Ex Reference Manual
Version 3.5/2.13 – September, 1980

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

ABSTRACT

Ex a line oriented text editor, which 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*, the *Ex/edit Command Summary*, and a *Vi Quick Reference* card.

July 20, 1987

Ex Reference Manual
Version 3.5/2.13 – September, 1980

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

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 by changing the default settings of some of these options. To simplify the description which 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 will seek the description of the terminal in that file (rather than the default /etc/termcap.) If there is a variable EXINIT in the environment, then the editor will execute 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* will be 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 ...

The most common case edits a single file with no options, i.e.:

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 is used in recovering after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files. 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, which 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 initially. Other useful commands here are scanning patterns of the form ‘‘/pat’’ or line

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

† Brackets ‘[’ ‘]’ surround optional parameters here.

‡ Not available in all v2 editors due to memory constraints.

numbers, e.g. ‘‘+100’’ starting at line 100.

2. File manipulation

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 a reasonable action. If the current file is not *edited* then *ex* will 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.

For saving blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names *a* through *z.*‡

2.5. Read only

It is possible to 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*. It is possible to write, even while in read only mode, by indicating that you really know what you are doing. You can write to a different file, or can use the ! form of write, even while in read only mode.

---

* 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.

‡ It is also possible to refer to *A* through *Z;* the upper case buffers are the same as the lower but commands append to named buffers rather than replacing if upper case names are used.

3. Exceptional Conditions

3.1. Errors and interrupts

When errors occur *ex* (optionally) rings the terminal bell and, in any case, prints an error diagnostic. If the primary input is from a file, editor processing will terminate. If an interrupt signal is received, *ex* prints ''Interrupt'' and returns to its command level. If the primary input is a file, then *ex* will exit when this occurs.

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) will attempt 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. If you were editing the file *resume*, then you should change to the directory where you were when the crash occurred, giving the command

ex −r *resume*

After checking that the retrieved file is indeed ok, you can *write* it over the previous contents of that file.

You will normally get mail from the system telling you when a file has been saved after a crash. The command

ex −r

will print a list of the files which have been saved for you. (In the case of a hangup, the file will not appear in the list, although it can 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. This mode is left 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 will be 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'' will print the tenth line in the buffer while ''delete 5'' will delete five lines from the buffer, starting with the current line.

---

* As an 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.

Some commands take other information or parameters, this information always being given after the command name.‡

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

It is possible to give editor commands which 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 and map 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 which 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 undesirably large changes so that they may be quickly and easily reversed with an *undo*. After commands with more global effect such as *global* or *visual*, you will be 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 which 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

‡ Examples would be 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''.

** 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.

buffer. If all that is desired is 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.‡

´´ ´*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, which are *not,* however, part of the command.

abbreviate *word rhs* abbr: **ab**

Add the named abbreviation to the current list. When in input mode in visual, if *word* is typed as a complete word, it will be changed to *rhs*.

(**.**) **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
.

---

‡ The forms \/ and **?** scan using the last regular expression used in a scan; after a substitute **//** and **??** would 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 which expects none.

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
.

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. If a named *buffer* is specified by giving a letter, then the specified lines are saved in that buffer, or appended to it if an upper case letter is used.

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. After insuring that this file is sensible† the editor reads the file into its buffer.

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 occurred, the file is considered *edited.* If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be 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 which 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 of the current line.*

---

† I.e., that it is not a binary file such as a directory, a block or character special file other than */dev/tty,* a terminal, or a binary or executable file (as indicated by the first word).

‡ 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.

file *file*

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

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

First marks each line among those specified which 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 will be 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.

map *lhs rhs*

The *map* command is used to define macros for use in *visual* mode. *Lhs* should be a single character, or the sequence ‘‘#n’’, for n a digit, referring to function key *n*. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* had been typed. On terminals without function keys, you can type ‘‘#n’’. See section 6.9 of the ‘‘Introduction to Display Editing with Vi’’ for more details.

(**.**) **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 which 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 will be 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 v2 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 *buffer* is specified, then the last *deleted* or *yanked* text is restored.* By using a named buffer, text may be restored that was saved there at any previous time.

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 without complaint.

(**.**) **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. This is not a variant form of the command, rather a read specifying a *command* rather than a *filename;* 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. Except when you use *preserve* you will be notified by mail when a file is saved.

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.

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

---

* But no modifying commands may intervene between the *delete* or *yank* and the *put,* nor may lines be moved between files without using a named buffer.

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

‡ 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.

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* ; they 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. By typing an ‘y’ one can 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.

stop

Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form **stop**! is used. This commands is only available where supported by the teletype driver and operating system.

(**.** , **.**) **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 which can be used by the editor 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* was set.

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

unabbreviate *word* abbr: **una**

Delete *word* from the list of abbreviations.

---

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

‡ Not available in all v2 editors due to memory constraints.

