csh.**

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to **csh** so you should change your shell to **csh** before you begin reading it.

4.3 DETAILS ON THE SHELL FOR TERMINAL USERS

4.3.1 Shell Startup and Termination

When you login, the shell is placed by the system in your **home** directory and begins by reading commands from a file **.cshrc** in this directory. All shells which you may create during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A **login** shell, executed after you login to the system, will, after it reads commands from **.cshrc,** read commands from a file **.login** also in your home directory. This file contains commands which you wish to do each time you login to the VENIX system. An example **.login** file is shown below:

```
tset −d adm3a −p adm3a
fixexrc
set history=20
set time=3
```

This file contains four commands to be executed by VENIX each time you login. The first is a **tset** command which informs the system that you are using a Lear-Siegler ADM−3A terminal and that if you are on a patchboard port you are also on an ADM−3A. The second command is a **fixexrc** which manipulates your **ex** startup file in certain ways if you are on a dialup port. You need not be concerned with exactly what this command does. In general you may have certain commands in your **.login** which are particular to you.

The next two **set** commands are interpreted directly by the shell and affect the values of certain shell variables to modify the future behavior of the shell. Setting the variable **time** tells the shell to print time statistics on commands which take more than a certain threshold of machine time (in this case 3 CPU seconds). Setting the variable **history** tells the shell how much history of previous command words it should save in case I wish to repeat or rerun modified versions of previous commands. Since there is a certain overhead in this mechanism the shell does not set this variable by default, but rather lets users who wish to use the mechanism set it themselves. The value of 20 is a reasonably large value to assign to **history.** More casual users of the **history** mechanism would probably set a value of 5 or 10. The use of the **history** mechanism will be described subsequently.

After executing commands from **.login** the shell reads commands from your terminal, prompting for each with '%'. When it receives an end-of-file from the terminal, the shell will print 'logout' and execute commands from the file '.logout' in your home directory. After that the shell will die and VENIX will log you off the system. If the system is not going down, you will receive a new login message. In any case, after the 'logout' message the shell is doomed and will take no further input from the terminal.

4.3.2 Shell Variables

The shell maintains a set of **variables.** We saw above the variables **history** and **time** which had values '20' and '3'. In fact, each shell variable has as value an array of zero or more **strings.** Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

set *name = value*

Shell variables may be used to store values which are to be reintroduced into commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable **path.** This variable contains a sequence of directory names where the shell searches for commands. The **set** command shows the value of all variables currently defined (we usually say **set)** in the shell. The default value for path will be shown by **set** to be

```
% set
argv
home    /mnt/bill
path    (. /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
%
```

This notation indicates that the variable path points to the current directory '.' and then '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). The most heavily used system commands live in '/bin'. Less heavily used system commands live in '/usr/bin'.

A number of new programs on the system live in the directory '/usr/new'. If we wish, as well we might, all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /usr/new /bin /usr/bin)
```

in our file **.cshrc** in our home directory. Try doing this and then logging out and back in and do

```
set
```

again to see that the value assigned to **path** has changed.

Other useful built in variables are the variable **home** which shows your home directory, the variable **ignoreeof** which can be set in your **.login** file to tell the shell not to exit when it receives an end-of-file from a terminal. To logout from VENIX with **ignoreeof** set you must type

```
logout
```

This is one of several variables which the shell does not care about the value of, only whether they are **set** or **unset.** Thus to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

Both **set** and **unset** are built-in commands of the shell.

Finally, some other built-in shell variables of use are the variables **noclobber** and **mail.** The metasyntax

> *filename*

which redirects the output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
set noclobber
```

in your **.login** file. Then trying to do

```
date > now
```

would cause a diagnostic if 'now' existed already. You could type

```
date >! now
```

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok.

If you receive mail frequently while you are logged in and wish to be informed of the arrival of this mail you can put a command

set mail = /usr/mail/*yourname*

in your **.login** file. Here you should change *yourname* to your login name. The shell will look at this file every 10 minutes to see if new mail has arrived. If you receive mail only infrequently you are better off not setting this variable. In this case it will only serve to delay the shells response to you when it checks for mail.

The use of shell variables to introduce text into commands, which is most useful in shell command scripts, will be introduced in section 4.3.4.

4.3.3 The Shell's History List

The shell can maintain a history list into which it places the words of previous commands. It is possible to use a metanotation to reintroduce commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

Consider the following transcript:

```
% where michael
michael is on tty0    dialup        300 baud      642-7927
% write !$
write michael
Long time no see michael.
Why don't you call me at 524-4510.
EOF
%
```

Here we asked the system where **michael** was logged in. It told us he was on 'tty0' and we told the shell to invoke a 'write' command to '!$'. This is a history notation which means the last word of the last command executed, in this case ‘michael’. The shell performed this substitution and then echoed the command as it would execute it. Let us assume that we don’t hear anything from michael. We might do

```
% ps t0
  PID TTY TIME COMMAND
 4808  0    0:05 –
% !!
ps t0
  PID TTY TIME COMMAND
 5104  0    0:00 – 7
% !where
where michael
michael is not logged in
%
```

Here we ran a **ps** on the teletype **michael** was logged in on to see that he had a shell. Repeating this command via the history substitution ‘!!’ we saw that he had logged out and that only a **getty** process was running on his terminal. Repeating the **where** command showed that he was indeed gone, most likely having hung up the phone in order to be able to call.

