PLEXUS

Plexus Sys3 UNIX Programmer's Manual -- vol 2C

98-05047.2 May 25, 1984

PLEXUS

Sys3 UNIX vol 2C Programmer's Manual

Plexus Sys3 UNIX Programmer's Manual -- vol 2C

98-05047.2 May 25, 1984

PLEXUS COMPUTERS, INC.

2230 Martin Ave.

Santa Clara, CA 95050

408/988-1755

Printed in the United States of America

PREFACE

This manual contains a collection of documents that describe specific aspects of the UNIX* operating system. These include descriptions of special utilities developed at the University of California at Berkeley including the C shell, *edit*, *ex*, and *vi*.

Additional documents describing the operating system, document preparation tools, and language tools may be found in the *Plexus Sys3 UNIX Programmer's Manual -- vol 2A* (Plexus publication #98-05036). Documents describing programming, language, administrative and maintenance tools are collected in the *Plexus Sys3 UNIX Programmer's Manual -- vol 2B* (Plexus publication #98-05037).

Both these volumes (2A and 2B) should be used as supplementary documents for the *Plexus Sys3 UNIX Programmer's Manual -- vol 1A* (Plexus publication #98-05045), and *Plexus Sys3 UNIX Programmer's Manual -- vol 1B* (Plexus publication #98-05046), the basic reference manual for the operating system.

Comments
Please address all comments concerning this manual to:

Plexus Computers, Inc.
Technical Publications Dept.
2230 Martin Ave.
Santa Clara, CA 95050
408/988-1755

Revision History
Plexus publication number 98-05047.1 was the first edition.

This edition (98-05047.2) is typeset, and corrects some minor errors.

---

* UNIX is a trademark of AT&T Bell Laboratories. Plexus Computers, Inc. is licensed to distribute UNIX under the authority of AT&T.

An Introduction to the C Shell

ABSTRACT

Csh is a new command language interpreter for UNIX systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells that make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes capabilities that you can explore after you have become more acquainted with the shell. Later sections introduce features that are useful, but not necessary for all users of the shell.

Back matter includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

May 17, 1984

An Introduction to the C Shell

Introduction

A *shell* is a command language interpreter. *Csh* is the name of one particularly useful command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs.

This document has four sections. The first two describe how to form command lines to the shell; these sections are written for people who will use the shell from a terminal, but will not necessarily write shell scripts. Section 1 describes the basics; Section 2 details more advanced features. The third section is directed toward those who wish to use the programming facilities of the shell to write shell scripts. The fourth section describes miscellaneous and less used features of the shell.

In addition to this document, you need access to the *Plexus Sys3 UNIX Programmer's Manual -- vol 1*. The *csh* entry in that manual provides a full description of all features of the shell and is a final reference for questions about the shell.

Many words in this document are shown in *italics.* These are important words: names of commands, and words that have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what a word means, you should look for it in the glossary.

Acknowledgements

This document is based on *An introduction to the C shell* by William Joy.

1. USING THE SHELL FROM THE TERMINAL

1.1. The Basic Notion of Commands

A *shell* in UNIX acts mostly as a medium through which other *commands* are invoked. While it has a set of *built-in* commands, which it performs directly, most useful commands are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system expect a list of strings or *words* as arguments. Thus 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. It looks in a number of *directories* for a file with the name *mail* and expects the file called "mail" to contain the *mail* program.

The rest of the words of the command are given to the command itself to execute. In this case the word *bill* is also specified; this is interpreted by the *mail* program to be the name of a user to whom mail is to be sent.

For example, Chris can send mail to Bill 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?
        Chris
%
```

Here Chris 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 the message. The characters '% ' (per cent sign followed by space) were printed before and after the *mail* command by the shell to indicate that the shell was awaiting input.

After typing the '% ' prompt, the shell reads command input from the terminal. Chris 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 the terminal until Chris signalled an end-of-file, after which the shell noticed that *mail* had completed. It signaled Chris that it was ready to read from the terminal again by printing another '% ' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, the shell 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.

1.2. Flag Arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names, some arguments specify optional capabilities of commands. By convention, such arguments begin with the character '-'. Thus the command

```
ls
```

produces 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 1024 characters. The

manual page for each command in the *Plexus Sys3 UNIX Programmer's Manual* gives the available options for each command. The *ls* command has many 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 that are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard-to-remember options.

1.3. Output to Files

The concepts of *standard input* and *standard output* are very important in UNIX. The default for both is the terminal; this means that unless you tell UNIX otherwise, UNIX expects to receive input for its commands from the terminal and send output of commands to the terminal. But often you want to read input from or write output to *files* rather than simply taking input and output from the terminal. The shell provides simple ways to accomplish this.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

prints 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 such that its standard output is the file 'now' rather than our terminal. Thus this command places the current date and time in the file 'now'. Note that the *date* command is unaware that its output is going to a file rather than to our terminal. *Date* sends its results to standard output, however that standard output is currently defined--terminal or file or other device or whatever. The *shell* performed this *redirection* before the command began executing.

Note also that 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 C-shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in section 2.2.

1.4. Metacharacters in the Shell

The shell has a large number of special characters (like '>') that indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters that 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 that contain *metacharacters* and to thus work without constantly worrying about whether certain characters are metacharacters.

Note that the C-shell is only reading input when it has prompted with '% '. Thus metacharacters normally have effect only then. So, for example, we need not worry about placing shell metacharacters in a letter we are sending via *mail.*

1.5. Input from Files; Pipelines

We learned above how to route the standard output of a command to a file. We can also route the standard input of a command from a file. This is not often necessary, however, since most commands will read from a file name given as argument. For example, we can give the command

```
sort < data
```

to run the *sort* command with standard input from the file 'data'. But we would more likely say

```
sort data
```

and let 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.

We can even 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.* This is an extremely useful feature. For instance, the command

```
ls -s
```

normally produces a list of the files in our directory with the size, in 1024-byte blocks, of each. If we are interested in learning which of our files is largest, we may wish to have this list sorted by size rather than by name--*ls* by default sorts by name. We could investigate the many options of *ls* to see if one lists in order of size, but we would eventually discover that no such option exists. Instead the shell lets us 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 of *its* standard input. 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 functions as the input of the next. The leftmost command in a pipeline normally takes its standard input from the terminal and the rightmost places 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 illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands need the names of files as arguments. UNIX 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 that 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* or *cd* change directory command.

Most filenames consist of a number of alphanumeric characters and '.'s. In fact, all printing characters except '/' may appear in filenames. However, non-alphabetic charcters usually do not belong 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 *basename.* 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 that begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names that match are sorted into the argument list to the command alphabetically. Thus the command

echo prog.*

echoes the names

prog.c prog.errs prog.o prog.output

Note that the names are in lexicographic order here, different from the way we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as an 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 ? ?? ???

echoes (i.e., writes on standard output) 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.

Note 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.

Also note 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 user's login name. For instance the word '~bill' would map to the pathname '/mnt/bill' if the home directory for 'bill' were 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 user's 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

i.e., the copy command interprets this 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

A mechanism using the characters '{' and '}' abbreviates a set of words that have common parts but cannot be abbreviated by the above mechanisms because they are not files, or are the names of files that do not yet exist. This mechanism will be described much later, in section 4.2, as it is used much less frequently.

1.7. 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 *

does not echo the character '*'. It either echoes a sorted list of filenames in the current directory, or prints the message 'No match' if there are no files in the current directory.

The recommended mechanism for placing characters that are neither numbers, digits, '/', '.' or '-' in an argument word to a command is to enclose it with single quotation characters ''', e.g.

echo '*'

The *history* mechanism of the shell uses the special character '!', so '!' 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 that is an argument to a shell command.

1.8. Terminating Commands

When you are running a command from the shell and the shell is 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 /usr/man/docs/csh/csh
```

the system prints this document on your terminal. This will continue for several minutes unless you stop it. You can send an INTERRUPT signal to the *cat* command by hitting the DEL or RUBOUT 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 is *cat.* Since *cat* does not take any precautions to catch this signal the INTERRUPT causes it to terminate. The shell notices that *cat* has died and prompts you again with '% '. If you hit INTERRUPT again, the shell just repeats its prompt, since it catches INTERRUPT signals but continues to execute commands anyway. If the shell went away like *cat,* this would log you out.

Many other programs terminate 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 an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing too many control-d's can accidentally log you off, the shell has a mechanism for preventing this. This *ignoreeof* option is discussed in section 2.2.

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 terminates without our typing a control-d. This is because it read to the end-of-file of our file 'prepared.text'. 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 that are not fully debugged, then you may want to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, generated by a control-\. This usually provokes 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 2.6), then these commands ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* program. See section 2.6 for an example.

1.9. What Now?

We have so far seen a number of mechanisms of the shell and learned something about the way in which it operates. The remaining sections will go further into the internals of the shell, but you will surely want to try using the shell before you go any further. To get the C shell as your login shell, your entry in the file */etc/passwd* must be modified. */etc/passwd* is writable only by the superuser, so you may have to get your system administrator to perform the change. The last field of each line in */etc/passwd* is the shell field. No entry in this field means this user

has the regular shell, */bin/sh*. This field of your line must be modified to read */usr/plx/csh*. You will get the C shell the next time you log in. It's a good idea to have your .login and .cshrc files set up appropriately before you try the C shell, so your terminal will behave properly.

You can also invoke a csh by typing '/usr/plx/csh'. This gives you a temporary C shell, and you may return to your login shell by typing control-d.

Much of the discussion in this manual so far is applicable to '/bin/sh' as well as the C shell. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

2. DETAILS ON THE SHELL

2.1. 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 that you may create during your terminal session 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 you wish to do each time you login to the UNIX system. A *.login* file might look something like:

```
tset -d adm3a
set history=20
set time=3
```

This file contains four commands to be executed by UNIX each time the user logs in. The first is a *tset* command, which informs the system that this user usually dials in on a Lear-Siegler ADM-3A terminal. 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 that 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 the user wishes to repeat or rerun modified versions of previous commands. This mechanism involves a certain overhead, so the shell does not set this variable by default. 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 prints 'logout' and executes commands from the file '.logout' in your home directory. After that the shell dies and UNIX logs you off the system. If the system is not going down, you receive a new login message. In any case, after the 'logout' message, the shell from which the end-of-file was received is doomed and takes no further input from the terminal.

2.2. 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. This command has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may store values that are to be reintroduced into commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those that the shell itself refers to. By changing the values of these variables, you 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 that 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 useful programs that are not part of standard UNIX SYSTEM III--including *csh*--live in the directory '/usr/plx'. If you want all shells that you invoke to have access to these new programs, place the command

```
set path=(. /usr/plx /bin /usr/bin)
```

in your file *.cshrc* in your 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, and 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 UNIX with *ignoreeof* set you must type

```
logout
```

This is one of several variables that 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 useful built-in shell variables are *noclobber* and *mail.* The metasyntax

```
> filename
```

which redirects the output of a command, overwrites and destroys the previous contents of the named file. In this way you may accidentally overwrite a valuable file. If you prefer that the shell not automatically 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 all right.

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 looks 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; it only delays the shell's response to you.

The use of shell variables to introduce text into commands, which is most useful in shell command scripts, will be introduced in section 2.4.

2.3. The Shell's History List

The shell can maintain a history list into which it places the words of previous commands. You can use a metanotation to reintroduce commands or words from commands in forming new commands. This mechanism can repeat previous commands or correct minor typing mistakes in commands.

Consider the following transcript:

```
% ls -l shell.proc
-rw-r--r-- 1 sandy    6506 Jan 19 20:05 shell.proc
% chmod 755 !$
chmod 755 shell.proc
```

Here we asked for a long (-l) listing of the file 'shell.proc' and were told, among other things, that it was not executable. We need to change the permissions attached to it, so we execute a *chmod* command on '!$'. '!$' is a history notation that means the last word of the last command executed, in this case 'shell.proc'. The shell performed this substitution and then echoed the command ('chmod 755 shell.proc') as it would execute it. Suppose now that we want to verify that the permission changes were made by doing another 'ls -l'. We can do

```
% !l
-rwxrwxrwx 1 sandy    6506 Jan 19 20:05 shell.proc
%
```

We repeat the *ls* -l command with the history notation '!l', which repeats the last command that began with a word of which 'l' is a prefix.

The form '!!' re-executes the last command. Another useful command form is '↑lhs↑rhs', which performs a substitution similar to that in *ed* or *ex*. Thus in this example,

```
% cat ~bill/csh/sh..c
/mnt/bill/csh/sh..c: No such file or directory
% ↑..↑.
cat ~bill/csh/sh.c
#include "sh.h"

char    *pathlist[] =    { SRCHP
%
```

we used the substitution to correct a typing mistake, and then rubbed the command out after we saw that we had found the file 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* that worked above)

```
% pr !$ | lpr
pr ~bill/csh/sh.c | lpr
%
```

More advanced forms of the history mechanism also exist. Substitutions themselves may be modified, so you can 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 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 entry in the *Plexus Sys3 UNIX Programmer's Manual.*

2.4. Aliases

The shell has an *alias* mechanism that can simplify the commands you type, supply default arguments to commands, or 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 also be obtained using shell command files, but these take place in another instance of the shell and cannot directly affect the current shell's environment. Thus commands such as *chdir,* which must be done in the current shell, may not work the way you expect.

As an example, suppose you wish to use a new version of the *mail* program. The new program is called 'Mail', and the standard mail program continues to be 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 translates this to

```
ls -s /mnt/bill
```

Thus the *alias* mechanism can provide deafult arguments and short names for commands, and can define new short commands in terms of other commands. You can also define aliases that 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 '
```

does 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 that looks up its first argument in the password file.

2.5. Detached Commands; >> and >& Redirection

A few more metanotations are useful. The metacharacter '&' may be placed after a command, or after a sequence of commands separated by ';' or '|'. This causes the shell not to 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 that 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 that is normally directed to the terminal even when the standard output is directed to a file or a pipe. You may occasionally want to direct the diagnostic output along with the standard output. For instance, you may want to redirect the output of a long-running command into a file and have a record of any error diagnostic it produces. The command

```
command >& file
```

accomplishes this. The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon *lpr.*#

---

†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.

#A command form

```
command >&! file
```

exists, and is used when *noclobber* is set and *file* already exists.

Finally, you can use the form

command >> file

to place output at the end of an existing file.†

2.6. Useful Built-in Commands

We now describe a few of the useful built-in commands of the shell.

The *alias* command described above assigns new aliases and shows 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, in this case, 'ls'.

The *cd* and *chdir* commands are equivalent; they change the working directory of the shell. An experienced UNIX user usually makes a subdirectory for each of his projects and places all files related to each project in the appropriate subdirectory. 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 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. The current directory is usually a subdirectory of your home directory. Thus, the home directory (here '/mnt/bill') path forms the prefix of the new directory name. In the example above, '/mnt/bill' forms the prefix of '/mnt/bill/newpaper'.

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 shows the contents of the history list. The numbers given with the history events can be used to refer to previous events that are difficult to refer to using the contextual mechanisms introduced above. There is also a shell variable called *prompt.* By placing a '!' character in its value the shell there substitutes 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 terminates a login shell that has *ignoreeof* set.

The *repeat* command repeats a command. Thus to make 5 copies of the file *one* in the file *five* you could do

---

†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.