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 it's pre-command value after an *undo.*

unmap *lhs*

The macro expansion associated by *map* for *lhs* is removed.

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

A synonym for the *global* command variant **g!**, running the specified *cmds* on each line which 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.

(**.**) **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 will be 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 which the system permits.

---

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

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

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

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 will be echoed. The current line is unchanged by this command.

If there has been ''[No write]'' of the buffer contents since the last change to the editing buffer, then a diagnostic will be printed before the command is executed as a warning. 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.

---

* 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. On some v2 editors, no *type* may be given.

(**.** , **.**) > *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 which 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, there being 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.*†

8.3. Basic regular expression summary

The following basic constructs are used to construct *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.

---

† To discern what is true with *nomagic* it suffices to remember that the only special characters in this case will be '↑' at the beginning of a regular expression, '$' at the end of a regular expression, and '\'. With *nomagic* the characters '~' and '&' also lose their special meanings related to the replacement pattern of a substitute.

↑ 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 which 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 collating 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 those characters which 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 which 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 which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character '~' may be used in a regular expression, and matches the text which 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 which the regular expression matched. The metacharacter '~' stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible 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 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

Can be used to ease 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.

---

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

If the user then types lines of text in, they will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop one can 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. In each case, there is an equivalent way of switching when autowrite is set to avoid the *autowrite* (*edit* for *next*, *rewind!* for .I 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 in not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible default: noedcompatible

Causes the presence of absence of **g** and **c** suffixes on substitute commands to be remembered, and to be toggled by repeating the suffices. The suffix **r** makes the substitution be as in the ~ command, instead of like &. ‡‡

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).

---

‡‡ Version 3 only.

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

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 will be 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. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a '\'.

mesg default: mesg

Causes write permission to be turned off to the terminal while you are in visual mode, if *nomesg* is set. ‡‡

number, nu default: nonumber

Causes all output lines to be printed with their line numbers. In addition each input line will be 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 which start paragraphs.

prompt default: prompt

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

redraw default: noredraw

The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. 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.

---

† *Nomagic* for *edit*.
‡‡ Version 3 only.

remap default: remap

If on, macros are repeatedly tried until they are unchanged. ‡‡ For example, if **o** is mapped to **O**, and **O** is mapped to **I**, then if *remap* is set, **o** will map to **I**, but if *noremap* is set, it will map to **O**.

report default: report=5†

Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide 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 options's value are the names of the macros which 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 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. ‡‡ A requested tag is searched for in the specified files, sequentially. By default (even in version 2) files called **tags** are searched for in the

---

‡‡ Version 3 only.

† 2 for *edit*.

‡‡ Version 3 only.

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.

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 but 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 will wrap around past the end of the file.

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 which the system protection mechanism will allow.

10. Limitations

Editor limits that the user is 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.

The *visual* implementation limits the number of macros defined with map to 32, and the total number of characters in macros to be less than 512.

Acknowledgments. Chuck Haley contributed greatly to the early development of *ex*. Bruce Englar encouraged the redesign which led to *ex* version 1. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

Ex/Edit Command Summary (Version 2.0)

Ex and *edit* are text editors, used for creating and modifying files of text on the UNIX computer system. *Edit* is a variant of *ex* with features designed to make it less complicated to learn and use. In terms of command syntax and effect the editors are essentially identical, and this command summary applies to both.

The summary is meant as a quick reference for users already acquainted with *edit* or *ex*. Fuller explanations of the editors are available in the documents *Edit: A Tutorial* (a self-teaching introduction) and the *Ex Reference Manual* (the comprehensive reference source for both *edit* and *ex*). Both of these writeups are available in the Computing Services Library.

In the examples included with the summary, commands and text entered by the user are printed in **boldface** to distinguish them from responses printed by the computer.

The Editor Buffer

In order to perform its tasks the editor sets aside a temporary work space, called a *buffer*, separate from the user's permanent file. Before starting to work on an existing file the editor makes a copy of it in the buffer, leaving the original untouched. All editing changes are made to the buffer copy, which must then be written back to the permanent file in order to update the old version. The buffer disappears at the end of the editing session.

Editing: Command and Text Input Modes

During an editing session there are two usual modes of operation: *command* mode and *text input* mode. (This disregards, for the moment, *open* and *visual* modes, discussed below.) In command mode, the editor issues a colon prompt (:) to show that it is ready to accept and execute a command. In text input mode, on the other hand, there is no prompt and the editor merely accepts text to be added to the buffer. Text input mode is initiated by the commands *append*, *insert*, and *change*, and is terminated by typing a period as the first and only character on a line.

Line Numbers and Command Syntax

The editor keeps track of lines of text in the buffer by numbering them consecutively starting with 1 and renumbering as lines are added or deleted. At any given time the editor is positioned at one of these lines; this position is called the *current line*. Generally, commands that change the contents of the buffer print the new current line at the end of their execution.