This illustrates several useful features of the history mechanism. The form ‘!!’ repeats the last command execution. The form ‘!string’ repeats the last command which began with a word of which ‘string’ is a prefix. Another useful command form is ‘↑lhs↑rhs’ performing a substitute similar to that in **ed** or **ex.** Thus after

```
% cat ~bill/csh/sh..c
/mnt/bill/csh/sh..c: No such file or directory
% ↑..↑.
cat ~bill/csh/sh.c
#include "sh.h"

/*
 * C Shell
 *
 * Bill Joy, UC Berkeley
 * October, 1978
 */

char      *pathlist[] =      { SRCHP
%
```

here we used the substitution to correct a typing mistake, and then rubbed the command out after we saw that we had found the file that we wanted. The substitution changed the two '.' characters to a single '.' character.

After this command we might do

```
% !! | lpr
cat ~bill/csh/sh.c | lpr
```

to put a copy of this file on the line printer, or (immediately after the **cat** which worked above)

```
% pr !$ | lpr
pr ~bill/csh/sh.c | lpr
%
```

to print a copy on the printer using **pr.**

More advanced forms of the history mechanism are also possible. A notion of modification on substitutions allows one to say (after the first successful **cat** above).

```
% cd !$:h
cd ~bill/csh
%
```

The trailing ':h' on the history substitution here causes only the head portion of the pathname reintroduced by the history mechanism to be substituted. This mechanism and related mechanisms are used less often than the forms above.

A complete description of history mechanism features is given in the C shell manual pages in this volume.

4.3.4 Aliases

The shell has an **alias** mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to the macro facility of many assemblers.

Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment and commands such as **chdir** which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'Mail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
alias mail Mail
```

in your **.login** file, the shell will transform an input line of the form

```
mail bill
```

into a call on 'Mail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '−s'. We can do

```
alias ls ls −s
```

or even

```
alias dir ls −s
```

creating a new command syntax 'dir' which does an 'ls −s'. If we say

```
dir ~bill
```

then the shell will translate this to

```
ls −s /mnt/bill
```

Thus the **alias** mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an **ls** command after each change directory **cd** command. We enclosed the entire alias definition in '′' characters to prevent most substitutions from occurring and the character ';' from being recognized as a parser metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing **cd** command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

```
alias whois 'grep \!↑ /etc/passwd'
```

defines a command which looks up its first argument in the password file.

4.3.5 Detached Commands; >> and >& Redirection

There are a few more metanotations useful to the terminal user which have not been introduced yet. The metacharacter '&' may be placed after a command, or after a sequence of commands separated by ';' or '|'. This causes the shell to not wait for the commands to terminate before prompting again. We say that they are **detached** or **background** processes. Thus

```
% pr ~bill/csh/sh.c | lpr &
5120
5121
%
```

Here the shell printed two numbers and came back very quickly rather than waiting for the **pr** and **lpr** commands to finish. These numbers are the process numbers assigned by the system to the **pr** and **lpr** commands.†

Since havoc would result if a command run in the background were to read from your terminal at the same time as the shell does, the default standard input for a command run in the background is not your terminal, but an empty file called '/dev/null'. Commands run in the background are also made immune to INTERRUPT and QUIT signals which you may subsequently generate at your terminal.*

If you intend to log off the system before the command completes you must run the command immune to HANGUP signals. This is done by placing the word 'nohup' before each program in the command, i.e.:

```
nohup man csh | nohup lpr &
```

In addition to the standard output, commands also have a diagnostic output which is normally directed to the terminal even when the standard output is directed to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

```
command >& file
```

† Running commands in the background like this tends to slow down the system and is not a good idea if the system is overloaded. When overloaded, the system will just bog down more if you run a large number of processes at once.

* If a background command stops suddenly when you hit INTERRUPT or QUIT it is likely a bug in the background program.

The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. of the standard output. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon **lpr**. #

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file.†

4.3.6 Useful Built-in Commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The **alias** command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given an argument such as

```
alias ls
```

to show the current alias for, e.g., 'ls'.

# A command form

```
command >&! file
```

exists, and is used when **noclobber** is set and **file** already exists.

† If **noclobber** is set, then an error will result if **file** does not exist, otherwise the shell will create **file** if it doesn't exist. A form

```
command >>! file
```

makes it not be an error for file to not exist when **noclobber** is set.

The **cd** and **chdir** commands are equivalent, and change the working directory of the shell. It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. Thus after you login you can do

```
% pwd
/mnt/bill
% mkdir newpaper
% chdir newpaper
% pwd
/mnt/bill/newpaper
%
```

after which you will be in the directory **newpaper.** You can place a group of related files there. You can return to your 'home' login directory by doing just

```
chdir
```

with no arguments. We used the **pwd** print working directory command to show the name of the current directory here. The current directory will usually be a subdirectory of your home directory, and have it (here '/mnt/bill') at the start of it.

The **echo** command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions will yield.

The **history** command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called **prompt.** By placing a '!' character in its value the shell will there substitute the index of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

```
set prompt='\! % '
```

Note that the '!' character had to be escaped here even within '' characters.

The **logout** command can be used to terminate a login shell which has **ignoreeof** set.

The **repeat** command can be used to repeat a command several times. Thus to make 5 copies of the file **one** in the file **five** you could do

```
repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

will set the value of the environment variable TERM to 'adm3a'. A user program **printenv** exists which will print out the environment. It might then show:

```
% printenv
HOME  /usr/bill
SHELL /bin/csh
TERM  adm3a
%
```

The **source** command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the **.cshrc** file which you wish to take effect before the next time you login.