```
repeat 5 cat one >> five
```

The *setenv* command sets variables in the environment. Thus

```
setenv TERM adm3a
```

sets the value of the environment variable TERM to 'adm3a'. A user program *printenv* prints out the environment. It might then show:

```
% printenv
HOME /usr/bill
SHELL /bin/csh
TERM adm3a
%
```

The *source* command forces the current shell to read commands from a file. Thus

```
source .cshrc
```

causes any changes to *.cshrc* to take effect before the next time you login.

The *time* command can time a command 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,* which counts the number of words, character and lines in 'five.save', used 1.2 seconds of user time and 0.5 seconds of system time in 3 seconds of elapsed time. 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 remove aliases and variable definitions from the shell.

The *wait* command can be used after starting processes with '&' to see quickly 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 RUBOUT or DELETE character. If the shell is interrupted, it prints 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 you must use the *kill* program. You must use the process id number(s) (PIDs) of the process(es) you wish to kill. (If you don't know, do a *ps*). 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.*

2.7. What Else?

This concludes the basic discussion of the shell for terminal users. The programming features of the shell are described in the next section. One especially useful feature 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 UNIX a lot, you should look through the rest of this document and the shell manual pages to become familiar with the other facilities available to you.

3. SHELL CONTROL STRUCTURES AND COMMAND SCRIPTS

3.1. Introduction

Rather than inputting commands one at a time to the shell, you can place commands in files and cause these commands to be read and executed. Using these command files, you can put long command sequences in motion without having to wait for each one to complete serially; arithmetic and loop control constructs are also available. An added plus is that you can use all the expansion and substitution facilities of your editor to create these files quickly. These command files are called *shell scripts.* We here detail those features of the shell useful to the writers of such scripts.

3.2. Make

Note what shell scripts are *not* useful for. The program *make* 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.* Shell scripts are less suitable for this kind of maintenance.

Similarly when working on a document, a *makefile* may be created that defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

3.3. Invocation

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 used to refer to 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 '/usr/plx/csh' is automatically invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' executes it. This allows you to convert your older shell scripts to use *csh* at your convenience.

A complication arises, however, when you run shell scripts under the C shell in UNIX SYSTEM III or Plexus Sys3. The C shell was written to be used with UNIX Version 7, the release *before* SYSTEM III, upon which Plexus Sys3 is based. Many standard Sys3 commands are actually shell scripts, and some begin with the comment character, so they look like C shell scripts to the C shell. But Sys3 contains commands not found in Version 7, so some of these shell scripts call commands the C shell has never heard of, and can't execute. Such commands fail when run under the C shell. Therefore, if you run the C shell with Sys3, you should set your environment variable SHELL to **/bin/sh**, so any shell scripts are automatically run by the Sys3 shell. If you really want the C shell to execute your shell script, you can explicitly execute it with 'csh' or put the call to 'csh' within the script.

3.4. 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, *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 causes the current value of the variable *argv* to be echoed to the output of the shell script. *Argv* may not be unset at this point.

A number of notations are provided for accessing attributes of variables. The notation

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set*. This is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

You can also access the components of a variable that has several values. Thus

```
$argv[1]
```

gives the first component of *argv* or, in the example above, 'a'. Similarly

```
$argv[$#argv]
```

would give 'c'.

Other notations useful in shell scripts are

```
$n
```

(where *n* is an integer) as a shorthand for

```
$argv[n]
```

(the *n th* parameter) and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the

system it can be used in generation of unique temporary file names.

One minor difference between '$*n*' and '$argv[*n*]' should be noted here. The form '$argv[*n*]' yields an error if *n* is not in the range '1-$#argv' while '$n' never yields an out-of-range-subscript error. This is for compatibility with the way older shells handled parameters.

Note also 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.

3.5. Expressions

Expressions in the shell must be able to be evaluated based on the values of variables; otherwise, the shell wouldn't be able to do anything interesting. 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' and 'or' operations.

The shell also allows file enquiries of the form

```
-? filename
```

where '?' is replaced by a number of single characters. For instance the expression primitive

```
-e filename
```

tells 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.

You can also 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 (with exit status non-zero). If you need more detailed information about the execution status of a command, execute it and in the next command examine the variable '$status'. Since '$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 command immediately following.

For a full list of expression components available see the manual entry for the shell.

3.6. Sample Shell Script

A sample shell script that 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 that have already been expanded or if the arguments may contain filename expansion metacharacters. You can also 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 somewhat flexible.†

The shell does have another form of the *if* statement

---

†Leaving out the 'then' altogether works, for example. The 'then' may also be put on a separate line, as follows:

```
if ( expression )              # Produces diagnostic but works!
then
        command
        ...
endif
```

but this produces a diagnostic to the effect that the command 'then' cannot be found. The following format is not currently acceptable to the shell:

```
if ( expression ) then command endif   #Won't work
```

```
if ( expression ) command
```

which can be written

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. 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.:

```
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 entry. You can also use the *command substitution* mechanism described in section 4.3 to perform modifications on strings to reenter the shell's environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive 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 '#' are discarded by the shell. This character can be quoted using '' or '\' to place it in an argument word.

3.7. 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

---

*Note also 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.

```
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 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
```

3.8. Supplying Input to Commands

Commands run from shell scripts receive by default the standard input of the shell that is running the script. Thus it is different from previous shells running under UNIX. It allows shell scripts to participate fully in pipelines, but mandates extra notation for commands that 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 not to perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' is quoted, then these substitutions are not 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.

3.9. 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 into 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'.

3.10. What Else?

Other features of the shell are useful to writers of shell procedures. The *verbose* and *echo* options and the related *-v* and *-x* command line options can help trace the actions of the shell. The *-n* option causes the shell only to read commands and not to execute them.

Note that *csh* does not execute shell scripts that do not begin with the character '#'--that is, shell scripts that do not begin with a comment.

Another quotation mechanism using '"' 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. See the manual entry for more information.

4. MISCELLANEOUS, LESS GENERALLY USEFUL, SHELL MECHANISMS

4.1. Loops at the Terminal; Variables as Vectors

You may occasionally want to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, three shells were once in use on the Cory UNIX system at Cory Hall at UC Berkeley: '/bin/sh', '/bin/nsh', and '/bin/csh'. To count the number of persons using each shell, one could 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 we 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 that 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. This prevents the lines from being split into words at blanks and tabs. A modifier ':x' can later expand each component of the variable into another variable, splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in Argument Expansion

Another form of filename expansion 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. If no other expansion mechanisms are used, the resulting filenames need not exist. This means that this mechanism can be used to generate arguments that are not filenames, but 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}}

This command changes the ownership of all the following files: /usr/bin/ex, /usr/bin/edit, /usr/lib/ex1.1strings, and /usr/lib/how_ex.

4.3. 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* supplying as arguments those files whose names end in '.c' and have the string 'TRACE' in them.*

4.4. Other Details Not Covered Here

Sometimes you may need 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 manual section.

The shell has a number of command line option flags that are mostly of use in writing UNIX programs and debugging shell scripts. See the manual entry 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 UNIX 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 entry for a complete list.

Syntactic metacharacters

;	2.4	separates commands to be executed sequentially
|	1.5	separates commands in a pipeline
( )	2.2,3.6	brackets expressions and variable values
&	2.5	follows commands to be executed without waiting for completion

Filename metacharacters

/	1.6	separates components of a file's pathname
?	1.6	expansion character matching any single character
*	1.6	expansion character matching any sequence of characters
[ ]	1.6	expansion sequence matching any single character from a set
~	1.6	used at the beginning of a filename to indicate home directories
{ }	4.2	used to specify groups of arguments with common parts

Quotation metacharacters

\	1.7	prevents meta-meaning of following single character
'	1.7	prevents meta-meaning of a group of characters
"	4.3	like ', but allows variable and command expansion

Input/output metacharacters

<	1.3	indicates redirected input
>	1.5	indicates redirected output

Expansion/substitution metacharacters

$	3.4	indicates variable substitution
!	2.3	indicates history substitution
:	3.6	precedes substitution modifiers
↑	2.3	used in special forms of history substitution
`	4.3	indicates command substitution

Other metacharacters

#	3.6	begins a shell comment
-	1.2	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 this document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the *Plexus Sys3 UNIX Programmer's Manual* in section 1. You can get an online copy of its manual page by doing

```
man 1 pr
```

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual.

. 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 that are in '.' are found first (2.2). The character '.' is also used in separating components of filenames (1.6). The character '.' at the beginning of a component of a pathname is treated specially and not matched by the filename expansion metacharacters '?', '*', and '[' ']' pairs (1.6).

.. 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* (2.6).

alias An *alias* specifies a shorter or different name for a UNIX 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 (2.6).

argument Commands in UNIX 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' (1.1).

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 (3.4).

background Commands started without waiting for them to complete are called *background* commands (1.5).

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', which contains the most heavily used commands, and '/usr/bin', which contains most other user programs. Other binaries are contained in directories such as '/usr/plx' where new and Plexus-specific programs are placed. You can place binaries in any directory. If you wish to execute them often, the directory's name 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 (3.6).

built-in A command executed directly by the shell is called a *built-in* command. Most commands in UNIX 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 shell's control structure, similar to that of the language C. Details are given in the shell's documentation 'csh (NEW)' (3.7).

cat — The *cat* program catenates a list of specified files on the standard output. It is usually invoked to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3).

cd — The *cd* command is used to change the working directory. With no arguments, *cd* changes your working directory to be your *home* directory (2.3) (2.6).

chdir — The *chdir* command is a synonym for *cd. Cd* is usually typed because it is shorter.

cmp — *Cmp* is a program that compares files. It is usually used on binary files, or to see if two files are identical (3.6). 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 built-in command) or by a program residing in a file in a directory within the UNIX system is called a *command* (1.1).

command substitution — The replacement of a command enclosed in '`' characters by the text output by that command is called *command substitution* (3.6, 4.3).

component — A part of a *pathname* between '/' characters is called a *component* of that pathname. A *variable* that has multiple strings as value is said to have several *components,* and each string is a *component* of the variable.

continue — A built-in command that causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).

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 'adb (1)' and 'sdb (1)' in order to determine what went wrong with the program (1.8). 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.

cp — The *cp* (copy) program copies the contents of one file into another file. It is one of the most commonly used UNIX commands (2.6).

.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 take effect globally (2.1).

date — The *date* command prints the current date and time (1.3).

debugging — *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables that may be used to aid in shell debugging (4.4).

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 (3.7).

DELETE — The DELETE or RUBOUT key on the terminal is used to generate an INTERRUPT signal, which stops the execution of most programs on UNIX (2.6).

detached — A command run without waiting for it to complete is said to be detached (2.5).

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 (1.3, 1.5). 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 (2.5).

directory — A structure that contains files. At any time you are 'in' one particular directory whose names can be printed by the command 'pwd'. The *chdir* command will position you 'in' another directory. The directory in which you are when you first login is your *home* directory (1.1, 1.6).

echo — The *echo* command prints its arguments (1.6, 2.6, 3.6, 3.10).

else — The *else* command is part of the 'if-then-else-endif' control command construct (3.6).

EOF — An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file that 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; this option may help you keep from logging out accidentally (1.1, 1.8, 3.8).

escape — A character \ used to prevent the special meaning of a metacharacter is called an *escape* character, because the special meaning is 'escaped'. Thus

```
echo \*
```

will echo the character '*' while just

```
echo *
```

will echo the names of the file in the current directory. In this example, \ *escapes* '*' (1.7). There is also a non-printing character called *escape,* usually labeled ESC or ALTMODE on terminal keyboards. Some UNIX 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 (2.3). 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 (3.9).

exit status — A command that discovers a problem may reflect this back to the command (such as a shell) that 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 (3.5).

expansion — The replacement of strings that 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* (1.6, 3.4, 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 (3.5).

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' (1.6).

filename — Each file in UNIX has a name consisting of up to 14 characters and not including the character '/'. This name 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 (1.6).

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 that have a common root name. Other filename expansion mechanisms use the metacharacter '~' and allow files in other users directories to be named easily (1.6, 4.2).

flag — Many UNIX commands accept arguments that 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 consist of one or more letters preceded by the character '-' (1.2). 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 (3.6, 4.1).

getty — The *getty* program is part of the system that 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 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 (2.3).

goto — The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7).

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' that contains the string 'bill'. Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed (1)' and 'ex (1)' (2.3). *Grep* stands for 'globally find regular expression and print' or, some say, 'get regular expression pattern'.

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* (2.6).

head — The *head* command prints the first few lines of one or more files. If you have files containing text you are wondering about, try running *head* with these files as arguments. This usually shows enough of what is in these files to let you decide which you are interested in (1.5, 2.3).

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 (1.7, 2.6).

home directory — Each user has a home directory, which is given in your entry in the password file, */etc/passwd.* This is the directory 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 '~' (1.6).

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 (3.6).

ignoreeof — Normally, your shell exits, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. But you can *set* the *ignoreeof* variable in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2, 2.6).

input — Many commands on UNIX take information from the terminal or from files that 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 also read from a file specified as argument. Commands placed in pipelines read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a file name to use as standard input. Special mechanisms exist for supplying input to commands in shell scripts (1.2, 1.6, 3.8).

interrupt — An *interrupt* is a signal to a program that is generated by hitting the RUBOUT or DELETE key. 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 (1.8, 2.6, 3.9).

kill — A program that terminates processes run without waiting for them to complete. (2.6)

.login — The file *.login* in your *home* directory is read by the shell each time you log in to UNIX and the commands there are executed. A number of commands are usefully placed here, especially *tset* commands and *set* commands to the shell itself (2.1).

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 your *.login* file, then this will not work and you must use *logout* to log off the UNIX system (2.2).

.logout — When you log off of UNIX 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 UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. *Lpr* is usually the last component of a *pipeline* (2.3).

ls — The *ls* list files command is one of the most commonly used UNIX 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 (1.2).

mail — The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.2).

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 (3.2).

makefile — The file containing commands for *make* is called 'makefile' (3.2).

manual — The 'manual' often referred to is the *Plexus Sys3 UNIX Programmer's Manual.* It contains a description of each UNIX program. An online version of the manual is accessible through the *man* command. Its documentation can be obtained online via

```
man man
```

metacharacter — Many characters that are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters.* If these characters must be used without their special meaning in arguments to commands, 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 (1.4). The appendix to this manual lists the metacharacters in groups by their function.

mkdir — The *mkdir* command is used to create a new directory (2.6).

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 (3.6).

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 (2.2, 2.5).

nohup — A shell command used to allow background commands to run to completion even if you log off a dialup before they complete. (2.5)

nroff — The standard text formatter on UNIX is the program *nroff.* Using *nroff* and one of the available *macro* packages for it, documents may be automatically formatted and prepared for phototypesetting using the typesetter program *troff* (3.2).

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 (3.9).

output — Many commands in UNIX produce *output.* This output is usually placed on what is known as the *standard output*, which is normally connected to the user's terminal. The shell metacharacter '>' redirects the standard output of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter '|', the standard output of one command may become the standard input of another

command (1.5). 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 (2.3). Similarly the *write* command places its output on another user's terminal rather than its standard output (2.3). 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 you may direct error diagnostics along with standard output using a special metanotation (2.5).

path — The shell has a variable *path,* which gives the names of the directories in which it searches for the commands that 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 built-in, 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 (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands are 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 (3.3).

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 that begin with the character '/' are interpreted relative to the *root* directory in the file system. Other pathnames are interpreted relative to the current directory as reported by *pwd.* The last component of a pathname may name a directory, but usually names a file.

pipeline — A group of commands connected together, the standard output of each 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 '|' (1.5, 2.3).

pr — The *pr* command prepares listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).

printenv — The *printenv* command is used on version 7 UNIX systems to print the current setting of variables in the environment (2.6).

process — A instance of a running program is called a process (2.6). The numbers used by *kill* and printed by *wait* are unique numbers generated for these processes by UNIX. They are useful in *kill* commands, which can be used to stop background processes. (2.6)

program — Usually synonymous with *command;* a binary file or shell command script that performs a useful function is often called a program.

Programmer's Manual — Also referred to as the *manual.* See the glossary entry for 'manual'.

prompt — Many programs print a prompt on the terminal when they expect input. Thus the editor 'ex' prints a ':' when it expects input. The shell prompts for input with '% ' and occasionally with '? ' when reading commands from the terminal (1.1). The shell has a variable *prompt,* which may be set to a different value to

change the shell's main prompt. This is mostly used when debugging the shell (2.6).

ps — The *ps* command shows 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 (2.3, 2.6). 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 control-\ is used to terminate programs that are behaving unreasonably. It normally produces a core image file (1.8).

quotation — The process by which metacharacters are prevented their special meaning, usually by using the character '' in pairs, or by using the character '\' (1.4).

redirection — The routing of input or output from or to a file is known as *redirection* of input or output (1.3).

repeat — The *repeat* command iterates another command a specified number of times (2.6).

RUBOUT — The RUBOUT or DELETE key generates an interrupt signal that is used to stop programs or to return and prompt for more input (2.6).

script — Sequences of shell commands placed in a file are called shell command scripts. You may 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 (3.2, 3.3, 3.10).

set — The built-in *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 (2.1).

setenv — Variables in the environment 'environ (5)' can be changed by using the *setenv* built-in command (2.6). The *printenv* command can be used to print the value of the variables in the environment.

shell — A shell is a command language interpreter. You may write and run your own shell, as shells are no different from 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* (3.2, 3.3, 3.10).

sort — The *sort* program sorts a sequence of lines in ways that can be controlled by argument flags (1.5).

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 (2.6).

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* (1.3, 3.8).

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 (3.5, 3.6).

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* (3.4).

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 (3.7).

termination — When a command that is being executed finishes, 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 (1.8). The *kill* program terminates specified command whose numbers are given (2.6).

then — The *then* command is part of the shell's 'if-then-else-endif' control construct used in command scripts (3.6).

time — The *time* command can be used to measure the amount of CPU and real time consumed by a specified command (2.1, 2.6).

troff — The *troff* program is used to typeset documents. See also *nroff* (3.2).

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 (2.1).

unalias — The *unalias* command removes aliases (2.6).

UNIX — UNIX is an operating system on which *csh* runs. UNIX provides facilities that allow *csh* to invoke other programs such as editors and text formatters.

unset — The *unset* command removes the definitions of shell variables (2.2, 2.6).

variable expansion — See *variables* and *expansion* (2.2, 3.4).

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) (2.2).

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 (3.10).

wait — The built-in command *wait* causes the shell to pause, and not prompt, until all commands run in the background have terminated (2.6).

while — The *while* built-in control construct is used in shell command scripts (3.7).

word — A sequence of characters that forms an argument to a command is called a *word.* Many characters that 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 (1.1, 1.6). 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 directory's 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 UNIX (2.3).

Edit: A Tutorial

Acknowledgement

This document is based on *Edit: A Tutorial* by Ricki Blau and James Joyce.

Introduction

Text editing using a terminal connected to a computer allows you to create, modify, and print text easily. Creating text is much like typing it as you would on an electric typewriter. Modifying text involves telling the text editor what to add, change, or delete. Printing text is accomplished merely by giving the proper command, with or without special instructions as to the format of the desired output.