Most commands can be preceded by one or two line-number addresses which indicate the lines to be affected. If one number is given the command operates on that line only; if two, on an inclusive range of lines. Commands that can take line-number prefixes also assume default prefixes if none are given. The default assumed by each command is designed to make it convenient to use in many instances without any line-number prefix. For the most part, a command used without a prefix operates on the current line, though exceptions to this rule should be noted. The *print* command by itself, for instance, causes one line, the current line, to be printed at the terminal.

The summary shows the number of line addresses that can be prefixed to each command as well as the defaults assumed if they are omitted. For example, *(.,.)* means that up to 2 line-numbers may be given, and that if none is given the command operates on the current line. (In the address prefix notation, ''.'' stands for the current line and ''$'' stands for the last line of the buffer.) If no such notation appears, no line-number prefix may be used.

Some commands take trailing information; only the more important instances of this are mentioned in the summary.

Open and Visual Modes

Besides command and text input modes, *ex* and *edit* provide on some CRT terminals other modes of editing, *open* and *visual*. In these modes the cursor can be moved to individual words or characters in a line. The commands then given are very different from the standard editor commands; most do not appear on the screen when typed. *An Introduction to Display Editing with Vi* provides a full discussion.

Special Characters

Some characters take on special meanings when used in context searches and in patterns given to the *substitute* command. For *edit*, these are ''^'' and ''$'', meaning the beginning and end of a line, respectively. *Ex* has the following additional special characters:

. & * [ ] ~

To use one of the special characters as its simple graphic representation rather than with its special meaning, precede it by a backslash (\). The backslash always has a special meaning.

Name	Abbr	Description	Examples
(.)**append**	**a**	Begins text input mode, adding lines to the buffer after the line specified. Appending continues until ‘‘.’’ is typed alone at the beginning of a new line, followed by a carriage return. *0a* places lines at the beginning of the buffer.	:**a** **Three lines of text** **are added to the buffer** **after the current line.** **.** :
(.,.)**change**	**c**	Deletes indicated line(s) and initiates text input mode to replace them with new text which follows. New text is terminated the same way as with *append*.	:**5,6c** **Lines 5 and 6 are** **deleted and replaced by** **these three lines.** **.** :
(.,.)**copy** *addr*	**co**	Places a copy of the specified lines after the line indicated by *addr*. The example places a copy of lines 8 through 12, inclusive, after line 25.	:**8,12co 25** Last line copied is printed :
(.,.)**delete**	**d**	Removes lines from the buffer and prints the current line after the deletion.	:**13,15d** New current line is printed :
edit *file* **edit!** *file*	**e** **e!**	Clears the editor buffer and then copies into it the named *file*, which becomes the current file. This is a way of shifting to a different file without leaving the editor. The editor issues a warning message if this command is used before saving changes made to the file already in the buffer; using the form **e!** overrides this protective mechanism.	:**e ch10** No write since last change :**e! ch10** "ch10" 3 lines, 62 characters :
file *name*	**f**	If followed by a *name*, renames the current file to *name*. If used without *name*, prints the name of the current file.	:**f ch9** "ch9" [Modified] 3 lines ... :**f** "ch9" [Modified] 3 lines ... :
(1,$)**global** (1,$)**global!**	**g** **g!** or **v**	**global**/*pattern*/*commands* Searches the entire buffer (unless a smaller range is specified by line-number prefixes) and executes *commands* on every line with an expression matching *pattern*. The second form, abbreviated either **g!** or **v**, executes *commands* on lines that *do not* contain the expression *pattern*.	:**g/nonsense/d** :
(.)**insert**	**i**	Inserts new lines of text immediately before the specified line. Differs from *append* only in that text is placed before, rather than after, the indicated line. In other words, **1i** has the same effect as **0a**.	:**1i** **These lines of text will** **be added prior to line 1.** **.** :
(.,.+1)**join**	**j**	Join lines together, adjusting white space (spaces and tabs) as necessary.	:**2,5j** Resulting line is printed :