The **time** command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp five five.save
0.0u 0.3s 0:01 26%
% time wc five.save
  1200   6300   37650 five.save
1.2u 0.5s 0:03 55%
%
```

indicates that the **cp** command used less that 1/10th of a second of user time and only 3/10th of a second of system time in copying the file 'five' to 'five.save'. The command word count **wc** on the other hand used 1.2 seconds of user time and 0.5 seconds of system time in 3 seconds of elapsed time in counting the number of words, character and lines in 'five.save'. The percentage '55%' indicates that over this period of 3 seconds, our command **wc** used an average of 55 percent of the available CPU cycles of the machine. This is a very high percentage and indicates that the system is lightly loaded.

The **unalias** and **unset** commands can be used to remove aliases and variable definitions from the shell.

The **wait** command can be used after starting processes with '&' to quickly see if they have finished. If the shell responds immediately with another prompt, they have. Otherwise you can wait for the shell to prompt at which point they will have finished, or interrupt the shell by sending a RUB or DELETE character. If the shell is interrupted, it will print the names and numbers of the processes it knows to be unfinished. Thus:

```
% nroff paper | lpr &
2450
2451
% wait
  2451  lpr
  2450  nroff
wait: Interrupted.
%
```

You can check again later by doing another **wait,** or see which commands are still running by doing a **ps.** As **time** will show you, **ps** is fairly expensive. It is thus counterproductive to run many **ps** commands to see how a background process is doing.†

† If you do you are usurping with these **ps** commands the processor time the job needs to finish, thereby delaying its completion!

If you run a background process and decide you want to stop it for whatever reason you must use the **kill** program. You must use the number of the processes you wish to kill. Thus to stop the **nroff** in the above pipeline you would do

```
% kill 2450
% wait
2450: nroff: Terminated.
%
```

Here the shell printed a diagnostic that we terminated 'nroff' only after we did a **wait.** If we want the shell to discover the termination of all processes it has created we must, in general, use **wait.**

4.3.7 What Else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the **foreach** built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use VENIX a lot you you should look through the rest of this document and the shell manual pages to become familiar with the other facilities which are available to you.

4.4 SHELL CONTROL STRUCTURES AND COMMAND SCRIPTS

4.4.1 Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called **shell scripts.** We here detail those features of the shell useful to the writers of such scripts.

4.4.2 Make

It is important to first note what shell scripts are **not** useful for. There is a program called **make** which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a **makefile** which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this **makefile.** This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a **makefile** may be created which defines how different versions of the document are to be created and which options of **nroff** or **troff** are appropriate.

4.4.3 Invocation and the argv Variable

A **csh** command script may be interpreted by saying

```
% csh script ...
```

where **script** is the name of the file containing a group of **csh** commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable **argv** and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file 'script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a '/bin/csh' will automatically be invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use **csh** at your convenience.

4.4.4 A Variable Substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as **variable substitution** is done on these words. Keyed by the character '$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable **argv** to be echoed to the output of the shell script. It is an error for **argv** to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to '1' if name is **set** or to '0' if name is not **set.** It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable **name.** Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

$argv[1]

gives the first component of **argv** or in the example above 'a'. Similarly

$argv[$#argv]

would give 'c', and

$argv[1−2]

would give 'a b'. Other notations useful in shell scripts are

$*n*

where **n** is an integer as a shorthand for

$argv[*n***]**

the **nth** parameter and

$*

which is a shorthand for

$argv

The form

$$

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names.

One minor difference between '$*n*' and '$argv[*n*]' should be noted here. The form '$argv[*n*]' will yield an error if **n** is not in the range '1−$#argv' while '$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form 'n−'; if there are less than **n** components of the given variable then no words are substituted. A range of the form 'm−n' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

4.4.5 Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '||' implement the boolean and/or operations.

The shell also allows file enquiries of the form

−? filename

where '?' is replace by a number of single characters. For instance the expression primitive

−e filename

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '$status' examined in the next command. Since '$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

4.4.6 Sample Shell Script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i:r.c != $i) continue   # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t             # to set $status
        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script makes use of the **foreach** command, which causes the shell to execute the commands between the **foreach** and the matching **end** for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop we may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable **noglob** here to prevent filename expansion of the members of **argv.** This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords here is **not** flexible due to the current implementation of the shell.†

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
    if ( expression ) \
            command
```

Here we have escaped the newline for the sake of appearance, and the '\' must **immediately**. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to **immediately** precede the end-of-line.