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions, which will lead you through the fundamental steps of creating and revising a file of text. After scanning each lesson and before beginning the next, you should follow the examples at a terminal to get a feeling for the actual process of text editing. Set aside some time for experimentation, and you will soon become familiar with using the computer to write and modify text. In addition to the actual use of the text editor, other features of UNIX will be very important to your work. You can begin to learn about these other features by reading *UNIX for Beginners* or one of the other tutorials that provide a general introduction to the system. You will be ready to proceed with this lesson as soon as you are familiar with your terminal and its special keys, the login procedure, and the ways of correcting typing errors. Let's first define some terms:

program — A set of instructions given to the computer, describing the sequence of steps that the computer performs in order to accomplish a specific task. As an example, a series of steps to balance your checkbook is a program.

UNIX — UNIX is a special type of program called an *operating system*. It supervises the machinery and all other programs comprising the total computer system.

edit — *edit* is the name of the UNIX text editor that you will be learning to use. A *text editor* (or "editor" for short) is a program that aids you in writing or revising text. *Edit* was designed for beginning users, and is a simplified version of an editor called *ex.*

file — Each UNIX account is allotted space for the permanent storage of information, such as programs, data or text. A file is a logical unit of data, for example, an essay, a program, or a chapter from a book, which is stored on a computer system. Once you create a file, it is kept until you instruct the system to remove it. You may create a file during one UNIX session, log out, and return to use it at a later time. Files contain anything you choose to write and store in them. The sizes of files vary to suit your needs; one file might hold only a single number while another might contain a very long document or program. The only way to save information from one session to the next is to store it in a file.

filename — Filenames distinguish one file from another. They serve the same purpose as the labels of manila folders in a file cabinet. In order to get access to information in a file, you use the name of that file in a UNIX command, and the system automatically locates the file.

disk — Files are stored on an input/output device called a disk, which looks something like a stack of phonograph records. Each surface is coated with a material similar to the coating on magnetic recording tape, on which information is recorded.

buffer — A temporary work space, made available to the user for the duration of a session of text editing and used for building and modifying the text file. We can imagine the buffer as a blackboard that is erased after each class, where each session with the editor is a class.

Session 1: Creating a File of Text

To use the editor you must first make contact with the computer by logging in to UNIX. We'll quickly review the standard UNIX login procedure.

If the terminal you are using is directly linked to the computer, turn it on and press carriage return, usually labeled "RETURN". If your terminal connects with the computer over a telephone line, turn on the terminal, dial the system access number, and, when you hear a high-pitched tone, place the receiver of the telephone in the acoustic coupler. Press carriage return once and wait for the login message:

:login:

Type your login name, which identifies you to UNIX, on the same line as the login message, and press carriage return. If the terminal you are using has both upper and lower case, be sure you enter your login name in lower case; otherwise UNIX assumes your terminal has only upper case and will not recognize lower case letters you may type. UNIX types ":login:" and you reply with your login name, for example "susan":

:login: **susan** *(and press carriage return)*

(In the examples, input typed by the user appears in **bold** to distinguish it from the responses from UNIX.)

UNIX next responds with a request for a password as an additional precaution to prevent unauthorized people from using your account. The password does not appear when you type it, to prevent others from seeing it. The message is:

Password: *(type your password and press carriage return)*

If any of the information you gave during the login sequence was mistyped or incorrect, UNIX responds with

Login incorrect.
:login:

in which case you should start the login process anew. Assuming that you have successfully logged in, UNIX may print the message of the day and eventually presents you with a $ or % at the beginning of a fresh line. The $ or % is the UNIX prompt symbol, which tells you that UNIX is ready to accept a command.

Asking for *edit*

You are ready to tell UNIX that you want to work with *edit*, the text editor. Now is a convenient time to choose a name for the file of text that you are about to create. To begin your editing session type **edit** followed by a space and then the filename you have selected, for example "text". When you have completed the command, press carriage return and wait for *edit*'s response:

```
$ edit text    (followed by a carriage return)
"text" No such file or directory
:
```

If you typed the command correctly, you are now in communication with *edit*. *Edit* has set aside a buffer for use as a temporary working space during your current editing session. It also checked to see if the file you named, "text", already existed. As we expected, it was unable to find such a file since "text" is the name of a new file. Edit confirms this with the line:

```
"text" No such file or directory
```

On the next line appears edit's prompt ":", announcing that edit expects a command from you. You may now begin to create the new file.

The "not found" Message

If you misspelled "edit" by typing, say, "editor", your request is handled as follows:

```
$ editor
editor:  not found
$
```

Your mistake in calling edit "editor" was treated by UNIX as a request for a program named "editor". Since there is no program named "editor", UNIX reported that the program was "not found". A new $ indicates that UNIX is ready for another command, so you may enter the correct command.

Summary

Your exchange with UNIX as you logged in and made contact with *edit* should look something like this:

```
:login: susan
Password:
Computer Center UNIX System
... A Message of General Interest ...
$ edit text
"text" No such file or directory
:
```

Entering Text

You may now begin to enter text into the buffer. This is done by *appending* text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, you are creating text. Most edit commands have two forms: a word that describes what the command does and a shorter abbreviation of that word. Either form may be used. Many beginners find the full command names easier to remember, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append" which may be abbreviated "a". Type **append** and press carriage return.

```
% edit text
:append
```

Messages from *edit*

If you make a mistake in entering a command and type something that *edit* does not recognize, *edit* responds with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing, perhaps, "add" instead of "append" or "a", you receive this message:

```
:add
add: Not an editor command
:
```

When you receive a diagnostic message, check what you typed in order to determine what part of your command confused *edit.* The message above means that edit was unable to recognize your mistyped command and, therefore, did not execute it. Instead, a new ":" appeared to let you know that *edit* is again ready to execute a command.

Text Input Mode

By giving the command "append" (or using the abbreviation "a"), you entered *text input mode,* also known as *append mode.* When you enter text input mode, *edit* responds by doing nothing. You do not receive any prompts while in text input mode. This is your signal that you are to begin entering lines of text. You can enter pretty much anything you want on the lines. The lines are transmitted one by one to the buffer and held there during the editing session. You may append as much text as you want. *When you wish to stop entering text, type a period as the only character on the line and press carriage return.* When you give this signal, you exit from text input mode and reenter command mode. *Edit* again prompts you for a command by printing ":".

Leaving append mode does not destroy the text in the buffer. You have to leave append mode to do any of the other kinds of editing, such as changing, adding, or printing text. If you type a period as the first character and type any other character on the same line, *edit* believes you want to remain in append mode and does not let you out. As this can be very frustrating, be sure to type **only** the period and carriage return.

This is as good a place as any to learn an important lesson about computers and text: a blank space is a character as far as a computer is concerned. If you so much as type a period followed by a blank (that is, type a period and then the space bar on the keyboard), you will remain in append mode with the last line of text being:

```
.
```

Let's say that the lines of text you enter are (try to type **exactly** what you see, including "thiss"):

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
.
```

The last line is the period followed by a carriage return that gets you out of append mode. If while typing the line you hit an incorrect key, recall that you may delete the incorrect character or cancel the entire line of input by erasing in the usual way. Erasing a character or cancelling a line must be done before the line has been completed by a carriage return. We will discuss changes in lines already typed in session 2.

Writing Text to Disk

You are now ready to edit the text. The simplest kind of editing is to write it to disk as a file for safekeeping after the session is over. This is the only way to save information from one session to the next, since the editor's buffer is temporary and lasts only until the end of the editing session. Thus, learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command "write" (or its abbreviation "w"):

```
:write
```

Edit copies the buffer to a disk file. If the file does not yet exist, a new file is created automatically and the presence of a "[New file]" is noted. The newly-created file is given the name specified when you entered the editor, in this case "text". To confirm that the disk file has been

successfully written, *edit* repeats the filename and gives the number of lines and the total number of characters in the file. The buffer remains unchanged by the "write" command. All of the lines that were written to disk are still in the buffer, should you want to modify or add to them.

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, *edit* prints

No current filename

in response to your write command. If this happens, you can specify the filename in a new write command:

:write text

After the "write" (or "w") type a space and then the name of the file.

Signing Off

We have done enough for this first lesson on using the UNIX text editor, and are now ready to quit the edit session. To do this we type "quit" (or "q") and press carriage return:

```
:write
"text" [New file]  3 lines, 90 characters
:quit
$
```

The $ is from UNIX to tell you that your session with *edit* is over and you may command UNIX further. Since we want to end the entire session at the terminal we also need to exit from UNIX. In response to the UNIX prompt of "%" type a "control d". This is done by holding down the control key (usually labeled "CTRL") and simultaneously pressing the d key. This ends your session with UNIX and makes the terminal ready for the next user. It is always important to type a "control-d" at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, tempting even the most honest of souls.

This is the end of the first session on UNIX text editing.

Session 2

Login with UNIX as in the first session:

```
:login: susan  (carriage return)
Password:      (give password and carriage return)
Computer Center UNIX System
$
```

This time when you say that you want to edit, you can specify the name of the file you worked on last time. This starts *edit* working and it fetches the contents of the file into the buffer, so that you can resume editing the same file. When *edit* has copied the file into the buffer, it repeats the file name and tells you the number of lines and characters the file contains. Thus,

```
$ edit text
"text" 3 lines, 90 characters
:
```

means you asked *edit* to fetch the file named "text" for editing, causing it to copy the 90 characters of text into the buffer. *Edit* awaits your further instructions. In this session, we will append more text to our file, print the contents of the buffer, and learn to change the text of a line.

Adding More Text to the File

If you want to add to the end of your text you may do so by using the append command to enter text input mode. Here we'll use the abbreviation for the append command, "a":

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
.
```

Interrupt

Should you press the RUBOUT key (sometimes labeled DELETE) while working with *edit,* it will send this message to you:

```
Interrupt
:
```

Any command that *edit* might be executing is terminated by RUBOUT or DELETE, causing *edit* to prompt you for a new command. If you are appending text at the time, you exit from append mode and return to command mode. The line of text that you were typing when the append command was interrupted is not entered into the buffer.

Making Corrections

If you have read a general introduction to UNIX, such as *UNIX for Beginners*, you will recall that you may erase individual letters that you have typed. This is done by typing the designated erase character, usually the number sign (#), as many times as there are characters you want to erase. If you make a bad start in a line and would like to begin again, this technique is cumbersome - what if you had 15 characters in your line and wanted to get rid of them? To do so either requires:

```
This is yucky tex###############
```

with no room for the great text you'd like to type, or,

```
This is yucky tex@This is great text.
```

When you type the at-sign (@), you erase the entire line typed so far. You may immediately

begin to retype the line. This, unfortunately, does not help after you type the line and press carriage return. To make corrections in lines that have been completed, you must use the editing commands covered in this session and those that follow.

Listing What's in the Buffer

Having appended text to what you wrote in Lesson 1, you might be curious to see what is in the buffer. To print the contents of the buffer, type the command:

:**1,$p**

The "1" stands for line 1 of the buffer, the "$" is a special symbol designating the last line of the buffer, and "p" (or **print**) is the command to print from line 1 to the end of the buffer. Thus, "1,$p" gives you:

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
```

Occasionally, you may enter into the buffer a character that can't be printed, which is done by striking a key while the CTRL key is depressed. In printing lines, *edit* uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character "control-a" into the word "illustrate" by accidently holding down the CTRL key while typing "a". *Edit* would display

```
it does illustr^Ate the editor.
```

if you asked to have the line printed. To represent the control-a, edit shows "^A". The sequence "^" followed by a capital letter stands for the one character entered by holding down the CTRL key and typing the letter that appears after the "^". We'll soon discuss the commands that can be used to correct this typing error.

In looking over the text we see that "this" is typed as "thiss" in the second line, as suggested. Let's correct the spelling.

Finding Things in the Buffer

In order to change something in the buffer we first need to find it. We can find "thiss" in the text we have entered by looking at a listing of the lines. Physically speaking, we search the lines of text looking for "thiss" and stop searching when we have found it. To tell *edit* to search for something type it inside slash marks:

:**/thiss/**

By typing **/thiss/** and pressing carriage return *edit* is instructed to search for "thiss". If we asked *edit* to look for a pattern of characters that it could not find in the buffer, it would respond "Pattern not found". When *edit* finds the characters "thiss", it prints the line of text for your inspection:

```
And thiss is some more text.
```

Edit is now positioned in the buffer at the line that it just printed, ready to make a change in the line.

The Current Line

At all times during an editing session, *edit* keeps track of the line in the buffer where it is positioned. In general, the line that has been most recently printed, entered, or changed is considered to be the current position in the buffer. You can refer to your current position in the buffer by the symbol period (.) usually known by the name "dot". If you type "." and carriage

return you will be instructing *edit* to print the current line:

```
:.
And thiss is some more text.
```

If you want to know the number of the current line, type .= and carriage return, and *edit* responds with the line number:

```
:.=
2
```

If you type the number of any line and a carriage return, *edit* positions you at that line and prints its contents:

```
:2
And thiss is some more text.
```

You should experiment with these commands to assure yourself that you understand what they do.

Numbering Lines (nu)

The **number (nu)** command is similar to print, giving both the number and the text of each printed line. To see the number and the text of the current line type

```
:nu
     2  And thiss is some more text.
```

Notice that the shortest abbreviation for the number command is "nu" (and not "n',' which is used for a different command). You may specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, "1,$nu" lists all lines in the buffer with the corresponding line numbers.

Substitute Command (s)

Now that we have found our misspelled word it is time to change it from "thiss" to "this". As far as *edit* is concerned, changing things is a matter of substituting one thing for another. As *a* stood for *append,* so *s* stands for *substitute.* We use the abbreviation "s" to reduce the chance of mistyping the substitute command. This command instructs *edit* to make the change:

```
2s/thiss/this/
```

We first indicate the line to be changed, line 2, and then type an "s" to indicate we want substitution. Inside the first set of slashes are the characters that we want to change, followed by the characters to replace them and then a closing slash mark. To summarize:

2s/ *what is to be changed* / *what to change to* /

If *edit* finds an exact match of the characters to be changed, it makes the change **only** in the first occurrence of the characters. If it does not find the characters to be changed it responds:

```
Substitute pattern match failed
```

indicating your instructions could not be carried out. When *edit* does find the characters you want to change, it makes the substitution and automatically prints the changed line, so that you can check that the correct substitution was made. In the example,

```
:2s/thiss/this/
And this is some more text.
:
```

line 2 (and line 2 only) is searched for the characters "thiss", and when the first exact match is found, "thiss" is changed to "this". Strictly speaking, it was not necessary above to specify the number of the line to be changed. In

:s/thiss/this/

edit assumes that we mean to change the line where we are currently positioned ("."). In this case, the command without a line number would have produced the same result because we were already positioned at the line we wished to change.

For another illustration of substitution we may choose the line:

Text editing is strange, but nice.

We might like to be a bit more positive. Thus, we could take out the characters "strange, but " so the line would read:

Text editing is nice.

A command that first positions *edit* at that line and then makes the substitution is:

:/strange/s/strange, but //

Here we have combined our search with our substitution. Such combinations are perfectly legal. This illustrates that we do not necessarily have to use line numbers to identify a line to edit. Instead, we may identify the line we want to change by asking *edit* to search for a specified pattern of letters that occurs in that line. The parts of the above command are:

/strange/	tells *edit* to find the characters "strange" in the text
s	tells *edit* we want to make a substitution
/strange, but //	substitutes nothing at all for the characters "strange, but "

You should note the space after "but" in "/strange, but /". If you do not indicate the space is to be taken out, your line will be:

Text editing is nice.

which looks a little funny because of the extra space between "is" and "nice". Again, we realize from this that a blank space is a real character to a computer, and in editing text we need to be aware of spaces within a line just as we would be aware of an "a" or a "4".

Another Way to List What's in the Buffer (z)

Although the print command is useful for looking at specific lines in the buffer, other commands can be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command **z.** If you type

:1z

edit starts with line 1 and continues printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the next segment of text, give the command

:z

If no starting line number is given for the z command, printing starts at the "current" line, in this case the last line printed. Viewing lines in the buffer one screen full at a time is known as *paging*. Paging can also be used to print a section of text on a hard-copy terminal.

Saving the Modified Text

This seems to be a good place to pause in our work, and so we should end the second session. If you (in haste) type "q" to quit the session your dialogue with *edit* will be:

```
:q
No write since last change (q! quits)
:
```

This is edit's warning that you have not written the modified contents of the buffer to disk. You run the risk of losing the work you have done during the editing session since the latest write command. Since in this lesson we have not written to disk at all, everything we have done would be lost. If we did not want to save the work done during this editing session, we would have to type "q!" to confirm that we indeed wanted to end the session immediately, losing the contents of the buffer. However, since we want to preserve what we have edited, we need to say:

```
:w
"text" 6 lines, 171 characters
```

and then,

```
:q
$ {control d}
```

and hang up the phone or turn off the terminal when UNIX asks for a name. This is the end of the second session on UNIX text editing.

Session 3

Bringing Text into the Buffer (e)

Login to UNIX and make contact with *edit*. Try to login without looking at the notes.

Did you remember to give the name of the file you wanted to edit? That is, did you say

```
% edit text
```

or simply

```
% edit
```

Both ways get you in contact with edit, but the first way brings a copy of the file named "text" into the buffer. If you did forget to tell *edit* the name of your file, you can get it into the buffer by saying:

```
:e text
"text" 6 lines, 171 characters
```

The command **edit,** which may be abbreviated "**e**", tells *edit* that you want to erase anything that might already be in the buffer and bring a copy of the file "text" into the buffer for editing. You may also use the *edit* (e) command to change files in the middle of an editing session or to give *edit* the name of a new file that you want to create. Because the *edit* command clears the buffer, you receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving Text in the Buffer (m)

Edit allows you to move lines of text from one location in the buffer to another by means of the **move** (**m**) command:

```
:2,4m$
```

This command directs *edit* to move lines 2, 3, and 4 to the end of the buffer ($). The format for the move command is to specify the first line to be moved, the last line to be moved, the move command "m", and the line after which the moved text is to be placed. Thus,

```
:1,6m20
```

instructs *edit* to move lines 1 through 6 (inclusive) to a position after line 20 in the buffer. To move only one line, say, line 4, to a position in the buffer after line 6, the command would be "4m6".

Let's move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command that changes more than one line of the buffer, *edit* tells how many lines were affected by the change. The last moved line is printed for your inspection. If you want to see more than just the last line, use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

```
This is some sample text.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
```

We can restore the original order by typing:

:4,$m1

or, combining context searching and the move command:

:/And this is some/,/This is text/m/This is some sample/

The problem with combining context searching with the move command is that the chance of making a typing error in such a long command is greater than if one types line numbers.

Copying Lines (copy)

The **copy** command is used to make a second copy of specified lines, leaving the original lines where they were. Copy has the same format as the move command, for example:

:12,15copy $

makes a copy of lines 12 through 15, placing the added lines after the buffer's end ($). Experiment with the copy command so that you can become familiar with how it works. Note that the shortest abbreviation for copy is "co" (and **not** the letter "c", which has another meaning).

Deleting Lines (d)

Suppose you want to delete the line

This is text added in Session 2.

from the buffer. If you know the number of the line to be deleted, you can type that number followed by "**delete**" or "**d**". This example deletes line 4:

:4d
It doesn't mean much here, but

Here "4" is the number of the line to be deleted and "delete" or "d" is the command to delete the line. After executing the delete command, *edit* prints the line that has become the current line (".").

If you do not happen to know the line number you can search for the line and then delete it using this sequence of commands:

:/added in Session 2./
This is text added in Session 2.
:d
It doesn't mean much here, but

The "/added in Session 2./" asks *edit* to locate and print the next line that contains the indicated text. Once you are sure that you have correctly specified the line that you want to delete, you can enter the delete (d) command. In this case you don't need to specify a line number before the "d". If no line number is given, *edit* deletes the current line ("."), that is, the line found by our search. After the deletion, your buffer should contain:

This is some sample text.
And this is some more text.
Text editing is nice.
It doesn't mean much here, but
it does illustrate the editor.

To delete both lines 2 and 3:

And this is some more text.
Text editing is nice.

you type

:2,3d

which specifies the range of lines from 2 to 3, and the operation on those lines - "d" for delete.

Again, this presumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command as so:

:/And this is some/,/Text editing is nice./d

A word or two of caution:

In using the search function to locate lines to be deleted you should be **absolutely sure** the characters you give as the basis for the search will take *edit* to the line you want deleted. *Edit* searches for the first occurrence of the characters starting from where you last edited - that is, from the line you see printed if you type dot (.).

A search based on too few characters may result in the wrong lines being deleted, which *edit* will do as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing carriage return to send the command on its way.

Undo (u) to the Rescue

The **undo (u)** command can reverse the effects of the last command. To undo the previous command, type "u" or "undo". Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give. It is possible to undo only commands that have the power to change the buffer, for example delete, append, move, copy, substitute, and even undo itself. The commands write (w) and *edit* (e), which interact with disk files, cannot be undone, nor can commands such as print that do not change the buffer. Most importantly, the **only** command that can be reversed by undo is the one last "undo-able" command you gave.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command we gave deleted the lines formerly numbered 2 and 3. Executing undo at this moment reverses the effects of the deletion, causing those two lines to be replaced in the buffer.

```
:u
2 more lines in file after undo
And this is some more text.
```

Here again, *edit* informs you if the command affects more than one line, and prints the text of the line, which is now "dot" (the current line).

More about the Dot (.) and Buffer End ($)

The function assumed by the symbol dot depends on its context. It can be used

1. to exit from append mode. To do this, we type dot (and only a dot) on a line and press carriage return;
2. to refer to the line we are at in the buffer.

Dot can also be combined with the equal sign to get the number of the line currently being edited:

```
:.=
```

Thus if we type ".=" we are asking for the number of the line and if we type "." we are asking for the text of the line.

In this editing session and the last, we used the dollar sign to indicate the end of the buffer in commands such as print, copy, and move. The dollar sign as a command asks *edit* to print the last line in the buffer. If the dollar sign is combined with the equal sign ($=) edit will print the line number corresponding to the last line in the buffer.

"." and "$" therefore represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example

:.,$d

instructs *edit* to delete all lines from the current line (.) to the end of the buffer.

Moving around in the Buffer (+ and -)

During an editing session, you often want to go back and re-read a previous line. You could specify a context search for a line you want to read if you remember some of its text, but if you simply want to see what was written a few, say 3, lines ago, you can type

-3p

This tells *edit* to move back to a position 3 lines before the current line (.) and print that line. We can move forward in the buffer similarly:

+2p

instructs *edit* to print the line that is 2 ahead of our current position.

You may use "+" and "-" in any command where edit accepts line numbers. Line numbers specified with "+" or "-" can be combined to print a range of lines. The command

:-1,+2copy$

makes a copy of 4 lines: the current line, the line before it, and the two after it. The copied lines are placed after the last line in the buffer ($).

Try typing only "-"; you move back one line just as if you had typed "-1p". Typing the command "+" works similarly. You might also try typing a few plus or minus signs in a row (such as "+++") to see *edit*'s response. Typing a carriage return alone on a line is the equivalent of typing "+1p"; it moves you one line ahead in the buffer and prints that line.

If you are at the last line of the buffer and try to move further ahead, perhaps by typing a "+" or a carriage return alone on the line, *edit* reminds you that you are at the end of the buffer:

At end-of-file

Similarly, if you try to move to a position before the first line, edit prints one of these messages:

Nonzero address required on this command
Negative address - first buffer line is 1

The number associated with a buffer line is the line's "address", because it can be used to locate the line.

Changing Lines (c)

You may on occasion want to delete certain lines and insert new text in their place. This can be accomplished easily with the **change (c)** command. The change command instructs *edit* to delete specified lines and then switch to text input mode in order to accept the text that replaces them. Let's say we want to change the first two lines in the buffer:

This is some sample text.
And this is some more text.

to read

This text was created with the UNIX text editor.

To do so, you can type:

```
:1,2c
2 lines changed
This text was created with the UNIX text editor.
.
:
```

In the command **1,2c** we specify that we want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines will be deleted. After a carriage return enters the change command, *edit* notifies you if more than one line will be changed and places you in text input mode. Any text then typed on the following lines is inserted into the position where lines were deleted by the change command. You remain in text input mode until you exit in the usual way, by typing a period alone on a line. Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with UNIX.

Session 4

This lesson covers several topics, starting with commands that apply throughout the buffer, characters with special meanings, and how to issue UNIX commands while in the editor. The next topics deal with files: more on reading and writing, and methods of recovering files lost in a crash.

Making Commands Global (g)

One disadvantage to the commands we have used for searching or substituting is that if you have a number of instances of a word to change it appears that you have to type the command repeatedly, once for each time the change needs to be made. *Edit,* however, provides a way to make commands apply to the entire contents of the buffer - the **global (g)** command.

To print all lines containing a certain sequence of characters (say, "text") the command is:

:g/text/p

The "g" instructs *edit* to make a global search for all lines in the buffer containing the characters "text". The "p" prints the lines found.

To issue a global command, start by typing a "g" and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed on the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word "text" to the word "material" the command is a combination of the global search and the substitute command:

:g/text/s/text/material/g

Note the "g" at the end of the global command which instructs *edit* to change each and every instance of "text" to "material". If you do not type the "g" at the end of the command only the *first* instance of "text" in each line is changed (the normal result of the substitute command). The "g" at the end of the command is independent of the "g" at the beginning. The command

:14s/text/material/g

changes every instance of "text" in line 14 alone. Further, neither command changes "Text" to "material" because "Text" begins with a capital rather than a lower-case *t.*

Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a "p" at the end of the global command:

:g/text/s/text/material/gp

The usual qualification should be made about using the global command in combination with any other - in essence, be sure of what you are telling *edit* to do to the entire buffer. For example,

```
:g/ /d
72 less lines in file after global
```

deletes every line containing a blank anywhere in it. This could adversely affect your document, since most lines have spaces between words and thus would be deleted. After executing the global command, edit prints a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. You should experiment with the global command on a small buffer of text to see what it can do for you.

More about Searching and Substituting

In using slashes to identify a character string that we want to search for or change, we have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change "noun" to "nouns" we may type either

:/noun/s/noun/nouns/

as we have done in the past, or a somewhat abbreviated command:

```
:/noun/s//nouns/
```

In this example, the characters to be changed are not specified - there are no characters, not even a space, between the two slash marks that indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean "use the characters we last searched for as the characters to be changed."

Similarly, the last context search may be repeated by typing a pair of slashes with nothing between them:

```
:/does/
It doesn't mean much here, but
://
it does illustrate the editor.
```

Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters "does".

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the character string.

Special Characters

Two characters have special meanings when used in specifying searches: "$" and "^". *Edit* understands "$" to mean "end of the line" and is used to identify strings that occur at the end of a line. The command

```
:g/ing$/s//ed/p
```

tells the editor to search for all lines ending in "ing" (and nothing else, not even a blank space), to change each final "ing" to "ed" and print the changed lines.

The symbol "^" indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert "1." and a space at the beginning of the current line.

The characters "$" and "^" have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to temporarily lose its special significance by typing another special character, the backslash (\), before it.

```
:s/\$/dollar/
```

looks for the character "$" in the current line and replaces it by the word "dollar". Were it not for the backslash, the "$" would have represented "the end of the line" in your search, not necessarily the character "$". The backslash retains its special significance at all times.

Issuing UNIX Commands from the Editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not functions of the editor, and so they require the use of UNIX system commands (also referred to as "shell" commands, as "shell" is the name of the program that processes UNIX commands). You do not need to quit the editor to execute a UNIX command as long as you indicate that it is to be sent to the shell for execution. To use the UNIX command *rm* to remove the file named "junk" type:

```
:!rm junk
!
:
```

The exclamation mark (!) indicates that the rest of the line is to be processed as a UNIX command. If the buffer contents have not been written since the last change, a warning is printed before the command is executed. The editor prints a "!" when the command is completed. The tutorial *UNIX for Beginners* describes useful features of the system, of which the editor is only one part.

Filenames and File Manipulation

Throughout each editing session, *edit* keeps track of the name of the file being edited as the *current filename. Edit* remembers as the current filename the name given when you entered the editor. The current filename changes whenever the *edit* (e) command is used to specify a new file. Once *edit* has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, *edit,* as we have seen, supplies the current filename. You can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
"chapter3" 283 lines, 8698 characters
```

The current filename remembered by the editor *is not changed as a result of the write command unless it is the first filename given in the editing session.* Thus, in the next write command which does not specify a name, edit writes onto the current file and not onto the file "chapter3".

The File (f) Command

To ask for the current filename, type **file** (or **f**). In response, the editor provides current information about the buffer, including the filename, your current position, and the number of lines in the buffer:

```
:f
"text" [Modified] line 3 of 4 --75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor tells you that the file has been "[Modified]". After you save the changes by writing onto a disk file, the buffer is no longer considered modified:

```
:w
"text" 4 lines, 88 characters
:f
"text" line 3 of 4 --75%--
```

Reading Additional Files (r)

The **read (r)** command allows you to add the contents of a file to the buffer without destroying the text already there. To use it, specify the line after which the new text will be placed, the command *r,* and then the name of the file.