Name	Abbr	Description	Examples
(.,.)**list**	**l**	Prints lines in a more unambiguous way than the *print* command does. The end of a line, for example, is marked with a ‘‘$’’, and tabs printed as ‘‘^I’’.	:**9l** This is line 9$:
(.,.)**move** *addr*	**m**	Moves the specified lines to a position after the line indicated by *addr*.	:**12,15m 25** New current line is printed :
(.,.)**number**	**nu**	Prints each line preceded by its buffer line number.	:**nu** 10 This is line 10 :
(.)**open**	**o**	Too involved to discuss here, but if you enter open mode accidentally, press the ESC key followed by **q** to get back into normal editor command mode. *Edit* is designed to prevent accidental use of the open command.	
preserve	**pre**	Saves a copy of the current buffer contents as though the system had just crashed. This is for use in an emergency when a *write* command has failed and you don’t know how else to save your work.†	:**preserve** File preserved. :
(.,.)**print**	**p**	Prints the text of line(s).	:**+2,+3p** The second and third lines after the current line :
quit **quit!**	**q** **q!**	Ends the editing session. You will receive a warning if you have changed the buffer since last writing its contents to the file. In this event you must either type **w** to write, or type **q!** to exit from the editor without saving your changes.	:**q** No write since last change :**q!** %
(.)**read** *file*	**r**	Places a copy of *file* in the buffer after the specified line. Address 0 is permissible and causes the copy of *file* to be placed at the beginning of the buffer. The *read* command does not erase any text already in the buffer. If no line number is specified, *file* is placed after the current line.	:**0r newfile** "newfile" 5 lines, 86 characters :
recover *file*	**rec**	Retrieves a copy of the editor buffer after a system crash, editor crash, phone line disconnection, or *preserve* command.	
(.,.)**substitute**	**s**	**substitute**/*pattern*/*replacement*/ **substitute**/*pattern*/*replacement*/**gc** Replaces the first occurrence of *pattern* on a line with *replacement*. Including a **g** after the command changes all occurrences of *pattern* on the line. The **c** option allows the user to confirm each substitution before it is made; see the manual for details.	:**3p** Line 3 contains a misstake :**s/misstake/mistake/** Line 3 contains a mistake :

† Seek assistance from a consultant as soon as possible after saving a file with the *preserve* command, because the file is saved on system storage space for only one week.

Name	Abbr	Description	Examples
undo	**u**	Reverses the changes made in the buffer by the last buffer-editing command. Note that this example contains a notification about the number of lines affected.	:**1,15d** 15 lines deleted new line number 1 is printed :**u** 15 more lines in file ... old line number 1 is printed :
(1,$)**write** *file* (1,$)**write!** *file*	**w** **w!**	Copies data from the buffer onto a permanent file. If no *file* is named, the current filename is used. The file is automatically created if it does not yet exist. A response containing the number of lines and characters in the file indicates that the write has been completed successfully. The editor's built-in protections against overwriting existing files will in some circumstances inhibit a write. The form **w!** forces the write, confirming that an existing file is to be overwritten.	:**w** "file7" 64 lines, 1122 characters :**w file8** "file8" File exists ... :**w! file8** "file8" 64 lines, 1122 characters :
(.)**z** *count*	**z**	Prints a screen full of text starting with the line indicated; or, if *count* is specified, prints that number of lines. Variants of the *z* command are described in the manual.	
!*command*		Executes the remainder of the line after **!** as a UNIX command. The buffer is unchanged by this, and control is returned to the editor when the execution of *command* is complete.	:**!date** Fri Jun 9 12:15:11 PDT 1978 ! :
control-d		Prints the next *scroll* of text, normally half of a screen. See the manual for details of the *scroll* option.	
(.+1)<cr>		An address alone followed by a carriage return causes the line to be printed. A carriage return by itself prints the line following the current line.	:<cr> the line after the current line :
/*pattern*/		Searches for the next line in which *pattern* occurs and prints it.	:**/This pattern/** This pattern next occurs here. :
//		Repeats the most recent search.	:// This pattern also occurs here. :
?*pattern***?**		Searches in the reverse direction for *pattern*.	
??		Repeats the most recent search, moving in the reverse direction through the buffer.	

Ex differences – version 1.1 to 2.0

This sheet summarizes the differences between the old version 1.1 of *ex* and the new version 2.0. The new *ex* is available as the standard *ex* on the VAX on the 5th floor of Evans, and as a new and experimental version in /usr/new on the Cory 11/70. It will soon be available in /usr/new on the Computer Center and Ingres Machines. Send problems over the Berkeley network to ‘‘vax:bill’’.

Changes to existing features

Options.

The options *editany, edited, fork, hush, printall* and *sticky* have been deleted because of lack of use. The *notify* option has been renamed *report.*

The *home* option will soon be superseded by the environment feature of version 7 UNIX and has been deleted. Similarly the *mode* option is superseded by the *umask* of version 7 and has also been deleted.

The *visualmessage* option has been deleted; use ‘‘mesg n’’ at the system command level to inhibit interconsole messages.

The *iul* option is replaced by a more general mechanism which allows portions of the buffer to be processed through specified commands; you can get *iul* processing on lines 1 to 100 of a file by doing ‘‘1,100!iul’’. This replaces the lines 1 to 100 by the output of an *iul* command, giving the command these lines as input.

Invocation

The options **–o, –n** and **–p** have been deleted.

Filename formation

The alternate filename is now represented as ‘#’ rather than ‘`’, since ‘`’ is a shell metacharacter. The editor now uses a shell to expand filenames containing shell metacharacters. If you use *csh,* then you can use all the shell metasyntax in forming new filenames, including home directory references with ‘~’ and variables you define in *.cshrc* using ‘$’.