The more general **if** statements above also admit a sequence of **else−if** pairs followed by a single **else** and an **endif,** e.g.:

† The following two formats are not currently acceptable to the shell:

```
if ( expression )               # Won't work!
then
        command
        ...
endif
```

and

```
if ( expression ) then command endif                # Won't work
if ( expression ) then
        commands
else if (expression ) then
        commands
...

else
        commands
endif
```

Another important mechanism used in shell scripts is ':' modifiers. We can use the modifier ':r' here to extract a root of a filename. Thus if the variable **i** has the value 'foo.bar' then

```
% echo $i $i:r
foo.bar foo
%
```

shows how the ':r' modifier strips off the trailing '.bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the **csh** manual pages in the programmers manual. It is also possible to use the **command substitution** mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ':' modification mechanism.# Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a

---

# It is also important to note that the current implementation of the shell limits the number of ':' modifiers on a '$' substitution to 1. Thus

```
% echo $i $i:h:t /a/b/c /a/b:t %
```

does not do what one would expect.

'#' are discarded by the shell. This character can be quoted using '" or '\' to place it in an argument word.

4.4.7 Other Control Structures

The shell also has control structures **while** and **switch** similar to those of C. These take the forms

```
while ( expression )
        commands
end
```

and

```
switch ( word )

case str1:
        commands
        breaksw

...
case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the manual section for **csh.** C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in **csh** scripts is to use **break** rather than **breaksw** in switches.

Finally, **csh** allows a **goto** statement, with labels looking like they do in C, i.e.:

```
loop:
        commands
        goto loop
```

4.4.8 Supplying Input to Commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This it is different from previous shells running under VENIX. It allowing shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/↑[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the **ed** command is to come from the text in the shell script file up to the next line consisting of exactly ''EOF''. The fact that the 'EOF' is enclosed in '' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,$' in our editor script we needed to insure that this '$' was not variable substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

```
1,\$s/↑[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

4.4.9 Catching Interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where **label** is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do a **exit** command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

4.4.10 What Else?

There are other features of the shell useful to writers of shell procedures. The **verbose** and **echo** options and the related **−v** and **−x** command line options can be used to help trace the actions of the shell. The **−n** option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that **csh** will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the '/bin/sh' on your system may well defer to 'csh' to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using '"' which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as ''' does.

4.5 MISCELLANEOUS SHELL MECHANISMS

4.5.1 Loops at the Terminal; Variables as Vectors

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For example, if three shells '/bin/sh', '/bin/nsh', and '/bin/csh' were in use and you wanted to count the number of persons using each you would issue the commands:

```
% grep −c csh$ /etc/passwd
27
% grep −c nsh$ /etc/passwd
128
% grep −c −v sh$ /etc/passwd
430
%
```

Since these commands are very similar you can use **foreach** to do this more easily.

```
% foreach i ('sh$' 'csh$' '−v sh$')
? grep −c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with '?' when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The **set** command here gave the variable **a** a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within '`' characters is converted by the shell to a list of words. You can also place the '`' quoted string within '"' characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier ':x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.5.2 Braces { ... } in Argument Expansion

Another form of filename expansion, alluded to before involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories 'hdrs', 'retrofit' and 'csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown bin /usr/{bin/{ex,edit},lib/{ex1.1strings,how_ex}}
```

4.5.3 Command Substitution

A command enclosed in '`' characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable **pwd** or to do

```
ex `grep -l TRACE *.c`
```

to run the editor **ex** suppling as arguments those files whose names end in '.c' which have the string 'TRACE' in them.*

4.5.4 Other Details Not Covered Here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in the *User Reference Manual.*

The shell has a number of command line option flags mostly of use in writing VENIX programs, and debugging shell scripts. See the **sh**(1) section in the *User Reference Manual* for a list of these options.

* Command expansion also occurs in input redirected with '<<' and within '"' quotations. Refer to the shell manual section for full details.

APPENDIX

Special Characters

The following table lists the special characters of **csh** and the VENIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the **csh** manual section for a complete list.

Syntactic metacharacters

;	**4.3.4**	**separates commands to be executed sequentially**
|	**4.2.5**	**separates commands in a pipeline**
( )	**4.3.2**	**brackets expressions and variable values**
&	**4.3.5**	**follows commands to be executed without waiting for completion**

Filename metacharacters

/	**4.2.6**	**separates components of a file's pathname**
.	**4.2.6**	**separates root parts of a file name from extensions**
?	**4.2.6**	**expansion character matching any single character**
*****	**4.2.6**	**expansion character matching any sequence of characters**
[ ]	**4.2.6**	**expansion sequence matching any single character from a set**
~	**4.2.6**	**used at the beginning of a filename to indicate home directories**
{ }	**4.5.2**	**used to specify groups of arguments with common parts**

Quotation metacharacters

\	4.2.7	**prevents meta-meaning of following single character**
′	4.2.7	**prevents meta-meaning of a group of characters**
″	4.5.3	**like ′, but allows variable and command expansion**

Input/output metacharacters

<	4.2.3	**indicates redirected input**
>	4.2.5	**indicates redirected output**

Expansion/substitution metacharacters

$	**4.4.4**	**indicates variable substitution**
!	**4.3.3**	**indicates history substitution**
:	**4.4.6**	**precedes substitution modifiers**
↑	**4.3.3**	**used in special forms of history substitution**
`	**4.5.3**	**indicates command substitution**

Other metacharacters

#	**4.4.6**	**begins a shell comment**
–	**4.2.2**	**prefixes option (flag) arguments to commands**

GLOSSARY

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form **pr**(1) indicate that the command **pr** is in the *User Reference Manual* in section 1. References of the form (4.3.5) indicate that more information can be found in section 4.3.5 of this chapter.