```
:$r bibliography
"bibliography" 18 lines, 473 characters
```

This command reads in the file *bibliography* and adds it to the buffer after the last line. The current filename is not changed by the read command unless it is the first filename given in the editing session.

Writing Parts of the Buffer

The **write (w)** command can write all or part of the buffer to a file you specify. We are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

:45,$w ending

Here all lines from 45 through the end of the buffer are written onto the file named *ending.* The lines remain in the buffer as part of the document you are editing, and you may continue to edit the entire buffer.

Recovering Files

Under most circumstances, *edit*'s crash recovery mechanism is able to save work to within a few lines of changes after a crash or if the phone is hung up accidently. To recover the file, enter the editor and type the command **recover** (**rec**), followed by the name of the lost file.

:recover chap6

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file.

Other Recovery Techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command **preserve** (**pre**), which saves the buffer as if the system had crashed. If you are writing a file and you get the message "Quota exceeded", you have tried to use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case you should use the shell escape from the editor (!) to remove some files you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

:preserve

and then seek help. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. After a preserve, you can use the recover command once the problem has been corrected.

Ex Reference Manual

ABSTRACT

Ex a line oriented text editor that supports both command and display oriented editing. This reference manual describes the command oriented part of *ex;* the display editing features of *ex* are described in *An Introduction to Display Editing with Vi.* Other documents about the editor include the introduction *Edit: A Tutorial*, and a *Ex/Vi Quick Reference* card.

May 21, 1984

Ex Reference Manual

Acknowledgement

This document is based on the *Ex Reference Manual* by William Joy, revised by Mark Horton.

1. Starting Ex

Each instance of the editor has a set of options, which can be set to tailor it to your liking. The command *edit* invokes a version of *ex* designed for more casual or beginning users. It causes the default settings of some of these options to be changed. To simplify the description that follows, we assume the default settings of the options.

When invoked, *ex* determines the terminal type from the TERM variable in the environment. It there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, then that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /) then the editor seeks the description of the terminal in that file (rather than the default **/etc/termcap**.) If the environment contains a variable EXINIT, then the editor executes the commands in that variable; otherwise if there is a file *.exrc* in your HOME directory *ex* reads commands from that file, simulating a *source* command. Option setting commands placed in EXINIT or *.exrc* are executed before each editor session.

A command to enter *ex* has the following prototype:†

ex [**-**] [**-v**] [**-t** *tag*] [**-r**] [**-l**] [**-w***n*] [**-x**] [**-R**] [+*command*] name ...

Most commonly, you simply edit a single file, as in

ex name

The **-** command line option option suppresses all interactive-user feedback and is useful in processing editor scripts in command files. The **-v** option is equivalent to using *vi* rather than *ex*. The **-t** option is equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition. The **-r** option recovers the last saved version of the named file after an editor or system crash. If no file is specified, a list of saved files is typed. The **-l** option sets up for editing LISP, setting the *showmatch* and *lisp* options. The **-w** option sets the default window size to *n*, and is useful on dialups to start in small windows. The **-x** option causes *ex* to prompt for a *key*, which is used to encrypt and decrypt the contents of the file. The file should already be encrypted using the same key, see *crypt*(1). The **-R** option sets the *readonly* option at the start. *Name* arguments indicate files to be edited. An argument of the form +*command* indicates that the editor should begin by executing the specified command. If *command* is omitted, then it defaults to "$", positioning the editor at the last line of the first file. Other useful commands are scanning patterns of the form "/pat" or line numbers, e.g. "+100" starting at line 100.

2. File Manipulation

† Brackets '[' ']' surround optional parameters here.

2.1. Current File

Ex is normally editing the contents of a single file, whose name is recorded in the *current* file name. *Ex* performs all editing actions in a buffer (actually a temporary file) into which the text of the file is initially read. Changes made to the buffer have no effect on the file being edited unless and until the buffer contents are written out to the file with a *write* command. After the buffer contents are written, the previous contents of the written file are no longer accessible. When a file is edited, its name becomes the current file name, and its contents are read into the buffer.

The current file is almost always considered to be *edited.* This means that the contents of the buffer are logically connected with the current file name, so that writing the current buffer contents onto that file, even if it exists, is reasonable. If the current file is not *edited* then *ex* does not normally write on it if it already exists.*

2.2. Alternate File

Each time a new value is given to the current file name, the previous current file name is saved as the *alternate* file name. Similarly if a file is mentioned but does not become the current file, it is saved as the alternate file name.

2.3. Filename Expansion

Filenames within the editor may be specified using the normal shell expansion conventions. In addition, the character '%' in filenames is replaced by the *current* file name and the character '#' by the *alternate* file name.†

2.4. Multiple Files and Named Buffers

If more than one file is given on the command line, then the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list.* The current argument list may be displayed with the *args* command. The next file in the argument list may be edited with the *next* command. The argument list may also be respecified by specifying a list of names to the *next* command. These names are expanded, the resulting list of names becomes the new argument list, and *ex* edits the first file on the list.

2.5. Read Only

You can use *ex* in *read only* mode to look at files that you have no intention of modifying. This mode protects you from accidently overwriting the file. Read only mode is on when the *readonly* option is set. It can be turned on with the **-R** command line option, by the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting *noreadonly*. You can write, even while in read only mode, if you know what you are doing. You can write to a different file, or can use the ! form of write.

3. Exceptional Conditions

3.1. Errors and Interrupts

When errors occur, *ex* usually rings the terminal bell (except on intelligent terminals, where it makes the screen flash); in any case, it prints an error diagnostic. If the primary input is from a file, editor processing terminates. If an interrupt signal is received, *ex* prints "Interrupt" and returns to its command level. If the primary input is a file, *ex* exits.

---

* The *file* command will say "[Not edited]" if the current file is not considered edited.

† This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an *edit* command after a *No write since last change* diagnostic is received.

3.2. Recovering from Hangups and Crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots, in the second) attempts to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hangup or editor crash. To recover a file you can use the **-r** option of *ex*. If you were editing the file *resume,* then you should change to the directory where you were when the crash occurred, and then give the command

ex -r *resume*

Check that the retrieved file has indeed been salvaged. Then *write* it over the previous contents of that file.

The command

ex -r

prints a list of the files that have been saved for you. (In the case of a hangup, the file does not appear in the list, although it can still be recovered.)

4. Editing Modes

Ex has five distinct modes. The primary mode is *command* mode. Commands are entered in command mode when a ':' prompt is present, and are executed each time a complete line is sent. In *text input* mode *ex* gathers input lines and places them in the file. The *append, insert,* and *change* commands use text input mode. No prompt is printed when you are in text input mode. You leave insert mode by typing a '.' alone at the beginning of a line, and *command* mode resumes.

The last three modes are *open* and *visual* modes, entered by the commands of the same name, and, within open and visual modes *text insertion* mode. *Open* and *visual* modes allow local editing operations to be performed on the text in the file. The *open* command displays one line at a time on any terminal while *visual* works on CRT terminals with random positioning cursors, using the screen as a (single) window for file editing changes. These modes are described (only) in *An Introduction to Display Editing with Vi.*

5. Command Structure

Most command names are English words, and initial prefixes of the words are acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands.*

5.1. Command Parameters

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses are discussed below. A number of commands also may take a trailing *count* specifying the number of lines to be involved in the command.† Thus the command "10p" prints the tenth line in the buffer while "delete 5" deletes five lines from the buffer, starting with the current line.

Some commands take other information or parameters. This information is always given after the command name.‡

---

* For example, the command *substitute* can be abbreviated 's' while the shortest available abbreviation for the *set* command is 'se'.

† Counts are rounded down if necessary.

‡ Examples include option names in a *set* command i.e. "set number", a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, i.e. "1,5 copy 25".

5.2. Command Variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an '!' immediately after the command name. Some of the default variants may be controlled by options; in this case, the '!' serves to toggle the default.

5.3. Flags after Commands

The characters '#', 'p' and 'l' may be placed after many commands.** In this case, the command abbreviated by these characters is executed after the command completes. Since *ex* normally prints the new current line after each change, 'p' is rarely necessary. Any number of '+' or '-' characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

5.4. Comments

You can give editor commands that are ignored. This is useful when making complex editor scripts for which comments are desired. The comment character is the double quote: ". Any command line beginning with " is ignored. Comments beginning with " may also be placed at the ends of commands, except in cases where they could be confused as part of text (shell escapes and the substitute commands).

5.5. Multiple Commands per Line

More than one command may be placed on a line by separating each pair of commands by a '|' character. However the *global* commands, comments, and the shell escape '!' must be the last command on a line, as they are not terminated by a '|'.

5.6. Reporting Large Changes

Most commands that change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the *report* option. This feedback helps to detect large changes so that they may be quickly and easily reversed with an *undo* if they are undesirable. After commands with more global effect such as *global* or *visual,* you are informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

6. Command Addressing

6.1. Addressing Primitives

.	The current line. Most commands leave the current line as the last line they affect. The default address for most commands is the current line, thus '.' is rarely used alone as an address.
n	The *n*th line in the editor's buffer, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for "1,$", the entire buffer.
+*n* -*n*	An offset relative to the current buffer line.†
/*pat*/ ?*pat*?	Scan forward and backward respectively for a line containing *pat*, a regular expression (as defined below). The scans normally wrap around the end of the buffer. If you just want to print the next line containing *pat*, then the trailing / or ? may be omitted. If *pat* is omitted or explicitly empty, then the last regular expression specified is located.‡

** A 'p' or 'l' must be preceded by a blank or tab except in the single special case 'dp'.

† The forms '.+3' '+3' and '+++' are all equivalent; if the current line is line 100 they all address line 103.

‡ The forms \/ and \? scan using the last regular expression used in a scan; after a substitute // and ??

''x — Before each non-relative motion of the current line '.', the previous current line is marked with a tag, subsequently referred to as ''''. This makes it easy to refer or return to this previous context. Marks may also be established by the *mark* command, using single lower case letters *x* and the marked lines referred to as ''*x*'.

6.2. Combining Addressing Primitives

Addresses to commands consist of a series of addressing primitives, separated by ',' or ';'. Such address lists are evaluated left-to-right. When addresses are separated by ';', the current line '.' is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer.†

7. Command Descriptions

The following form is a prototype for all *ex* commands:

address **command** *! parameters count flags*

All parts are optional; the degenerate case is the empty command, which prints the next line in the file. For sanity with use from within *visual* mode, *ex* ignores a ":" preceding any command.

In the following command descriptions, the default addresses are shown in parentheses; these parentheses are *not* part of the command.

(.) **append** abbr: **a**
text
.

Reads the input text and places it after the specified line. After the command, '.' addresses the last line input or the specified line if no lines were input. If address '0' is given, text is placed at the beginning of the buffer.

a!
text
.

The variant flag to *append* toggles the setting for the *autoindent* option during the input of *text.*

args

The members of the argument list are printed, with the current argument delimited by '[' and ']'.

(. , .) **change** *count* abbr: **c**
text
.

Replaces the specified lines with the input *text*. The current line becomes the last line input; if no lines were input it is left as for a *delete*.

c!
text
.

---

scan using the substitute's regular expression.

† Null address specifications are permitted in a list of addresses, the default in this case is the current line '.'; thus ',100' is equivalent to '.,100'. It is an error to give a prefix address to a command that expects none.

The variant toggles *autoindent* during the *change.*

(. , .) **copy** *addr flags* abbr: **co**

A *copy* of the specified lines is placed after *addr,* which may be '0'. The current line '.' addresses the last line of the copy. The command *t* is a synonym for *copy.*

(. , .) **delete** *buffer count flags* abbr: **d**

Removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

edit *file* abbr: **e**
ex *file*

Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last *write* command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename.

If the read of the file completes without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurs, the file is considered *edited.* If the last line of the input file is missing the trailing newline character, it is supplied and a complaint is issued. This command leaves the current line '.' at the last line read.‡

e! *file*

The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes that have been made before editing the new file.

e +*n file*

Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, e.g.: "+/pat".

file abbr: **f**

Prints the current file name, whether it has been '[Modified]' since the last *write* command, whether it is *read only*, the current line, the number of lines in the buffer, and the percentage of the way through the buffer the current line is.*

file *file*

The current file name is changed to *file,* which is considered '[Not edited]'.

---

‡ If executed from within *open* or *visual,* the current line is initially the first line of the file.

* In the rare case that the current file is '[Not edited]' this is noted also; in this case you have to use the form **w!** to write to the file, since the editor is not sure that a **write** will not destroy a file unrelated to the current contents of the buffer.

(1 , $) **global** */pat/ cmds* abbr: **g**

First marks each line among those specified that matches the given regular expression. Then the given command list is executed with '.' initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a '\'. If *cmds* (and possibly the trailing / delimiter) is omitted, each line matching *pat* is printed. *Append, insert,* and *change* commands and associated input are permitted; the '.' terminating input may be omitted if it would be on the last line of the command list. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

The *global* command itself may not appear in *cmds.* The *undo* command is also not permitted there, as *undo* instead can be used to reverse the entire *global* command. The options *autoprint* and *autoindent* are inhibited during a *global,* (and possibly the trailing / delimiter) and the value of the *report* option is temporarily infinite, in deference to a *report* for the entire global. Finally, the context mark '' '' is set to the value of '.' before the global command begins and is not changed during a global command, except perhaps by an *open* or *visual* within the *global.*

g! */pat/ cmds* abbr: **v**

The variant form of *global* runs *cmds* at each line not matching *pat.*

(.) **insert** abbr: **i**
text
.

Places the given text before the specified line. The current line is left at the last line input; if there were none input, it is left at the line before the addressed line. This command differs from *append* only in the placement of text.

i!
text
.

The variant toggles *autoindent* during the *insert.*

(. , .+1) **join** *count flags* abbr: **j**

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was a '.' at the end of the line, or none if the first following character is a ')'. If there is already white space at the end of the line, then the white space at the start of the next line is discarded.

j!

The variant causes a simpler *join* with no white space processing; the characters in the lines are simply concatenated.

(.) **k** *x*

The *k* command is a synonym for *mark.* It does not require a blank or tab before the following letter.

(. , .) **list** *count flags*

Prints the specified lines in a more unambiguous way: tabs are printed as '^I' and the end of each line is marked with a trailing '$'. The current line is left at the last line printed.

(.) **mark** *x*

Gives the specified line mark *x,* a single lower case letter. The *x* must be preceded by a blank or a tab. The addressing form ''x' then addresses this line. The current line is not affected by this command.

(. , .) **move** *addr* abbr: **m**

The *move* command repositions the specified lines to be after *addr*. The first of the moved lines becomes the current line.

next abbr: **n**

The next file from the command line argument list is edited.

n!

The variant suppresses warnings about the modifications to the buffer not having been written out, discarding (irretrievably) any changes that may have been made.

n *filelist*
n +*command filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

(. , .) **number** *count flags* abbr: # or **nu**

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **open** *flags* abbr: **o**
(.) **open** /*pat*/ *flags*

Enters intraline editing *open* mode at each addressed line. If *pat* is given, then the cursor is placed initially at the beginning of the string matched by the pattern. To exit this mode use Q. See *An Introduction to Display Editing with Vi* for more details.
‡

preserve

The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a *write* command has resulted in an error and you don't know how to save your work. After a *preserve* you should seek help.

(. , .) **print** *count* abbr: **p** or **P**

Prints the specified lines with non-printing characters printed as control characters '^*x*'; delete (octal 177) is represented as '^?'. The current line is left at the last line printed.

---

‡ Not available in all editors due to memory constraints.

(.) **put** *buffer* abbr: **pu**

Puts back previously *deleted* or *yanked* lines. Normally used with *delete* to effect movement of lines, or with *yank* to effect duplication of lines. If no numbered *buffer* is specified, then the last *deleted* or *yanked* text is restored.*

quit abbr: **q**

Causes *ex* to terminate. No automatic write of the editor buffer to a file is performed. However, *ex* issues a warning message if the file has changed since the last *write* command was issued, and does not *quit.*† Normally, you will wish to save your changes, and you should give a *write* command; if you wish to discard them, use the **q!** command variant.

q!

Quits from the editor, discarding changes to the buffer. No diagnostic message is issued.

---

* But no modifying commands may intervene between the *delete* or *yank* and the *put*, nor may lines be moved between files.

† *Ex* will also issue a diagnostic if there are more files in the argument list.

(.) **read** *file* abbr: **r**

Places a copy of the text of the given file in the editing buffer after the specified line. If no *file* is given the current file name is used. The current file name is not changed unless there is none in which case *file* becomes the current name. The sensibility restrictions for the *edit* command apply here also. If the file buffer is empty and there is no current name then *ex* treats this as an *edit* command.

Address '0' is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the *edit* command when the *read* successfully terminates. After a *read* the current line is the last line read.‡