Character representation

Control characters are now represented as ‘^*x*’; thus a control X is printed as ‘^X’; the delete character is represented ‘^?’.

Command changes

There have been major changes to open/visual (incompatible ones are described below).

It is no longer possible to discard changes by repeating the *quit* command twice. You must use the variant form **quit!** to get out of the editor discarding changes. Similarly the variant forms **e!** and **next!** must be used to edit a new file or the next file without saving changes you have made.†

A new form of the ‘!’ shell escape replaces the *expand* and *tabulate* commands. Thus the command ‘‘1,10expand’’ of the old version is replaced by ‘‘1,10!expand’’ in the new. Note also that the command abbreviation **ta** no longer refers to the *tabulate* command, which has been deleted, but rather refers to the new *tag* command.

The format of the *args* command has been changed; the files are no longer numbered, rather the entire argument list is always printed with the current file name enclosed by ‘[’ and ‘]’.

The format of the *file* command output has been changed; the editor says ‘[Not edited]’ in the rare case that this is true rather than saying ‘[Edited]’. The command also gives the percentage of the way into

† Less useful are **rewind!** and **recover!**.

the buffer that the current line is.

The format of the *set* command has been improved; ‘‘set all’’ now prints in a three column format. The commands ‘‘set %’’, ‘‘set !’’ and ‘‘set `’’ have been deleted. The command ‘‘set’’ now prints in a one line format rather than down the screen.

The commands *echo, expand, help, reset, sync, tabulate* and *xpand* have been deleted.

Changes to open and visual

A large number of changes have been made to open and visual; we summarize only the most noticeable ones here. See the attached reference card for more information, and (even if you know how to use visual already) you should look at *An Introduction to Text Editing with Vi*. We do not discuss any of the new commands in visual here.†

The delete line command is now **dd** rather than \\ (\\ no longer works!.) In fact, **d** and other operators can now operate on lines; thus **dL** deletes to the last line on the screen. The shift commands < and > are now operators, thus << and >> now have the effect that < and > used to have.

The command **v** has been deleted; only its synonym **z** remains. The **K** operation has been moved to **m**; **K** has no meaning in the new version. The **^S** operation has been deleted, but **^G** does a sync, and also prints some information. The **^W** operation has been deleted (use **B**). The **#**, **@** and **^X** operations have been deleted. To delete to the beginning of the line use **d0**; the commands and **x** and **X** are similar to **#**.

During inputs, **^W** backs up like **b** rather than **B**.

Terminal support has been vastly improved; the editor will now drive most any display terminal, using all terminal features such as cursor addressing, clear to end of line, insert and delete line and insert and delete character. To help performance on slow terminals some options are now set based on the intelligence and speed of the terminal; in particular, the default window size is 1/2 a full screen at 300 baud, or 2/3 of a full screen at 1200 baud.

Ex changes – Version 2.0 to 3.1

This update describes the new features and changes which have been made in converting from version 2.0 to 3.1 of *ex*. Each change is marked with the first version where it appeared. Versions 2.1 through 2.7 were implemented by Bill Joy; Mark Horton produced versions 2.8, 2.9 and 3.1 and is maintaining the current version.

Update to Ex Reference Manual

Command line options

2.1 Invoking *ex* via

```
% ex –l
```

now sets the *lisp* and *showmatch* options. This is suitable for invocations from within *lisp*(1). If you don’t like *showmatch* you can still use ‘‘ex –l’’ to get *lisp* set, just put the command ‘‘set noshowmatch’’ in your *.exrc* file.

3.1 Invoking *ex* with an argument **–w***n* sets the value of the *window* option before starting; this is particularly suitable when invoking *vi*, thus

```
% vi –w5 ex2.0-3.1
```

edits the file with a 5 line initial window.

2.9 The text after a **+** on the command line is no longer limited to being a line number, but can be any single command. This generality is also available within the editor on *edit* and *next* commands (but no blanks are allowed in such commands.) A very useful form of this option is exemplified by

† It is now possible to edit with the focus of the editing being *visual* using a command *vi* rather than *ex* on the command line, and using a new **:** command from within visual to run command mode commands.

```
% vi +/main more.c
```

Command addressing

2.9 The address form **%** is short for ‘‘1,$’’.

Commands

2.2 The editor now ignores a ‘‘:’’ in front of commands, so you can say ‘‘:wq’’ even in command mode.

2.8 The *global* command now does something sensible when you say

```
g/pat/
```

printing all lines containing *pat;* before this printed the first line after each line containing *pat.* The trailing / may be omitted here.

3.1 New commands *map* and *unmap* have been added which are used with macros in *visual* mode. These are described below.

3.1 The *next* command now admits an argument of the form ‘‘+command’’ as described above.

3.1 The *substitute* command, given no arguments, now repeats the previous *substitute,* just as ‘‘&’’ does. This is easier to type.