Your current directory has the name '.' as well as the name printed by the command **pwd.** The current directory '.' is usually the first component of the search path contained in the variable **path,** thus commands which are in '.' are found first (4.3.2). The character '.' is also used in separating components of filenames (4.2.6). The character '.' at the beginning of a component of a pathname is treated specially and not matched by the filename expansion metacharacters '?', '*', and '[' ']' pairs (4.2.6).

Each directory has a file '..' in it which is a reference to its **parent** directory. After changing into the directory with **chdir,** i.e.

```
chdir paper
```

you can return to the parent directory by doing

```
chdir ..
```

The current directory is printed by **pwd** (4.3.6).

alias	An **alias** specifies a shorter or different name for a VENIX command, or a transformation on a command to be performed in the shell. The shell has a command **alias** which establishes aliases and can print their current values. The command **unalias** is used to remove aliases (4.3.6).
argument	Commands in VENIX receive a list of argument words. Thus the command **echo a b c** consists of a command name 'echo' and three argument words 'a', 'b' and 'c' (4.2.1).
argv	The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called **argv** within the shell. This name is taken from the conventional name in the C programming language (4.4.4).
background	Commands started without waiting for them to complete are called **background** commands (4.2.5).
bin	A directory containing binaries of programs and shell scripts to be executed is typically called a 'bin' directory. The standard system 'bin' directories are '/bin' containing the most heavily used commands and '/usr/bin' which contains most other user programs. Other binaries are contained in directories such as '/usr/new' where new programs are placed. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a component of the variable **path.**
break	**Break** is a built-in command used to exit from loops within the control structure of the shell (4.4.6).
builtin	A command executed directly by the shell is called a **builtin** command. Most commands in VENIX are not built into the shell, but rather exist as files in 'bin' directories. These commands are accessible because the directories in which they reside are named in the **path** variable.

case	A **case** command is used as a label in a **switch** statement in the shells control structure, similar to that of the language C. Details are given in the shells documentation 'csh (NEW)' (4.4.7).
cat	The **cat** program catenates a list of specified files on the standard output. It is usually used to look at the contents of a single file on the terminal, to 'cat a file' (4.2.8, 4.3.3).
cd	The **cd** command is used to change the working directory. With no arguments, **cd** changes your working directory to be your **home** directory (4.3.3) (4.3.6).
chdir	The **chdir** command is a synonym for **cd**. **Cd** is usually used because it is easier to type.
chsh	The **chsh** command is used to change the shell which you use on VENIX. By default, you use an older 'standard' version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing **chsh your-login-name /bin/csh** Thus I would do **chsh bill /bin/csh** It is only necessary to do this once. The next time you log in to VENIX after doing this command, you will be using **csh** rather than the shell in '/bin/sh' (4.2.9).
cmp	**Cmp** is a program which compares files. It is usually used on binary files, or to see if two files are identical (4.4.6). For comparing text files the program **diff,** described in 'diff (1)' is used.
command	A function performed by the system, either by the shell (a builtin command) or by a program residing in a file in a directory within the VENIX system is called a **command** (4.2.1).

command substitution	The replacement of a command enclosed in ‘`’ characters by the text output by that command is called **command substitution** (4.4.6, 4.4.3).
component	A part of a **pathname** between ‘/’ characters is called a **component** of that pathname. A **variable** which has multiple strings as value is said to have several **components,** each string is a **component** of the variable.
continue	A builtin command which causes execution of the enclosing **foreach** or **while** loop to cycle prematurely. Similar to the **continue** command in the programming language C (4.4.6).
core dump	When a program terminates abnormally, the system places an image of its current state in a file named ‘core’. This ‘core dump’ can be examined with the system debuggers ‘db (1)’ and ‘cdb (1)’ in order to determine what went wrong with the program (4.2.8). If the shell produces a message of the form: **commandname: Illegal instruction -- Core dumped** (where ‘Illegal instruction’ is only one of several possible messages) you should report this to the author of the program and save the ‘core’ file. If this was a system program you should report this with the **trouble** command ‘trouble (1)’.
cp	The **cp** (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used VENIX commands (4.3.6).
.cshrc	The file **.cshrc** in your **home** directory is read by each shell as it begins execution. It is usually used to change the setting of the variable **path** and to set **alias** parameters which are to take effect globally (4.3.1).
CTRL-C	The CTRL-C or key on the terminal is used to generate an INTERRUPT signal in VENIX which stops the execution of most programs (4.3.6).