(.) **read** !*command*

Reads the output of the command *command* into the buffer after the specified line. A blank or tab before the **!** is mandatory.

recover *file*

Recovers *file* from the system save area. Used after a accidental hangup of the phone** or a system crash** or *preserve* command.

rewind abbr: **rew**

The argument list is rewound, and the first file in the list is edited.

rew!

Rewinds the argument list discarding any changes made to the current buffer.

set *parameter*

With no arguments, prints those options whose values have been changed from their defaults; with parameter *all* it prints all of the option values.

‡ Within *open* and *visual* the current line is set to the first line read rather than the last.

** The system saves a copy of the file you were editing only if you have made changes to the file.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form 'set *option*' to turn them on or 'set no*option*' to turn them off; string and numeric options are assigned via the form 'set *option*=value'.

More than one parameter may be given to *set* ; parameters are interpreted left-to-right.

shell abbr: **sh**

A new shell is created. When it terminates, editing resumes.

source *file* abbr: **so**

Reads and executes commands from the specified file. *Source* commands may be nested.

(. , .) **substitute** */pat/repl/ options count flags* abbr: **s**

On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl.* If the *global* indicator option character 'g' appears, then all instances are substituted; if the *confirm* indication character 'c' appears, then before each substitution, the line to be substituted is typed with the string to be substituted marked with '↑' characters. Type a 'y' to cause the substitution to be performed; any other input causes no change to take place. After a *substitute* the current line is the last line substituted.

Lines may be split by substituting new-line characters into them. The newline in *repl* must be escaped by preceding it with a '\'. Other metacharacters available in *pat* and *repl* are described below.

(. , .) **substitute** *options count flags* abbr: **s**

If *pat* and *repl* are omitted, then the last substitution is repeated. This is a synonym for the **&** command.

(. , .) **t** *addr flags*

The *t* command is a synonym for *copy*.

ta *tag*

The focus of editing switches to the location of *tag,* switching to a different line in the current file where it is defined, or if necessary to another file.‡

The tags file is normally created by a program such as *ctags,* and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form that the editor uses to find the tag; this field is usually a contextual scan using '/*pat*/' to be immune to minor changes in the file. Such scans are always performed as if *nomagic* were set.

The tag names in the tags file must be sorted alphabetically. ‡

undo abbr: **u**

Reverses the changes made in the buffer by the last buffer editing command. Note that *global* commands are considered a single command for the purpose of *undo* (as are *open* and *visual.)* Also, the commands *write* and *edit,* which interact with the file system, cannot be undone. *Undo* is its own inverse.

Undo always marks the previous value of the current line '.' as ''''. After an *undo* the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as *global* and *visual* the current line regains its pre-command value after an *undo.*

(1 , $) **v** /*pat*/ *cmds*

A synonym for the *global* command variant **g!**, running the specified *cmds* on each line that does not match *pat*.

version abbr: **ve**

Prints the current version number of the editor as well as the date the editor was last changed.

---

‡ If you have modified the current file before giving a *tag* command, you must write it out; giving another *tag* command; specifying no *tag* reuses the previous tag.

‡ Not available in all editors due to memory constraints.

(.) **visual** *type count flags* abbr: **vi**

Enters visual mode at the specified line. *Type* is optional and may be '-' , '↑' or '.' as in the *z* command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option *window.* See the document *An Introduction to Display Editing with Vi* for more details. To exit this mode, type Q.

visual file
visual +*n* file

From visual mode, this command is the same as edit.

(1 , $) **write** *file* abbr: **w**

Writes changes made back to *file*, printing the number of lines and characters written. Normally *file* is omitted and the text goes back where it came from. If a *file* is specified, then text is written to that file.* If the file does not exist, it is created. The current file name is changed only if there is no current file name; the current line is never changed.

If an error occurs while writing the current and *edited* file, the editor considers that there has been "No write since last change" even if the buffer had not previously been modified.

(1 , $) **write**>> *file* abbr: **w**>>

Writes the buffer contents at the end of an existing file.

w! *name*

Overrides the checking of the normal *write* command, and will write to any file that the system permits.

(1 , $) **w** !*command*

Writes the specified lines into *command.* Note the difference between **w!**, which overrides checks, and **w !**, which writes to a command.

* The editor writes to a file only if (1) it is the current file and is *edited*; or (2) the file does not exist, or (3) the file is actually a teletype, */dev/tty*, */dev/null*. Otherwise, you must give the variant form **w!** to force the write.

wq *name*

Like a *write* and then a *quit* command.

wq! *name*

The variant overrides checking on the sensibility of the *write* command, as **w!** does.

xit *name*

If any changes have been made and not written, writes the buffer out. Then, in any case, quits.

(. , .) **yank** *buffer count* abbr: **ya**

Places the specified lines in the named *buffer,* for later retrieval via *put.* If no buffer name is specified, the lines go to a more volatile place; see the *put* command description.

(.+**1**) **z** *count*

Print the next *count* lines, default *window*.

(.) **z** *type count*

Prints a window of text with the specified line at the top. If *type* is '-' the line is placed at the bottom; a '.' causes the line to be placed in the center.* A count gives the number of lines to be displayed rather than double the number specified by the *scroll* option. On a CRT the screen is cleared before display begins unless a count which is less than the screen size is given. The current line is left at the last line printed.

! *command*

The remainder of the line after the '!' character is sent to a shell to be executed. Within the text of *command* the characters '%' and '#' are expanded as in filenames and the character '!' is replaced with the text of the previous command. Thus, in particular, '!!' repeats the last such shell escape. If any such expansion is performed, the expanded line is echoed. The current line is unchanged by this command.

* Forms 'z=' and 'z↑' also exist; 'z=' places the current line in the center, surrounds it with lines of '-' characters and leaves the current line at this line. The form 'z↑' prints the window before 'z-' would. The characters '+', '↑' and '-' may be repeated for cumulative effect.

If there has been "[No write]" of the buffer contents since the last change to the editing buffer, then a diagnostic is printed as a warning before the command is executed. A single '!' is printed when the command completes.

(*addr* , *addr*) **!** *command*

Takes the specified address range and supplies it as standard input to *command;* the resulting output then replaces the input lines.

($) =

Prints the line number of the addressed line. The current line is unchanged.

(. , .) > *count flags*
(. , .) < *count flags*

Perform intelligent shifting on the specified lines; < shifts left and > shift right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non-white characters are discarded in a left-shift. The current line becomes the last line that changed due to the shifting.

^D

An end-of-file from a terminal input scrolls through the file. The *scroll* option specifies the size of the scroll, normally a half screen of text.

(.+1 , .+1)
(.+1 , .+1) |

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

(. , .) **&** *options count flags*

Repeats the previous *substitute* command.

(. , .) ~ *options count flags*

Replaces the previous regular expression with the previous replacement pattern from a substitution.

8. Regular Expressions and Substitute Replacement Patterns

8.1. Regular Expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. *Ex* remembers two previous regular expressions: the previous regular expression used in a *substitute* command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression.) The previous regular expression can always be referred to by a null *re*, e.g. '//' or '??'.

8.2. Magic and Nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are *magic* and precede them with the character '\' to use them as "ordinary" characters. With *nomagic,* the default for *edit,* regular expressions are much simpler, because there are only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\'. Note that '\' is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is *magic.*†

† To discern what is true with *nomagic* remember that the only special characters in this case are '↑' at the beginning of a regular expression, '$' at the end of a regular expression, and '\'. With *nomagic* the charac-

8.3. Basic Regular Expression Summary

The following are the elements of *magic* mode regular expressions.

char — An ordinary character matches itself. The characters '↑' at the beginning of a line, '$' at the end of line, '*' as any character other than the first, '.', '\', '[', and '~' are not ordinary characters and must be escaped (preceded) by '\' to be treated as such.

↑ — At the beginning of a pattern forces the match to succeed only at the beginning of a line.

$ — At the end of a regular expression forces the match to succeed only at the end of the line.

. — Matches any single character except the new-line character.

\< — Forces the match to occur only at the beginning of a "variable" or "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

\> — Similar to '\<', but matching the end of a "variable" or "word", i.e. either the end of the line or before character that is neither a letter, nor a digit, nor the underline character.

[*string*] — Matches any (single) character in the class defined by *string.* Most characters in *string* define themselves. A pair of characters separated by '-' in *string* defines the set of characters between the specified lower and upper bounds, thus '[a-z]' as a regular expression matches any (single) lower-case letter. If the first character of *string* is an '↑' then the construct matches characters it otherwise would not; thus '[↑a-z]' matches anything but a lower-case letter (and of course a newline). To place any of the characters '↑', '[', or '-' in *string* you must escape them with a preceding '\'.

8.4. Combining Regular Expression Primitives

The concatenation of two regular expressions matches the leftmost and then longest string that can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single character matching) regular expressions mentioned above may be followed by the character '*' to form a regular expression that matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character '~' matches the text that defined the replacement part of the last *substitute* command. A regular expression may be enclosed between the sequences '\(' and '\)' with side effects in the *substitute* replacement patterns.

8.5. Substitute Replacement Patterns

The basic metacharacters for the replacement pattern are '&' and '~'; these are given as '\&' and '\~' when *nomagic* is set. Each instance of '&' is replaced by the characters that the regular expression matched. The metacharacter '~' stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences in the replacement pattern are always introduced by the escaping character '\'. The sequence '*n*' is replaced by the text matched by the *n*-th regular subexpression enclosed between '\(' and '\)'.† The sequences '\u' and '\l' cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this

---

ters '~' and '&' also lose their special meanings in the replacement pattern of a substitute.

† In nested, parenthesized subexpressions, *n* is determined by counting occurrences of '\(' starting from the left.

character is a letter. The sequences '\U' and '\L' turn such conversion on, either until '\E' or '\e' is encountered, or until the end of the replacement pattern.

9. Option Descriptions

autoindent, ai default: noai

Eases the preparation of structured program text. At the beginning of each *append*, *change* or *insert* command, or when a new line is *opened* or created by an *append*, *change*, *insert*, or *substitute* operation within *open* or *visual* mode, *ex* looks at the line being appended after, the first line changed or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

All lines of text input from then on are justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line starts aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop, hit **^D**. The tab stops going backwards are defined at multiples of the *shiftwidth* option. You *cannot* backspace over the indent, except by sending an end-of-file with a **^D**.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '↑' and immediately followed by a **^D**. This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '0' followed by a **^D** repositions at the beginning but without retaining the previous indent.

Autoindent doesn't happen in *global* commands or when the input is not a terminal.

autoprint, ap default: ap

Causes the current line to be printed after each *delete*, *copy*, *join*, *move*, *substitute*, *t*, *undo* or shift command. This has the same effect as supplying a trailing 'p' to each such command. *Autoprint* is suppressed in globals, and only applies to the last of many commands on a line.

autowrite, aw default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and give a *next, rewind, stop, tag,* or *!* command, or a **^↑** (switch files) or **^]** (tag goto) command in *visual.* Note that the *edit* and *ex* commands do **not** autowrite, even if *autowrite* is set. If *autowrite* is set but you don't want it in a particular case, you can override it with the following commands:

- *edit* for *next*,
- *rewind!* for *rewind*,
- *stop!* for *stop*,
- *tag!* for *tag*,
- *shell* for *!*, and
- **:e #** and a **:ta!** command from within *visual.*

beautify, bf default: nobeautify

Causes all control characters except tab, newline and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. *Beautify* does not apply to command input.

directory, dir default: dir=/tmp

Specifies the directory in which *ex* places its buffer file. If this directory is not writable, *ex* exits abruptly when it fails to create its buffer there.

errorbells, eb default: noeb

Error messages are preceded by a bell.* If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht default: ht=8

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic default: noic

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

lisp default: nolisp

Autoindent indents appropriately for *lisp* code, and the **( )** { } **[[** and **]]** commands in *open* and *visual* are modified to have meaning for *lisp*.

list default: nolist

All printed lines are displayed (more) unambiguously, showing tabs and end-of-lines as in the *list* command.

magic default: magic for *ex* and *vi*†

If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only '↑' and '$' having special effects. In addition the metacharacters '~' and '&' of the replacement pattern are treated as normal characters. When *nomagic* is set, all the normal metacharacters may be made *magic* by preceding them with a '\'.

number, nu default: nonumber

Causes all output lines to be printed with their line numbers. In addition each input line is prompted for by supplying the line number it will have.

open default: open

If *noopen*, the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to open or visual mode.

optimize, opt default: optimize

Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para default: para=IPLPPPQPP LIbp

Specifies the paragraphs for the { and } operations in *open* and *visual.* The pairs of characters in the option's value are the names of the macros that start paragraphs.

prompt default: prompt

Command mode input is prompted for with a ':'.

redraw default: noredraw

The editor simulates an intelligent terminal on a dumb terminal. For example, during insertions in *visual,* the characters to the right of the cursor position are refreshed as each input character is typed. Useful only at very high speed.

---

* Bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb*.

† *Nomagic* for *edit*.

report default: report=5†

Specifies a threshold for feedback from commands. Any command that modifies more than the specified number of lines provides feedback as to the scope of its changes. For commands such as *global*, *open*, *undo*, and *visual,* which have potentially more far-reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.

scroll default: scroll=½ window

Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode *z* command (double the value of *scroll*).

sections default: sections=SHNHH HU

Specifies the section macros for the **[[** and **]]** operations in *open* and *visual.* The pairs of characters in the option's value are the names of the macros that start paragraphs.

shell, sh default: sh=/bin/sh

Gives the path name of the shell forked for the shell escape command '!', and by the *shell* command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw default: sw=8

Gives the width of a software tab stop, used in reverse tabbing with **^D**, when using *autoindent* to append text, and by the shift commands.

showmatch, sm default: nosm

In *open* and *visual* mode, when a **)** or } is typed, move the cursor to the matching **(** or { for one second if this matching character is on the screen. Extremely useful with *lisp.*

slowopen, slow terminal dependent

Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent. See *An Introduction to Display Editing with Vi* for more details.

tabstop, ts default: ts=8

The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

taglength, tl default: tl=0

Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags default: tags=tags /usr/lib/tags

A path of files to be used as tag files for the *tag* command. By default files called **tags** are searched for in the current directory and in /usr/lib (a master file for the entire system.)

term from environment TERM

The terminal type of the output device.

terse default: noterse

Shorter error diagnostics are produced for the experienced user.

---

† 2 for *edit.*

warn default: warn

Warn if there has been '[No write since last change]' before a '!' command escape.

window default: window=speed dependent

The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600

These are not true options; they set **window** only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws default: ws

Searches using the regular expressions in addressing wrap around past the end of the file if this is set.

wrapmargin, wm default: wm=0

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *An Introduction to Text Editing with Vi* for details.

writeany, wa default: nowa

Inhibit the checks normally made before *write* commands, allowing a write to any file permitted by the system protection mechanism.

10. Limitations

Editor limits that you are likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual,* 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250000 lines in the file is silently enforced.

An Introduction to Display Editing with Vi

ABSTRACT

Vi (visual) is a display-oriented interactive text editor. When you use *vi,* the screen of your terminal acts as a window into the file you are editing. You can see the changes you make to the file.

Using *vi* you can easily insert new text any place in the file. Most of the commands to *vi* move the cursor around in the file. The cursor moves forward and backward in units of characters, words, sentences and paragraphs. A small set of operators, such as **d** for delete and **c** for change, are combined with the motion commands to form operations such as "delete word" or "change paragraph", in a simple and natural way. This regularity and the mnemonic assignment of commands to keys makes the editor command set easy to remember and use.

Vi works on a large number of display terminals, and new terminals are easily driven after editing a terminal description file. *Vi* is well suited to run on intelligent terminals that can locally insert and delete lines and characters. But it also functions well on dumb terminals over slow phone lines. The editor makes allowance for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

You can also use the command set of *vi* on hardcopy terminals, storage tubes and "glass TTYs" using a one-line editing window; thus *vi's* command set is available on all terminals. The full command set of the more traditional, line oriented editor *ex* is available within *vi;* switching between the two modes of editing is simple.

May 18, 1984

An Introduction to Display Editing with Vi

1. GETTING STARTED

This document provides a quick introduction to *vi* (pronounced *vee-eye*). You should be running *vi* on a file you are familiar with while you are reading this. The first part of this document (sections 1 through 5) describes the basics of using *vi.* Some topics of special interest are presented in section 6, and some nitty-gritty details of how the editor functions are saved for section 7 to avoid cluttering the presentation here.

The short appendix lists each character and any special meanings this character has in *vi.*

1.1. Specifying Terminal Type

Before you can start *vi,* you must tell the system what kind of terminal you are using. Here is a (necessarily incomplete) list of terminal type codes. If your terminal does not appear here, you should consult with one of the staff members on your system to find out the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal can be created.

Code	Full name	Type
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent
t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

Suppose, for example, that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is '2621'. In this case you can use one of the following commands to tell the system the type of your terminal:

```
% setenv TERM 2621
```

This command works with the shell *csh* on all Plexus systems. If you are using the standard Sys3 shell, give the commands

```
$ TERM=2621
$ export TERM
```

If you want to arrange to have your terminal type set up automatically when you log in, you can use the *tset* program. For example, if you dial in on a *mime*, but often use hardwired ports, a typical line for your *.login* file (if you use *csh*) would be

setenv TERM `tset - -d mime`

or for your *.profile* file (if you use *sh*)

TERM=`tset - -d mime`

Tset knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. *Tset* is usually used to change the erase and kill characters, too.

1.2. Editing a File

After telling the system what kind of terminal you have, you should make a copy of a file you are familiar with, and run *vi* on this file. Pick a file that is longer than one screenful (usually 24 lines) so you can see the effect of scrolling and other commands. Give the command

% **vi** *name*

replacing *name* with the name of the copy file you just created. The screen should clear and the text of your file should appear on the screen.‡

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 the changes you make back into the original file.

1.4. Notational Conventions

In our examples, input that must be typed exactly as written here is presented in **bold face**. Text that should be replaced with appropriate input is given in *italics*. We represent special characters in SMALL CAPITALS.

1.5. Arrow Keys

The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys, these keys also work within the editor. If you don't have cursor positioning keys, or even if you do, you can use the **h j k** and **l** keys as cursor positioning keys (these are labeled with arrows on an *adm3a).**

(Particular note for the HP2621: on this terminal the function keys must be *shifted* to send to the machine; otherwise they only act locally. Unshifted use leaves the cursor positioned incorrectly.)

1.6. Special Characters: ESC, CR and DEL

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labeled ESC or ALT. It should be near the upper left corner of your terminal. Try hitting this key a few times. The editor rings the bell to indicate that it is in a

---

‡ If something else happens, you may have given the system an incorrect terminal type code. In this case, the editor may have just made a mess out of your screen (your file is probably okay). This sort of mess happens when the editor sends control codes for one kind of terminal to some other kind of terminal. To restart, hit the keys :q (colon and the q key) and then hit the RETURN key. This should get you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again. You may also have 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. If the editor doesn't seem to respond to the commands that you type here, try sending an interrupt to it by hitting the DEL or RUB key on your terminal, and then hitting the :q command again, followed by a carriage return.

* As we will see later, *h* moves back to the left (like control-h, which is a backspace), *j* moves down (in the same column), *k* moves up (in the same column), and *l* moves to the right.

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, so you can just hit it if you don't know what is going on until the editor rings the bell.

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

Another very useful key is the DEL or RUB key, which 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 string to be searched for. The cursor should now be positioned at the bottom line of the terminal after a '/' printed as a prompt. You can get the cursor back to the current position by hitting the DEL or RUB key; try this now.* From now on we will simply refer to hitting the DEL or RUB key as "sending an interrupt."**

The editor often echoes your commands on the last line of the terminal. If the cursor is 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.

1.7. 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 **ZZ** to the editor. This writes the contents of the editor's buffer back into the file you are editing, if you made any changes, and then leaves the editor. You can also end an editor session by giving the command **:q!**CR;† this is a dangerous but occasionally essential command that ends the editor session and discards all your changes. You need to know about this command in case you change the editor's copy of a file you wish only to look at. Be very careful not to give this command when you really want to save the changes you have made.

2. MOVING AROUND IN THE FILE

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 control and D keys at the same time, a control-D or '^D'. We will use this two character notation for referring to these control keys from now on. You may have a key labeled '^' on your terminal. This key will be represented as '↑' in this document; '^' is exclusively used as part of the '^x' notation for control characters.‡

As you know now if you tried hitting **^D**, this command scrolls down in the file. The **D** thus stands for down. Many editor commands are mnemonic and this makes them much easier to remember. For instance the command to scroll up is **^U**. Many dumb terminals can't scroll up at all, in which case hitting **^U** clears the screen and refreshes it with a line which is farther back in the file at the top.

If you want to see more of the file below where you are, you can hit **^E** to expose one more line at the bottom of the screen, leaving the cursor where it is. The command **^Y** (which is hopelessly non-mnemonic, but next to **^U** on the keyboard) exposes one more line at the top of the screen. ‡‡

---

‡ On smart terminals where possible, the editor quietly flashes the screen rather than ringing the bell.

* Backspacing over the '/' also cancels the search.

** On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.

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

‡ If you don't have a '^' key on your terminal then there is probably a key labeled '↑'; in any case these characters are one and the same.

‡‡ Not currently available.

Other ways to move around in the file include the keys **^F** and **^B**, which move forward and backward a page, keeping a couple of lines of continuity between screens.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting **^F** to move forward a page leaves you only a little context to look back at. Scrolling on the other hand leaves more context, and happens more smoothly. You can continue to read the text as scrolling is taking place.

2.2. Searching, Goto, and Previous Context

Another way to position yourself in the file is by giving the editor a string to search for. Type the character / followed by a string of characters terminated by CR. The editor positions the cursor at the next occurrence of this string. Try hitting **n** to then go to the next occurrence of this string. The character **?** searches backwards from where you are, and is otherwise like /.†

If the search string you give the editor is not present in the file, the editor prints a message to that effect on the last line of the screen, and the cursor is returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with an ↑. To match only at the end of a line, end the search string with a **$**. Thus /↑**search**CR searches for the word 'search' at the beginning of a line, and /**last$**CR searches for the word 'last' at the end of a line.*

The command **G**, when preceded by a number, positions the cursor at that line in the file. Thus **1G** moves the cursor to the first line of the file. If you give **G** no count, it moves to the end of the file.

If you are near the end of the file, and the last line is not at the bottom of the screen, the editor places 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 **^G**. The editor shows you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and what percentage of the buffer you have traversed. Try doing this now, and remember the number of the line you are on. Give a **G** command to get to the end. You can get back to your previous position by using the command `` (two back quotes). Try giving a search with / or **?** and then a `` to get back to where you were. If you accidentally hit **n** or any command that moves you far away from a context of interest, you can quickly get back by hitting ``.

2.3. Moving around on the Screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (4 or 5 keys with arrows going in each direction) try them and convince yourself that they work. (On certain terminals, they won't.) If you don't have working arrow keys, you can always use **h**, **j**, **k**, and **l**. 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. Notice that if you go off the bottom or top with these keys then the screen scrolls down (and up if possible) to bring a line at a time into view. The RETURN key has the same effect as the + key.

---

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

*Actually, the string you give to search for 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 by putting the command **:se nomagic**CR; in EXINIT in your environment. (More about *EXINIT* later.)

Vi also has commands to take you to the top, middle and bottom of the screen. **H** takes you to the top (home or highest) line on the screen. Try preceding it with a number as in **3H**. This takes you to the third line on the screen. Many *vi* commands take preceding numbers and do interesting things with them. Try **M**, which takes you to the middle line on the screen, and **L**, which takes you to the last (or lowest) line on the screen. **L** also takes counts, thus **5L** takes you to the fifth line from the bottom.

2.4. Moving within a Line

Now try picking a word on some line on the screen, not the first word on the line. Move the cursor using RETURN and - to be on the line where the word is. Try hitting the **w** key. This advances the cursor to the next word on the line. Try hitting the **b** key to back up words in the line. Try the **e** key, which advances you to the end of the current word rather than to the beginning of the next word. Also try SPACE (the space bar), which moves right one character and the BS (backspace or ^**H**) key, which moves left one character. The key **h** works as ^**H** does and is useful if you don't have a BS key. (Also, as noted just above, **l** moves to the right.)

If the line has punctuation in it, you may notice that that the **w** and **b** keys stop at each group of punctuation. You can also go back and forward by words without stopping at punctuation by using **W** and **B** rather than the lower case equivalents. Think of these as bigger words. 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 line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly hitting **w**.

2.5. Summary

SPACE	advance the cursor one position
^**B**	backwards to previous page
^**D**	scrolls down in the file
^**E**	exposes another line at the bottom (v3)
^**F**	forward to next page
^**G**	tell what is going on
^**H**	backspace the cursor
^**N**	next line, same column
^**P**	previous line, same column
^**U**	scrolls up in the file
^**Y**	exposes another line at the top (v3)
+	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

2.6. View

If you want to use the editor to look at a file, rather than to make changes, invoke it as *view* instead of *vi*. This sets the *readonly* option which prevents you from accidently overwriting the file.

3. MAKING SIMPLE CHANGES

3.1. Inserting

One of the most useful commands is the **i** (insert) command. After you type **i**, everything you type until you hit ESC is inserted into the file. Try this now; position yourself to some word in the file and try inserting text before this word. If you are on an dumb terminal it may seem, for a minute, that some of the characters in your line have been overwritten, but they reappear when you hit ESC.

Now try finding a word which can, but does not, end in an 's'. Position yourself at this word and type **e** (move to end of word), then **a** for append and then '**s**ESC' to terminate the insertion. This sequence of commands (**eas**ESC) can be used to pluralize a word easily.

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

You may often want to add new lines before or after some specific line in the file. Find a line where this makes sense and then 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, all text you type up to an ESC is inserted. This may amount to many lines or just one (see below).

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lower case key and the other is given by 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.

Whenever you are typing in text--whether you begin with **i**, **a**, **o**, **O**, **s**, **C**, etc.--you can give many lines of input or just a few characters. To type in more than one line of text, hit a RETURN at the middle of your input. A new line is created for text, and you can continue to type. If you are on a slow and dumb terminal the editor may choose to wait to redraw the tail of the screen, and let you type over the existing screen lines. This avoids the lengthy delay that would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen is fixed, and the missing lines reappear, when you hit ESC.

While you are inserting new text, you can use the characters you normally use at the system command level (usually **^H** or **#**) to backspace over the last character you typed, and the character you use to kill input lines (usually **@**, **^X**, or **^U**) to erase the input you have typed on the current line.† The character **^W** erases a whole word and leaves you after 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.

Notice also that you can't erase characters you didn't insert with the current insertion command, 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.

---

† In fact, the character **^H** (backspace) always works to erase the last input character here, regardless of what your erase character is.

3.2. Making Small Corrections

You can make small corrections in existing text quite easily. Find a single character that is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace (hit the BS key or ^H or even just **h**) or SPACE (using the space bar) until the cursor is on the character that is wrong. If the character is not needed then hit the **x** key; this deletes the character from the file. It is analogous to the way you **x** out characters when you make mistakes on a typewriter (except it's not as messy).

If the character is incorrect, you can replace it with the correct character by giving the command **r***c*, where *c* is replaced by the correct character. Finally if the character that is incorrect should be replaced by more than one character, give the command **s**, which substitutes a string of characters, ending with ESC. If a small number of characters are wrong, you can precede **s** with a count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

3.3. More Corrections: Operators

You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator. Try the command **dw** to delete a word. Try hitting **.** a few times. Notice that this repeats the effect of the **dw**. The command **.** repeats the last command that made a change. You can remember it by analogy with an ellipsis '...'.

Now try **db**. This deletes a word backwards, namely the preceding word. Try **d**SPACE. This deletes a single character, and is equivalent to the **x** command.

Another very useful operator is **c** or change. The command **cw** thus changes the text of a single word. You follow it by the replacement text ending with an ESC. Find a word that you can change to another, and try this now. Notice that the end of the text to be changed was marked with the character '$' so that you can see this as you are typing in the new material.

3.4. Operating on Lines

You often want to operate on lines rather than just words or letters. Find a line you want to delete, and type **dd**, the **d** operator twice. This deletes the line. If you are on a dumb terminal, the editor may just erase the line on the screen, replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen, which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Try repeating the **c** operator twice; this changes a whole line, erasing its previous contents and replacing them 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 also always tells you when a change you make affects text you cannot see.

---

† The command **S** is a convenient synonym for for **cc**, by analogy with **s**. Think of **S** as a substitute on lines, while **s** is a substitute on characters.

* Using the / search after a **d** is subtle. This move normally deletes characters from the current position to the point of the match. If you want to delete whole lines including the two points, give the pattern as /**pat**/**+0**, a line address.

3.5. Undoing

Now suppose that the last change 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 you made. Try this a few times, and give it twice in a row to notice that an **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 the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text that you delete, even if undo will not bring it back; see the section on recovering lost text below.

3.6. Summary

SPACE	advance the cursor one position
^H	backspace the cursor
^W	erase a word during an insert
erase	your erase (usually ^H or #), erases a character during an insert
kill	your kill (usually @, ^X, or ^U), kills the insert on this line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text
d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change

4. MOVING ABOUT: REARRANGING AND DUPLICATING TEXT

4.1. Low Level Character Motions

Now move the cursor to a line containing a punctuation or a bracketing character such as a parenthesis or a comma or period. Try the command **f***x* where *x* is this character. This command finds the next *x* character to the right of the cursor in the current line. Try then hitting 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. The **F** command, which is like **f**, searches backward. The **;** command repeats **F** also.

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

When working with the text of a single line, an **↑** moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Hitting '0' (zero) also works to move to the beginning of a line. Thus **$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 positions.* When the cursor is at a tab, it sits on the last of the several spaces that represent that tab. Try moving the cursor back

* This is settable by a command of the form **:se ts=***x*CR, where *x* is the number of columns per tabstop. This has effect on the screen representation within the editor.

and forth over tabs so you understand how this works.

On rare occasions, your file may have nonprinting characters in it. These characters are displayed in the same way they are represented in this document, that is, with a two character code, the first character of which is '^'. On the screen non-printing characters resemble a '^' character adjacent to another, but if you space or backspace over the character, you see that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes won't allow you to insert control characters, depending on the character and the setting of the *beautify* option. You can force a control character into the file by beginning an insert and then typing a **^V** before the control character. The **^V** quotes the following character, causing it to be inserted directly into the file.

4.2. Higher Level Text Objects

Sometimes the capacity to work with characters or words or lines is not enough; you need to be able to manipulate sentences, paragraphs, and sections. A sentence is defined to end at a '.', '!' or '?', 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 to the beginning of the previous and next sentences respectively. Thus the command **d)** deletes the rest of the current sentence; likewise **d(** deletes 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 of the current sentence.

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

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, i.e. the '.IP', '.LP', '.PP' and '.QP', '.P' and '.LI' macros.‡ Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.

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.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking 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.

4.3. Rearranging and Duplicating Text

The editor has a single unnamed buffer where the last deleted or changed text is saved.

The operator **y** yanks a copy of the object that follows into the unnamed buffer. The text can then be put back in the file with the commands **p** and **P**; **p** puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text that you yank forms a part of a line, or is an object such as a sentence, which

---

† 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.

‡ You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See section 6.2 for details. The '.bp' directive is also considered to start a paragraph.

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

Try the command **YP**. This makes a copy of the current line and leaves you on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** also makes a copy of the current line, and places it after the current line. You can give **Y** a count of lines to yank, and thus duplicate several lines; try **3YP**.

To move text within the buffer, you need to delete it in one place, and put it back in another. 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.

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

5. HIGH LEVEL COMMANDS

5.1. Writing, Quitting, Editing New Files

So far we have seen how to enter *vi* and to write out our file using either **ZZ** or **:w**CR. The first exits from the editor, (writing if changes were made), 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 from the editor without writing the changes. You can also reedit the same file (starting over) by giving the command **:e!**CR. These commands should be used only rarely, and with caution, as you cannot 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. If you have not written out your file before you try to do this, then the editor will tell you this, and delay editing the other file. You can then give the command **:w**CR to save your work and then the **:e** *name*CR command again, or carefully give the command **:e!** *name*CR, which edits the other file discarding 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**.

5.2. Escaping to a Shell

You can get to a shell to execute a single command by giving a *vi* command of the form **:!***cmd*CR. The system runs the single command *cmd* and when the command finishes, the editor asks you to hit a RETURN to continue. When you have finished looking at the output on the screen, hit RETURN. The editor clears the screen and redraws it. You can then continue editing. You can also give another : command when it asks you for a RETURN; in this case the screen is

not redrawn.

If you wish to execute more than one command in the shell, you can give the command **:sh**CR. This gives you a new shell, and when you finish with the shell, ending it by typing a **^D**, the editor clears the screen and continues.

On systems that support it, **^Z** suspends the editor and returns to the (top level) shell. When the editor is resumed, the screen is redrawn.

5.3. Marking and Returning

The command `` returned to the previous place after a motion of the cursor by a command such as /, **?** or **G**. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command **m***x*, where you should pick some letter for *x*, say 'a'. Then move the cursor to a different line (any way you like) and hit **`a**. The cursor returns to the place you marked. Marks last only until you edit another file.

Sometimes you mark a line in the middle but want to return to the beginning of the marked line; for example, when using operators such as **d** or **c** on marked lines. 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 ``.

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 **^L**, the ASCII form-feed character, to cause the screen to be refreshed.

On a dumb terminal, if there are @ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing **^R** to cause the editor to retype the screen, closing up these holes.

Finally, if you wish to place a certain line on the screen at the top middle or bottom of the screen, you can position the cursor to that line, and then give a **z** command. You should follow the **z** command with a RETURN 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.

6. SPECIAL TOPICS

6.1. Editing on Slow Terminals

When you are on a slow terminal, you will probably want to limit the amount of output 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 erases lines to @ when they are deleted 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. ('Se' is short for 'set'.) 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 noredrawCR.

The editor also makes editing more pleasant at low speed by starting 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 that is redrawn each time the screen is cleared by giving window sizes as argument to the commands that cause large screen motions:

: / ? [[ ]] ` '

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

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 RETURN, **.** 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 a DEL or RUBOUT as usual. 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 **^L**; or move or search again, ignoring the current state of the display.

See section 7.8 on *open* mode for another way to use the *vi* command **set** on slow terminals.

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
lisp	nolisp	**(** **{** **)** **}** commands deal with S-expressions
list	nolist	Tabs print as **^I**; end of lines marked with **$**
magic	nomagic	The characters **.** **[** and ***** are special in scans
number	nonu	Lines are displayed prefixed 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.

The options are of three kinds: numeric options, string options, and toggle options. You can set numeric and string options by a statement of the form

set *opt*=*val*

and toggle options can be set or unset by statements of one of the forms

set *opt*
set no*opt*

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

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

You can get a list of all options that you have changed by the command **:set**CR; you can get the value of a single option by the command **:set** *opt***?**CR. 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, e.g. **:se ai aw nu**CR.

Options set by the **set** command only last while you stay in the editor. You may want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands† that are to be run every time you start up *ex*, *edit*, or *vi*. A typical list includes a series of **set** commands. Put these commands all on one line, and separate them 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

If you use the standard v7 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.

6.3. Recovering Lost Lines

You might have a serious problem if you deleted a number of lines inadvertently. 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 previous 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 inspect (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 repeats the last change you made, but in this case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. Thus a sequence of the form

"1pu.u.u.

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

6.4. Recovering Lost Files

If the system crashes, you can recover the work you were doing to within a few changes. Change to the directory you were in when the system crashed and give a command of the form:

% **vi -r** *name*

replacing *name* with the name of the file you were editing. This recovers your work to a point near where you left off.†

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

---

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

† In rare cases, some of the lines of the file may be lost. The editor gives you the numbers of these lines and the text of the lines are replaced by the string 'LOST'. These lines almost always are among the last few you changed. You can either choose to discard the changes (if they are easy to remake) or replace the few lost lines by hand.

% vi -r

The invocation "*vi -r*" does not always list all saved files, but they can be recovered with "vi -r *name*" even if they are not listed.

6.5. Continuous Text Input

When you are typing in large amounts of text it is convenient to have lines broken near the right margin automatically. You can cause this to happen by giving the command **:se wm=***n*CR, where *n* is some number of columns. So, the command **:se wm=10**CR 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 and you wish to put it back together you can tell it to join the lines with **J**. You can give **J** 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. Kill the white space with **x** if you don't want it.

6.6. Features for Editing Programs

The editor has a number of commands for editing programs. Editing programs is different from editing text because in programs, an indentation structure must often be maintained. The editor has a *autoindent* facility to help you generate correctly indented programs.

To enable this facility you can 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 now start another line, notice that the editor supplies white 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 **^D** key to backtab over the supplied indentation.

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