2.8 The substititute command ‘‘s/str’’, omitting the delimiter on the regular expression, now deletes ‘‘str’’; previously this was an error.

2.9 During pattern searches of a *tag* command, the editor uses *nomagic* mode; previously a funny, undocumented mode of searching was used.

3.1 The editor requires that the tag names in the *tags* file be sorted.

2.3 The command *P* is a synonym for *print.*

2.9 The default starting address for *z* is **.+1**. If *z* is followed by a number, then this number is remembered by setting the *scroll* option.

2.9 A command consisting of only two addresses, e.g. ‘‘1,10’’ now causes all the lines to be printed, rather than just the last line.

Options

2.8 *Autowrite* (which can be abbreviated *aw*) is an on/off option, off by default. If you set this option, then the editor will perform *write* commands if the current file is modified and you give a *next,* ^^ (in *visual), !* or *tag* commands, (and noticeably not before *edit* commands.) Note that there is an equivalent way to do the command with autowrite set without the write in each case: *edit, :e #, shell* and *tag!* do not *autowrite.*

3.1 A new option *edcompatible* causes the presence or absence of *g* and *c* suffixes on *substitute* commands to be remembered, and to be toggled by repeating the suffices. The suffix *r* makes the substitution be as in the ~ command instead of like &.

2.8 There is a new *hardtabs* option, which is numeric and defaults to 8. Changing this to, say, 4, tells *ex* that either you system expands tabs to every 4 spaces, or your terminal has hardware tabs set every 4 spaces.

3.1 There is a new boolean option *mapinput* which is described with the macro facility for *visual* below.

2.9 Whether *ex* prompts for commands now depends only on the setting of the *prompt* variable so that you can say ‘‘set prompt’’ inside *script*(1) and get *ex* to prompt.

Environment enquiries

3.1 *Ex* will now execute initial commands from the EXINIT environment variable rather than *.exrc* if it find such a variable.

2.9 *Ex* will read the terminal description from the TERMCAP environment variable if the description there is the one for the TERM in the environment. TERMCAP may still be a pathname (starting with a /; in that case this will be used as the termcap file rather than /etc/termcap, and the terminal description will be sought there.)

Vi Tutorial Update

Change in default option settings.

3.1 The default setting for the *magic* option is now *magic.* Thus the characters

. [* ~

are special in scanning patterns in *vi.* You should

set nomagic

in your *.exrc* if you don't use these regularly. This makes *vi* default like *ex.* In a related change, *beautify* is no longer the default for *vi.*

Line wrap around

2.4 The *w W b B e* and *E* operations in visual now wrap around line boundaries. Thus a sequence of enough **w** commands will get to any word after the current position in the file, and **b**'s will back up to any previous position. Thus these are more like the sentence operations *(* and *)*. (You still can't back around line boundaries during inserts however.)

2.3 The */* and *?* searches now find the next or previous instance of the searched for string. Previously, they would not find strings on the current line. Thus you can move to the right on the current line by typing ''/pref<ESC>'' where ''pref'' is a prefix of the word you wish to move to, and delete to a following string ''str'' by doing ''d/str<ESC>'', whether it is on the same or a succeeding line. (Previously the command ''d/pat/'' deleted lines through the next line containing ''pat''. This can be accomplished now by the somewhat unusual command ''d/pat/0'', which is short for ''d/pat/+0''. The point is that whole lines are affected if the search patter only specifies a line, and using address arithmetic makes the pattern only specify a line.)

3.1 Arrow keys on terminals that send more than 1 character now work. Home up keys are supported as are the four directions. (Note that the HP 2621 will turn on function key labels, and even then you have to hold shift down to use the arrow keys. To avoid turning on the labels, and to give up the function keys, use terminal type 2621nl instead of 2621.)

Macros

3.1 A parameterless macro facility is included from visual. This facility lets you say that when you type a particular key, you really mean some longer sequence of keys. It is useful when you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two flavors of macros:

a) Put the macro body in a buffer register, say x. Then type @x to invoke it. @ may be followed by another @ to repeat the last macro. This allows macros up to 512 chars.

b) Use the map command from command mode (typically in the *.exrc* file) as follows:

map *lhs rhs*

where *lhs* will be mapped to *rhs*. There are restrictions: *lhs* should be 1-keystroke (either 1 char or 1 function key) since it must be entered within 1 second. The *lhs* can be no longer than 10 chars, the *rhs* no longer than 100. To get space, tab, ''|'', or newline into *lhs* or *rhs,* escape them with ctrl V. (It may be necessary to escape the ctrl V with ctrl V if the map command is given from visual mode.) Spaces and tabs inside the *rhs* need not be escaped.

For example, to make the Q key write and exit the editor, you can do

:map Q :wq^VCR

which means that whenever you type 'Q', it will be as though you had typed the four characters :wqCR. The control V is needed because without it the return would end the colon command.