date	The **date** command prints the current date and time (4.2.3).
debugging	**Debugging** is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell debugging (4.5.4).
default	The label **default:** is used within shell **switch** statements, as it is in the C language to label the code to be executed if none of the **case** labels matches the value switched on (4.4.7).
detached	A command run without waiting for it to complete is said to be detached (4.3.5).
diagnostic	An error message produced by a program is often referred to as a **diagnostic.** Most error messages are not written to the standard output, since that is often directed away from the terminal (4.2.3, 4.2.5). Error messages are instead written to the **diagnostic output** which may be directed away from the terminal, but usually is not. Thus diagnostics will usually appear on the terminal (4.3.5).
directory	A structure which contains files. At any time you are in one particular directory whose names can be printed by the command **pwd**. The **chdir** command will change you to another directory, and make the files in that directory visible. The directory in which you are when you first login is your **home** directory (4.2.1, 4.2.6).
echo	The **echo** command prints its arguments (4.2.6, 4.3.6, 4.4.6, 4.4.10).
else	The **else** command is part of the 'if-then-else-endif' control command construct (4.4.6).
EOF	An **end-of-file** is generated by the terminal by a CTRL-D , and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a **pipe** receive an end-of-file when the command sending them input completes. Most commands terminate when they receive an end-of-file. The shell has an option to ignore end-of-file from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (4.2.1, 4.2.8, 4.4.8).

escape A character \ used to prevent the special meaning of a metacharacter is said to **escape** the character from its special meaning. Thus

```
echo \*
```

will echo the character '*' while just

```
echo *
```

will echo the names of the file in the current directory. In this example, \ **escapes** '*' (4.2.7). There is also a non-printing character called **escape,** usually labeled ESC or ALTMODE on terminal keyboards. Some VENIX systems use this character to indicate that output is to be suspended. Other systems use control-s.

/etc/passwd This file contains information about the accounts currently on the system. If consists of a line for each account with fields separated by ':' characters (4.3.3). You can look at this file by saying

```
cat /etc/passwd
```

The command **grep** is often used to search for information in this file. See 'passwd (5)' and 'grep (1)' for more details.

exit The **exit** command is used to force termination of a shell script, and is built into the shell (4.4.9).

exit status A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its **exit status,** a status of zero being considered 'normal termination'. The **exit** command can be used to force a shell command script to give a non-zero exit status (4.4.5).

expansion The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of **expansion.** Thus the replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. Expansions are also referred to as **substitutions** (4.2.6, 4.4.4, 4.4.2).

expressions Expressions are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell expressions are those of the language C (4.4.5).

extension Filenames often consist of a **root** name and an **extension** separated by the character '.'. By convention, groups of related files often share the same root name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '−me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (4.2.6).

filename Each file in VENIX has a name consisting of up to 14 characters and not including the character '/' which is used in **pathname** building. Most file names do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the root portion of the filename from an extension (4.2.6).

filename expansion
Filename expansion uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. Using filename expansion it is easy to name all the files in the current directory, or all files which have a common root name. Other filename expansion mechanisms use the metacharacter '~' and allow files in other users directories to be named easily (4.2.6, 4.4.2).

flag Many VENIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as **flag** options, and by convention consists of one or more letters preceded by the character '−' (4.2.2). Thus the **ls** list file commands has an option '−s' to list the sizes of files. This is specified

	ls −s
foreach	The **foreach** command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (4.4.6, 4.4.1).
getty	The **getty** program is part of the system which determines the speed at which your terminal is to run when you first log in. It types the initial system banner and 'login:'. When no one is logged in on a terminal a **ps** command shows a command of the form '− 7' where '7' here is often some other single letter or digit. This '7' is an option to the **getty** command, indicating the type of port which it is running on. If you see a **getty** command running on a terminal in the output of **ps** you know that no one is logged in on that terminal (4.3.3).
goto	The shell has a command **goto** used in shell scripts to transfer control to a given label (4.4.7).
grep	The **grep** command searches through a list of argument files for a specified string. Thus **grep bill /etc/passwd** will print each line in the file '/etc/passwd' which contains the string 'bill'. Actually, **grep** scans for **regular expressions** in the sense of the editors **ed**(1) and **ex**(1) (4.3.3). **Grep** stands for 'globally find regular expression and print.'
hangup	When you hangup a phone line, a HANGUP signal is sent to all running processes on your terminal, causing them to terminate execution prematurely. If you wish to start commands to run after you log off a dialup you must use the command **nohup** (4.3.6).
head	The **head** command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes is useful to run **head** with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (4.2.5, 4.3.3).

history	The **history** mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a **history list** where these commands are kept, and a **history** variable which controls how large this list is (4.2.7, 4.3.6).
home directory	Each user has a home directory, which is given in your entry in the password file, **/etc/passwd.** This is the directory which you are placed in when you first log in. The **cd** or **chdir** command with no arguments takes you back to this directory, whose name is recorded in the shell variable **home.** You can also access the home directories of other users in forming filenames using a file expansion notation and the character ‘~’ (4.2.6).
if	A conditional command within the shell, the **if** command is used in shell command scripts to make decisions about what course of action to take next (4.4.6).
ignoreeof	Normally, your shell will exit, printing ‘logout’ if you type a control-d at a prompt of ‘% ’. This is the way you usually log off the system. You can **set** the **ignoreeof** variable if you wish in your **.login** file and then use the command **logout** to logout. This is useful if you sometimes accidentally type too many CTRL-D characters, logging yourself off. If the system is slow, this can waste much time, as it may take a long time to log in again (4.3.2, 4.3.6).
input	Many commands on VENIX take information from the terminal or from files which they then act on. This information is called **input.** Commands normally read for input from their **standard input** which is, by default, the terminal. This standard input can be redirected from a file using a shell metanotation with the character ‘<’. Many commands will also read from a file specified as argument. Commands placed in pipelines will read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a file name to use as standard input. Special

mechanisms exist for suppling input to commands in shell scripts (4.2.2, 4.2.6, 4.4.8).