For shifting lines in the program left and right, the operators **<** and **>** shift the lines you specify 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 and right.

If you have a complicated expression and wish to see if 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 that starts with }; this is sometimes useful with **y]]**.

6.7. Filtering Portions of the Buffer

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

6.8. Commands for Editing LISP†

If you are editing a LISP program you should set the option *lisp* by doing **:se lisp**CR. This changes the **(** and **)** commands to move backward and forward over s-expressions. The { and } commands are like **(** and **)** but don't stop at atoms. These can be used to skip to the next list, or through a comment quickly.

† The LISP features are not available on some v2 editors due to memory constraints.

The *autoindent* option works differently for LISP, supplying indent to align at the first argument to the last open list. If there is no such argument then the indent is two spaces more than the last level.

Another option useful for typing in LISP is *showmatch.* Try setting it with **:se sm**CR and then try typing a '(' some words and then a ')'. Notice that the cursor shows the position of the '(' which matches the ')' briefly. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

The editor also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the = operator. Try the command **=%** at the beginning of a function. This will realign all the lines of the function declaration.

When you are editing LISP, **[[** and **]]** advance and retreat to lines beginning with a **{**, and are useful for dealing with entire function definitions.

6.9. 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 kinds 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* ('left hand side') into *rhs* ('right hand side'). Restrictions are: *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 **^V**. (It may be necessary to double the **^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 :wq**^V^V**CR CR

which means that whenever you type **q**, the system will behave as though you had typed the four characters **:wq**CR. A **^V** is needed because without it the CR would end the : command, rather than becoming part of the *map* definition. Two **^V**'s are required because from within *vi*, two **^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 **^V^V^I** #

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

‡ Plexus currently does not support the macro feature.

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 ^T ^V␣␣␣␣

where ␣ is a blank. The ^V is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

7. 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. An abbreviation need not be a single keystroke, as it should be with a macro.

7.1. Abbreviations

The editor has a number of short commands that abbreviate longer commands introduced here. You can find these commands easily on the quick reference card. They often save a bit of typing and you can learn them as convenient.

8. NITTY-GRITTY DETAILS

8.1. Line Representation in the Display

The editor folds long logical lines onto many physical lines in the display. Commands that advance "lines" advance *logical* lines, not physical lines. A logical line is everything between user-input carriage returns. (So if you use autowrap and let the system break lines for you, the physical lines thus broken are not delimited by your carriage returns and thus do not correspond to *vi*'s idea of logical lines.)

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 puts only 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 often just clears 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 ^R command.

The editor can 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. However, this may not work well with long input lines on some dumb terminals. The display of long lines may be scrambled and the apparent cursor position may not be reliable; i.e., you may change something inadvertently. Be prepared to issue lots of ^L commands.

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.

---

‡‡ Not currently available.

† You can make long lines very easily by using 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 that are past the logical end of file.

8.2. Counts

Most *vi* commands accept 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 terminals that run at speeds greater than 1200 baud, the editor uses the full terminal screen. On terminals 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.

The editor uses this size when it clears and refills the screen after a search or other motion moves far from the edge of the current window. All the commands that 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 expands if you move off the top with a - or similar command or off the bottom with a command such as RETURN or **^D**. The window reverts to the last specified size the next time it is cleared and refilled.†

The scroll commands **^D** and **^U** likewise remember the amount of scroll last specified. Initially they use half the basic window size. The simple insert commands use a count to specify a repetition of the inserted text. Thus **10a**+----ESC inserts 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 that ignore any counts (such as **^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**RETURN advances five lines. You can also give a count to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you delete first one and then three words. You can then delete two more words with **2.**.

8.3. More File Manipulation Commands

The following table lists the file manipulation commands you can use when you are in *vi*. 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 ends 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 one 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, 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*. Actually, *n* may be any editor command not containing a space, usefully a scan like +/*pat* or +?*pat*. In forming 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 name. Thus if you try to do a **:e** and get a diagnostic that you haven't written the file,

† But not by a **^L**, which just redraws the screen as it is.

: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
:ta *tag*	edit file containing tag *tag*, at *tag*

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 finding out the lines that bound the range to be written using **^G**, and giving these numbers after the : and before the **w**, separated by ,'s (commas). For example, to write out lines 10 through 200 into a file called *pip*, issue the command

:10,200w pipCR

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 also read in the output from a command; just use **!***cmd* instead of a file name. So, for example, to read in the output of the *ls* command, type **:r !ls**.

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**. You can also respecify the list of files to be edited by giving the **:n** command a list of file names, or a pattern to be expanded as you would have given it in the initial *vi* command to the shell.

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 requires 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. (The tag feature is not available in some v2 editors.)

8.4. More about Searching for Strings

When you search for strings with / and **?**, the editor normally places you at the next or previous occurrence of the string. But sometimes you want to affect the lines *between* your current position and the next or previous occurrence of the pattern, without affecting the line containing the pattern. This is especially so if you are using an operator such as **d**, **c** or **y**. 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 just affects characters up to the match place, rather than whole lines; thus use "+0" to affect to the line that matches.

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

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should

set nomagic

in your EXINIT. In this case, only the characters ↑ and **$** are special in patterns. The character \ is also then special (as it is most everywhere in the system), and may be used to get at the extended pattern matching facility. Thus, with *nomagic* set, the character \ functions to turn on magic; with *magic* set, \ turns it off. You must also use a \ before a literal / 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, including 0

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

8.5. More about Input Mode

A number of characters can make corrections during input mode. These are summarized in the following table.

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

The most usual way of making corrections during input mode is to type **^H** to correct a single character, or to type one or more **^W**'s to back over incorrect words. If you use # as your erase character in the normal system, it works like **^H**.

Your system kill character, normally @, **^X** or **^U**, erases all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor erase characters that you did not insert with this 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. The command **A**, which appends at the end of the current line, is often useful for continuing.

If you wish to type in your erase or kill character (say # or @) then you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a **^V**. The **^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 any character and it will be inserted into the file at that point.*

* This is not quite true. The implementation of 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 ap-

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

When you are using *autoindent* you may wish to place a label at the left margin of a line. This is easy: type ↑ and then ^**D**. The editor moves the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a **0** followed immediately by a ^**D** to kill all the indent and not have it come back on the next line.

8.6. Upper Case only Terminals

If your terminal has only upper case, you can still use *vi* by using the normal system convention for typing on such a terminal. Characters that you normally type are converted to lower case, and you can type upper case letters by preceding them with a \. The characters { ~ } | ` are not available on such terminals, but you can escape them as \(\↑ \) \! \ ´. These characters are represented on the display in the same way they are typed.‡ ‡

8.7. Vi and Ex

Vi is actually one mode of editing within the editor *ex.* When you are running *vi* you can escape to the line oriented editor of *ex* by giving the command **Q**. All the : commands introduced above 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 rare instances, an internal error may occur in *vi.* In this case you get a diagnostic and are left in the command mode of *ex.* You can then save your work and quit, if you wish, by giving a command **x** after the *ex* prompt :, or you can reenter *vi* by giving *ex* a *vi* command.

Some tasks are easier in *ex* than in *vi.* Systematic changes in line oriented material are particularly easy. You can read the advanced editing documents for the editor *ed* to find out more about this style of editing. Experienced users often mix their use of *ex* command mode and *vi* command mode to speed the work they are doing.

8.8. Open Mode: Vi on Hardcopy Terminals and "Glass TTYs"

If you are on a hardcopy terminal or a terminal that does not have a cursor that can move off the bottom line, you can still use the command set of *vi,* but in a different mode. When you give a *vi* command, the editor tells you that it is using *open* mode. This name comes from the *open* command in *ex,* which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way in which the text is displayed.

In *open* mode the editor uses a single line window into the file, and moving backward and forward in the file causes new lines to be displayed, always below the current line. Two commands of *vi* work differently in *open:* **z** and ^**R**. The **z** command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the ^**R** command retypes the current line. On such terminals, the editor normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \'s to show you the characters that are deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks

---

pear 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 treats a following letter as a request for the corresponding control character. This is the only way to type ^**S** or ^**Q**, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

‡ The \ character you give does not echo until you type another key.

‡ Not available in all v2 editors due to memory constraints.

like again.

This mode is sometimes useful on very slow terminals that can support *vi* in the full screen mode. You can do this by entering *ex* and using an *open* command.

Appendix: Character Functions

This appendix lists characters and their functions in *vi*. The characters are presented in their order in the ASCII character set. Control characters come first, then most special characters, then the digits, upper and then lower case characters.

For each character we tell what it means as a command and during an insert, if applicable. Section numbers in parentheses refer to sections in this manual where the character is discussed; a 'f' after the section number means that the character is mentioned in a footnote.

^@ Not a command character. If typed as the first character of an insertion, it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted, the mechanism is not available. A **^@** cannot be part of the file due to the editor implementation (7.5f).

^A Unused.

^B Backward window. A count (e.g., **5^B**) means back up the specified number of windows. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).

^C Unused.

^D As a command, scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future **^D** and **^U** commands (2.1, 7.2). During an insert, backtabs over *autoindent* white space at the beginning of a line (6.6, 7.5); this white space cannot be backspaced over.

^E Unused.

^F Forward window. A count (e.g., **5^F**) means go forward the specified number of windows. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).

^G Equivalent to **:f**CR, printing the current file, whether it has been modified, the current line number, the number of lines in the file, and the percentage of the way through the file that you are.

^H (BS) Same as **left arrow**. (See **h**). During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different (3.1, 7.5).

^I (TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab characters, it rests at the last of the spaces that represent the tab. The spacing of tabstops is controlled by the *tabstop* option (4.1, 6.6).

^J (LF) Same as **down arrow** (see **j**).

^K Unused.

^L The ASCII formfeed character, this causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt (5.4, 7.2f).

^M (CR) A carriage return advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines (2.3). During an insert, a CR causes the insert to continue onto another line (3.1).

^N Same as **down arrow** (see **j**).

^O Unused.

^P Same as **up arrow** (see **k**).

^Q Not a command character. In input mode, **^Q** quotes the next character, the same as **^V**, except that some teletype drivers will eat the **^Q** so that the editor never sees it.

^R Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line (5.4, 7.2, 7.8).

^S Unused. Some teletype drivers use ^S to suspend output until ^Qis

^T Not a command character. During an insert, with *autoindent* set and at the beginning of the line, inserts *shiftwidth* white space.

^U Scrolls the screen up--the opposite of ^D, which scrolls down. Counts work as they do for ^D, and the previous scroll amount is common to both. On a dumb terminal, ^U will often necessitate clearing and redrawing the screen further back in the file (2.1, 7.2).

^V Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file (4.2, 7.5).

^W Not a command character. During an insert, backs up as b would in command mode; the deleted characters remain on the display (see ^H) (7.5).

^X Unused.

^Y Unused.

^Z If supported by the UNIX system, stops the editor, exiting to the top level shell. Same as **:stop**CR. Otherwise, unused.

^[(ESC) Cancels a partially formed command, such as a **z** when no following character has yet been given; terminates inputs on the last line (read by commands such as : / and ?); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type ESC**a**, and then material to be input; the material is then inserted correctly whether or not you were in insert mode when you started (1.5, 3.1, 7.5).

^\ Unused.

^] Searches for the word that is after the cursor as a tag. Equivalent to typing **:ta**, this word, and then a CR. Mnemonically, this command is "go right to" (7.3).

^↑ Equivalent to **:e #**CR, returning to the previous position in the last edited file, or editing a file that you specified if you got a 'No write since last change' diagnostic and do not want to have to type the file name again (7.3). (You have to do a **:w** before ^↑ will work in this case. If you do not wish to write the file you should do **:e!** #CR instead.)

^_ Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.

SPACE Same as **right arrow** (see **l**).

! An operator that processes lines from the buffer with reformatting commands. Follow **!** with the object to be processed, and then the command name terminated by CR. Doubling **!** and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the **!**. To read a file or the output of a command into the buffer use **:r** (7.3). To simply execute a command use **:!** (7.3).

" Precedes a named buffer specification. There are named buffers **1-9** used for saving deleted text. (4.3, 6.3)

In input mode, if this is your erase character, it deletes the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.

$** Moves to the end of the current line. If you **:se list**CR, then the end of each line will be shown by printing a **$ after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus **2$** advances to the end of the following line.

% Moves to the parenthesis or brace { } that balances the parenthesis or brace at the current cursor position.

& A synonym for **:&**CR, by analogy with the *ex* **&** command.

' When followed by a **'**, returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a-z**, returns to the line that was marked with this letter with a **m** command, at the first non-white character in the line. (2.2, 5.3). When used with an operator such as **d**, the operation takes place over complete lines; if you use **`**, the operation takes place from the exact marked place to the current cursor position within the line.

(Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a **.**, **!**, or **?**, followed by either the end of a line or by two spaces. Any number of closing **)**, **]**, **"**, and **'** characters may appear after the **and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[** below). A count advances that many sentences (4.2, 6.8).

) Advances to the beginning of a sentence. A count repeats the effect. See **(** above for the definition of a sentence (4.2, 6.8).

***** Unused.

+ Same as CR when used as a command.

, Reverse of the last **f**, **F**, **t**, or **T** command, looking the other way in the current line. Especially useful after hitting too many **;** characters. A count repeats the search.

- Retreats to the previous line at the first non-white character. This is the inverse of **+** and RETURN. If the line moved to is not on the screen, it is made visible, either by scrolling, or clearing and redrawing. If a large amount of scrolling would be required the screen is also cleared and redrawn, with the current line at the center (2.3).

. Repeats the last command that changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then hit **.** to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, **3.** deletes three words (3.3, 6.3, 7.2, 7.4).

/ Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line. The search begins when you hit CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; the search may then be terminated with a DEL or RUB, or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern you can force whole lines to be affected. To do this, give a pattern with a closing / and then an offset **+***n* or **-***n*.

To include the character / in the search string, you must escape it with a preceding \. A ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this can speed the search. A **$** at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available, see section 7.4; unless you set **nomagic** in your *.exrc* file, you must precede the characters ., [, *, and ~ in the search pattern with a \ to get them to work as literals (1.5, 2,2, 6.1, 7.2, 7.4).

0 Moves to the first character on the current line. Also used, in forming numbers, after an initial **1-9**.

1-9 Used to form numeric arguments to commands (2.3, 7.2).

: A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with an CR. The command is then executed. You can return to where you were by hitting DEL or RUB if you hit **:** accidentally (see primarily 6.2 and 7.3).

; Repeats the last single character find that used **f**, **F**, **t**, or **T**. A count repeats the scan (4.1).

< An operator that shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in **<<**. Counts are passed through to the basic object, thus **3<<** shifts three lines (6.6, 7.2).

= Reindents lines for LISP, as though they were typed in with *lisp* and *autoindent* set (6.8).

> An operator that shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in **>>**. Counts repeat the basic object (6.6, 7.2).

? Scans backwards, the opposite of /. See the / description above for details on scanning (2.2, 6.1, 7.4).

@ Unused.

A Appends at the end of line, a synonym for **$a** (7.2).

B Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect (2.4).

C Changes the rest of the text on the current line; a synonym for **c$**.

D Deletes the rest of the text on the current line; a synonym for **d$**.

E Moves forward to the end of a word, defined as blanks and non-blanks, like **B** and **W**. A count repeats the effect.

F Finds a single following character, backwards in the current line. A count repeats this search (4.1).

G Goes to the line number given as preceding argument, or the end of the file if no preceding count is given. The screen is redrawn if necessary with the new current line in the center (7.2).

H **Home arrow.** Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-white character on the line. If used as the target of an operator, full lines are affected (2.3, 3.2).

I Inserts at the beginning of a line; a synonym for **↑i**.

J Joins together lines, supplying appropriate white space: one space between words, two spaces after a ., and no spaces at all if the first character of the joined on line is). A count causes that many lines to be joined rather than the default two (6.5, 7.1f).

K Unused.

L Moves the cursor to the first non-white character of the last line on the screen. With a count, to the first non-white of the count'th line from the bottom. Operators affect whole lines when used with **L** (2.3).

M Moves the cursor to the middle line on the screen, at the first non-white position on the line (2.3).

N Scans for the next match of the last pattern given to / or **?**, but in the reverse direction; this is the reverse of **n**.

O Opens a new line above the current line and inputs text there up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better (3.1).

P Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "*x* to retrieve the contents of the buffer; buffers **1**-**9** contain deleted material (6.3).

Q Quits from *vi* to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the : commands; the editor supplies the : as a prompt (7.7).

R Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESC.

S Changes whole lines, a synonym for **cc**. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.

T Takes a single following character, locates the character *before* the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d** (4.1).

U Restores the current line to its state before you started changing it (3.5).

V Unused.

W Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect (2.4).

X Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

Y Yanks a copy of the current line into the unnamed buffer, to be put back by a later **p** or **P**; a very useful synonym for **yy**. A count yanks that many lines.

ZZ Exits the editor. (Same as **:x**CR.) If any changes have been made, the buffer is written out to the current file. Then the editor quits.

[[Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a '.NH' or '.SH' and also at lines that start with a formfeed ^**L**. Lines beginning with { also stop **[[**; this makes it useful for looking backwards, a function at a time, in C programs. If the option *lisp* is set, stops at each **(** at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects. (4.2, 6.1, 6.6, 7.2).

**** Unused.

]] Moves forward to a section boundary. See **[[** for a definition (4.2, 6.1, 6.6, 7.2).

↑ Moves to the first non-white position on the current line (4.4).

_ Unused.

`** When followed by a ` returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a-z, returns to the position that was marked with this letter with a **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use ´, the operation takes place over complete lines (2.2, 5.3).

a Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with an ESC (3.1, 7.2).

b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect (2.4).

c An operator that changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text that is to be changed is removed from the display but saved in the numeric named buffers. For example, **c)** makes the rest of the current sentence disappear, and creates a new empty line so you can begin to rewrite the sentence. If only part of the current line is affected, then the last character to be changed is marked with a **$**. A count causes that many objects to be affected; thus both **3c)** and **c3)** change the following three sentences (7.4).

d An operator that deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w** (3.3, 3.4, 4.1, 7.4).

e Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect (2.4, 3.1).

f Finds the first instance of the next character following the cursor on the current line. A count repeats the find (4.1).

g Unused.

h **Left arrow.** Moves the cursor one character to the left. Like the other arrow keys, either **h**, the **left arrow** key, or one of the synonyms (**^H**) has the same effect. On v2 editors, arrow keys on certain kinds of terminals (those which send escape sequences, such as vt52, c100, or hp) cannot be used. A count repeats the effect (3.1, 7.5).

i Inserts text before the cursor, otherwise like **a** (7.2).

j **Down arrow.** Moves the cursor one line down in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms include **^J** (linefeed) and **^N**.

k **Up arrow.** Moves the cursor one line up. **^P** is a synonym.

l **Right arrow.** Moves the cursor one character to the right. SPACE is a synonym.

m Marks the current position of the cursor in the mark register that is specified by the next character **a-z**. Return to this position or use with an operator with ` or ´ (5.3).

n Repeats the last / or **?** scanning commands (2.2).

o Opens new lines below the current line; otherwise like **O** (3.1).

p Puts text after/below the cursor; otherwise like **P** (6.3).

q Unused.

r Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above, which is the more generally useful iteration of **r** (3.2).

s Changes the single character at the cursor to the text that follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c** (3.2).

t Advances the cursor up to the character before the next character typed. Most useful with operators such as **d** and **c** to delete the characters up to a following character. You can use **.** to delete more if this doesn't delete enough the first time (4.1).

u Undoes the last change made to the current buffer. If repeated, alternates between these two states. Thus **u** is its own inverse. When used after an insert that inserted text on more than one line, the lines are saved in the numeric named buffers (3.5).

v Unused.

w Advances to the beginning of the next word, as defined by **b** (2.4).

x Deletes the single character under the cursor. With a count deletes deletes that many characters forward from the cursor position, but only on the current line (6.5).

y An operator, yanks the following object into the unnamed temporary buffer. Text can be recovered by a later **p** or **P** (7.4).

z Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, **.** the center of the screen, and **-** at the bottom of the screen. A count may be given after the **z** and before the following character to specify the new screen size for the redraw. A count before the **z** gives the number of the line to place in the center of the screen instead of the default current line. (5.4)

{ Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the *paragraphs* option, normally '.IP', '.LP', '.PP', '.QP' and '.bp'. A paragraph also begins after a completely empty line, and at each section boundary (see **[[** above) (4.2, 6.8, 7.6).

| Places the cursor on the character in the column specified by the count (7.1, 7.2).

} Advances to the beginning of the next paragraph. See **{** for the definition of paragraph (4.2, 6.8, 7.6).

~ Unused.

^? (DEL**)** Interrupts the editor, returning it to command accepting state (1.5, 7.5)