For 1 shot macros it is best to put the macro in a buffer register and map a key to '@r', since this will allow the macro to be edited.

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 char # sequence, if the terminal has function keys. For terminals without function keys, the sequence #*x* means function key *x*, for any digit *x*. As a special case, on terminals without function keys, the #x sequence need not be typed within one second. The character # can be changed by using a macro in the usual way:

map ^V^I #

to use tab, for example. (This won't affect the map command, which still uses #, but just the invocation from visual mode.) The undo command will undo an entire macro call as a unit.

3.1 New commands in visual: ^Y and ^E. These scroll the screen up and down 1 line, respectively. They can be given counts, controlling the number of lines the screen is scrolled. They differ from ^U and ^D in that the cursor stays over the same line in the buffer it was over before rather than staying in the same place on the screen. (^Y on a dumb terminal with a full screen will redraw the screen moving the cursor up a few lines.) If you're looking for mnemonic value in the names, try this: Y is right next to U and E is right next to D.

Miscellaneous

3.1 In visual: '&' is a synonym for ':&<cr>'.

2.2 In input mode in open and visual ^V (like tenex) is now equivalent to ^Q (which is reminiscent of ITS) superquoting the next character.

2.8 The j, k, and l keys now move the cursor down, up, and right, respectively, in visual mode, as they used to do (and always did on some terminals). This is to avoid the creeping of these keys into the map descriptions of terminals and to compensate for the lack of arrow keys on some terminals.

2.5 The $ command now sets the column for future cursor motions to effective infinity. Thus a '$' followed by up/down cursor motions moves at the right margin of each line.

2.9 The way window sizes and scrolling commands are based on the options window and scroll has been rearranged. All command mode scrolling commands (z and ctrl D) are based on scroll: ^D moves scroll lines, z moves scroll*2 lines. Everything in visual (^D, ^U, ^F, ^B, z, window sizes in general) are based on the window option. The defaults are arranged so that everything seems as before, but on hardcopy terminals at 300 baud the default for scroll is 11 instead of 6.

Ex changes – Version 3.1 to 3.5

This update describes the new features and changes which have been made in converting from version 3.1 to 3.5 of *ex*. Each change is marked with the first version where it appeared.

Update to Ex Reference Manual

Command line options

3.4 A new command called *view* has been created. *View* is just like *vi* but it sets *readonly*.

3.4 The encryption code from the v7 editor is now part of *ex*. You can invoke *ex* with the –x option and it will ask for a key, as *ed*. The *ed x* command (to enter encryption mode from within the editor) is not available. This feature may not be available in all instances of *ex* due to memory limitations.

Commands

3.4 Provisions to handle the new process stopping features of the Berkeley TTY driver have been added. A new command, *stop*, takes you out of the editor cleanly and efficiently, returning you to the shell. Resuming the editor puts you back in command or visual mode, as appropriate. If *autowrite* is set

and there are outstanding changes, a write is done first unless you say ‘‘stop!’’.

3.4 A

:vi <file>

command from visual mode is now treated the same as a

:edit <file> or :ex <file>

command. The meaning of the *vi* command from *ex* command mode is not affected.

3.3 A new command mode command *xit* (abbreviated *x*) has been added. This is the same as *wq* but will not bother to write if there have been no changes to the file.

Options

3.4 A read only mode now lets you guarantee you won’t clobber your file by accident. You can set the on/off option *readonly* (*ro*), and writes will fail unless you use an ! after the write. Commands such as *x*, *ZZ*, the autowrite option, and in general anything that writes is affected. This option is turned on if you invoke ex with the –R flag.

3.4 The *wrapmargin* option is now usable. The way it works has been completely revamped. Now if you go past the margin (even in the middle of a word) the entire word is erased and rewritten on the next line. This changes the semantics of the number given to wrapmargin. 0 still means off. Any other number is still a distance from the right edge of the screen, but this location is now the right edge of the area where wraps can take place, instead of the left edge. Wrapmargin now behaves much like fill/nojustify mode in *nroff*.

3.3 The options *w300*, *w1200*, and *w9600* can be set. They are synonyms for *window*, but only apply at 300, 1200, or 9600 baud, respectively. Thus you can specify you want a 12 line window at 300 baud and a 23 line window at 1200 baud in your EXINIT with

:set w300=12 w1200=23

3.3 The new option *timeout* (default on) causes macros to time out after one second. Turn it off and they will wait forever. This is useful if you want multi character macros, but if your terminal sends escape sequences for arrow keys, it will be necessary to hit escape twice to get a beep.

3.3 The new option *remap* (default on) causes the editor to attempt to map the result of a macro mapping again until the mapping fails. This makes it possible, say, to map q to # and #1 to something else and get q1 mapped to something else. Turning it off makes it possible to map ^L to l and map ^R to ^L without having ^R map to l.