interrupt An **interrupt** is a signal to a program that is generated by hitting the CTRL-C It causes most programs to stop execution. Certain programs such as the shell and the editors handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to interrupts. The shell often wakes up when you hit interrupt because many commands die when they receive an interrupt (4.2.8, 4.3.6, 4.4.9).

kill A program which terminates processes run without waiting for them to complete. (4.3.6)

.login The file **.login** in your **home** directory is read by the shell each time you log in to VENIX and the commands there are executed. There are a number of commands which are usefully placed here especially **tset** commands and **set** commands to the shell itself (4.3.1).

logout The **logout** command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an end-of-file, but if you have set **ignoreeof** in you **.login** file then this will not work and you must use **logout** to log off the VENIX system (4.3.2).

.logout When you log off of VENIX the shell will execute commands from the file **.logout** in your **home** directory after it prints 'logout'.

lpr The command **lpr** is the line printer daemon. The standard input of **lpr** is spooled and printed on the VENIX line printer. You can also give **lpr** a list of filenames as arguments to be printed. It is most common to use **lpr** as the last component of a **pipeline** (4.3.3).

ls The **ls** list files command is one of the most commonly used VENIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful **flag** arguments, and can also be given the names of

	directories as arguments, in which case it lists the names of the files in these directories (4.2.2).
mail	The **mail** program is used to send and receive messages from other VENIX users (4.2.1, 4.3.2).
make	The **make** command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways **make** is easier to use, and more helpful than shell command scripts (4.4.2).
makefile	The file containing command for **make** is called 'makefile' (4.4.2).
manual	is often referred to as the *User Reference Manual* and the *Programmer Reference Manual.* It contains a number of sections and a description of each VENIX program.
metacharacter	Many characters which are neither letters nor digits have special meaning either to the shell or to VENIX. These characters are called **metacharacters.** If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be **quoted.** An example of a metacharacter is the character '>' which is used to indicate placement of output into a file. For the purposes of the **history** mechanism, most unquoted metacharacters form separate words (4.2.4). The appendix to this user's manual lists the metacharacters in groups by their function.
mkdir	The **mkdir** command is used to create a new directory (4.3.6).
modifier	Substitutions with the history mechanism, keyed by the character '!' or of variables using the metacharacter '$' are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the modifier itself. The **command substitution** mechanism can also be used to perform modification in a similar way, but this notation is less clear (4.4.6).

noclobber	The shell has a variable **noclobber** which may be set in the file **.login** to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (4.3.2, 4.3.5).
nohup	A shell command used to allow background commands to run to completion even if you log off a dialup before they complete (4.3.5).
nroff	The standard text formatter on VENIX is the program **nroff.** Using **nroff** and one of the available **macro** packages for it, it is possible to have documents automatically formatted and to prepare them for phototypesetting using the typesetter program **troff** (4.4.2).
onintr	The **onintr** command is built into the shell and is used to control the action of a shell command script when an interrupt signal is received (4.4.9).
output	Many commands in VENIX result in some lines of text which are called their **output.** This output is usually placed on what is known as the **standard output** which is normally connected to the users terminal. The shell has a syntax using the metacharacter '>' for redirecting the standard output of a command to a file (4.2.3). Using the **pipe** mechanism and the metacharacter '|' it is also possible for the standard output of one command to become the standard input of another command (4.2.5). Certain commands such as the line printer daemon **lpr** do not place their results on the standard output but rather in more useful places such as on the line printer (4.3.3). Similarly the **write** command places its output on another users terminal rather than its standard output (4.3.3). Commands also have a **diagnostic output** where they write their error messages. Normally these go to the terminal even if the standard output has been sent to a file or another command, but it is possible to direct error diagnostics along with standard output using a special metanotation (4.3.5).
path	The shell has a variable **path** which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the

command as it can do it internally. If the command is not builtin, then the shell searches for a file with the name given in each of the directories in the **path** variable, left to right. Since the normal definition of the **path** variable is

```
path    (. /bin /usr/bin)
```

the shell normally looks in the current directory, and then in the standard system directories '/bin' and '/usr/bin' for the named command (4.3.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' bits set. This is normally true because a command of the form

```
chmod 755 script
```

was executed to turn these execute bits on (4.4.3).

pathname	A list of names, separated by '/' characters forms a **pathname.** Each **component,** between successive '/' characters, names a directory in which the next component file resides. Pathnames which begin with the character '/' are interpreted relative to the **root** directory in the filesystem. Other pathnames are interpreted relative to the current directory as reported by **pwd.** The last component of a pathname may name a directory, but usually names a file.
pipeline	A group of commands which are connected together, the standard output of each connected to the standard input of the next is called a **pipeline.** The **pipe** mechanism used to connect these commands is indicated by the shell metacharacter '|' (4.2.5, 4.3.3).
pr	The **pr** command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (4.3.3).

process	A instance of a running program is called a process (4.3.6). The numbers used by **kill** and printed by **wait** are unique numbers generated for these processes by VENIX. They are useful in **kill** commands which can be used to stop background processes (4.3.6).
program	Usually synonymous with **command;** a binary file or shell command script which performs a useful function is often called a program.
prompt	Many programs will print a prompt on the terminal when they expect input. Thus the editor ‘ex (NEW)’ will print a ‘:’ when it expects input. The shell prompts for input with ‘% ’ and occasionally with ‘?’ when reading commands from the terminal (4.2.1). The shell has a variable **prompt** which may be set to a different value to change the shells main prompt. This is mostly used when debugging the shell (4.3.6).
ps	The **ps** command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (4.3.3, 4.3.6). Login shells, such as the **csh** you get when you login are shown as ‘–’.
pwd	The **pwd** command prints the full pathname of the current (working) directory.
quit	The **quit** signal, generated by a CTRL-\ is used to terminate programs which are behaving unreasonably. It normally produces a core image file (4.2.8).
quotation	The process by which metacharacters are prevented their special meaning, usually by using the character ‘′ in pairs, or by using the character ‘\’ is referred to as **quotation** (4.2.4).
redirection	The routing of input or output from or to a file is known as **redirection** of input or output (4.2.3).

repeat	The **repeat** command iterates another command a specified number of times (4.3.6).
script	Sequences of shell commands placed in a file are called shell command scripts. It is often possible to perform simple tasks using these scripts without writing a program in a language such as C, by using the shell to selectively run other programs (4.4.2, 4.4.3, 4.4.10).
set	The builtin **set** command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the set command the behavior of the shell can be affected (4.3.1).
setenv	Variables in the environment **environ**(5) can be changed by using the **setenv** builtin command (4.3.6).
shell	A shell is a command language interpreter. It is possible to write and run your own shell, as shells are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular shell, called **csh.**
shell script	See **script** (4.4.2, 4.4.3, 4.4.10).
sort	The **sort** program sorts a sequence of lines in ways that can be controlled by argument flags (4.2.5).
source	The **source** command causes the shell to read commands from a specified file. It is most useful for reading files such as **.cshrc** after changing them (4.3.6).
special character	See **metacharacters** and the appendix to this manual.
standard	We refer often to the **standard input** and **standard output** of commands. See **input** and **output** (4.2.3, 4.4.8).
status	A command normally returns a **status** when it finishes. By convention a **status** of zero indicates that the command succeeded. Commands may return non-zero status to indicate that some abnormal event has occurred. The shell variable **status** is set to the status returned by the last command. It is most useful in shell command scripts (4.4.5, 4.4.6).

substitution	The shell implements a number of **substitutions** where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history substitution keyed by the metacharacter '!' and variable substitution indicated by '$'. We also refer to substitutions as **expansions** (4.4.4).
switch	The **switch** command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the **switch** statement in the language C (4.4.7).
termination	When a command which is being executed finished we say it undergoes **termination** or **terminates.** Commands normally terminate when they read an end-of-file from their standard input. It is also possible to terminate commands by sending them an **interrupt** or **quit** signal (4.2.8). The **kill** program terminates specified command whose numbers are given (4.3.6).
then	The **then** command is part of the shells 'if-then-else-endif' control construct used in command scripts (4.4.6).
time	The **time** command can be used to measure the amount of CPU and real time consumed by a specified command (4.3.1, 4.3.6).
troff	The **troff** program is used to typeset documents. See also **nroff** (4.4.2).
tset	The **tset** program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a **.login** file (4.3.1).
unalias	The **unalias** command removes aliases (4.3.6).
VENIX	VENIX is an operating system on which **csh** runs. VENIX provides facilities which allow **csh** to invoke other programs such as editors and text formatters which you may wish to use.

unset	The **unset** command removes the definitions of shell variables (4.3.2, 4.3.6).
variable expansion	See **variables** and **expansion** (4.3.2, 4.4.4).
variables	Variables in **csh** hold one or more strings as value. The most common use of variables is in controlling the behavior of the shell. See **path, noclobber,** and **ignoreeof** for examples. Variables such as **argv** are also used in writing shell programs (shell command scripts) (4.3.2).
verbose	The **verbose** shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The **verbose** variable is set by the shells **−v** command line option (4.4.10).
wait	The builtin command **wait** causes the shell to pause, and not prompt, until all commands run in the background have terminated (4.3.6).
where	The **where** command shows where the users named as arguments are logged into the system (4.3.3).
while	The **while** builtin control construct is used in shell command scripts (4.4.7).
word	A sequence of characters which forms an argument to a command is called a **word.** Many characters which are neither letters, digits, ‘−’, ‘.’ or ‘/’ form words all by themselves even if they are not surrounded by blanks. Any sequence of character may be made into a word by surrounding it with ‘’’ characters except for the characters ‘’’ and ‘!’ which require special treatment (4.2.1, 4.2.6). This process of placing special characters in words without their special meaning is called **quoting.**
working directory	At an given time you are in one particular directory, called your working directory. This directories name is printed by the **pwd** command and the files listed by **ls** are the ones in this directory. You can change working directories using **chdir.**

write The **write** command is used to communicate with other users who are logged in to VENIX (4.3.3).

Printed in U.S.A.

AA-BM32A-TH