3.3 The new (string) valued option *tags* allows you to specify a list of tag files, similar to the ‘‘path’’ variable of csh. The files are separated by spaces (which are entered preceded by a backslash) and are searched left to right. The default value is ‘‘tags /usr/lib/tags’’, which has the same effect as before. It is recommended that ‘‘tags’’ always be the first entry. On Ernie CoVax, /usr/lib/tags contains entries for the system defined library procedures from section 3 of the manual.

Environment enquiries

3.4 The editor now adopts the convention that a null string in the environment is the same as not being set. This applies to TERM, TERMCAP, and EXINIT.

Vi Tutorial Update

Deleted features

3.3 The ‘‘q’’ command from visual no longer works at all. You must use ‘‘Q’’ to get to ex command mode. The ‘‘q’’ command was deleted because of user complaints about hitting it by accident too often.

3.5 The provisions for changing the window size with a numeric prefix argument to certain visual commands have been deleted. The correct way to change the window size is to use the z command, for example z5<cr> to change the window to 5 lines.

3.3 The option "mapinput" is dead. It has been replaced by a much more powerful mechanism: ‘‘:map!’’.

Change in default option settings

3.3 The default window sizes have been changed. At 300 baud the window is now 8 lines (it was 1/2 the screen size). At 1200 baud the window is now 16 lines (it was 2/3 the screen size, which was usually also 16 for a typical 24 line CRT). At 9600 baud the window is still the full screen size. Any baud rate less than 1200 behaves like 300, any over 1200 like 9600. This change makes *vi* more usable on a large screen at slow speeds.

Vi commands

3.3 The command ‘‘ZZ’’ from vi is the same as ‘‘:x<cr>’’. This is the recommended way to leave the editor. Z must be typed twice to avoid hitting it accidently.

3.4 The command ^Z is the same as ‘‘:stop<cr>’’. Note that if you have an arrow key that sends ^Z the stop function will take priority over the arrow function. If you have your ‘‘susp’’ character set to something besides ^Z, that key will be honored as well.

3.3 It is now possible from visual to string several search expressions together separated by semicolons the same as command mode. For example, you can say

```
/foo/;/bar
```

from visual and it will move to the first ‘‘bar’’ after the next ‘‘foo’’. This also works within one line.

3.3 ^R is now the same as ^L on terminals where the right arrow key sends ^L (This includes the Televideo 912/920 and the ADM 31 terminals.)

3.4 The visual page motion commands ^F and ^B now treat any preceding counts as number of pages to move, instead of changes to the window size. That is, 2^F moves forward 2 pages.

Macros

3.3 The ‘‘mapinput’’ mechanism of version 3.1 has been replaced by a more powerful mechanism. An ‘‘!’’ can follow the word ‘‘**map**’’ in the *map* command. **Map!**’ed macros only apply during input mode, while **map**’ed macros only apply during command mode. Using ‘‘**map**’’ or ‘‘**map!**’’ by itself produces a listing of macros in the corresponding mode.

3.4 A word abbreviation mode is now available. You can define abbreviations with the *abbreviate* command

```
:abbr foo find outer otter
```

which maps ‘‘foo’’ to ‘‘find outer otter’’. Abbreviations can be turned off with the *unabbreviate* command. The syntax of these commands is identical to the *map* and *unmap* commands, except that the ! forms do not exist. Abbreviations are considered when in visual input mode only, and only affect whole words typed in, using the conservative definition. (Thus ‘‘foobar’’ will not be mapped as it would using ‘‘map!’’) Abbreviate and unabbreviate can be abbreviated to ‘‘ab’’ and ‘‘una’’, respectively.

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

ABSTRACT

This narrative introduction to the use of the text editor *edit* assumes no prior familiarity with computers or with text editing. Its aim is to lead the beginning UNIX† user through the fundamental steps of writing and revising a file of text. Edit, a version of the text editor *ex*, was designed to provide an informative environment for new and casual users.

We welcome comments and suggestions about this tutorial and the UNIX documentation in general.

September 1981

†UNIX is a trademark of Bell Laboratories.

Contents

Introduction 3

Session 1 4
- Making contact with UNIX 4
- Logging in 4
- Asking for *edit* 4
- The ‘‘Command not found’’ message 5
- A summary 5
- Entering text 5
- Messages from *edit* 5
- Text input mode 6
- Making corrections 6
- Writing text to disk 7
- Signing off 7

Session 2 8
- Adding more text to the file 8
- Interrupt 8
- Making corrections 8
- Listing what’s in the buffer (p) 9
- Finding things in the buffer 9
- The current line 10
- Numbering lines (nu) 10
- Substitute command (s) 10
- Another way to list what’s in the buffer (z) 11
- Saving the modified text 12

Session 3 13
- Bringing text into the buffer (e) 13
- Moving text in the buffer (m) 13
- Copying lines (copy) 14
- Deleting lines (d) 14
- A word or two of caution 15
- Undo (u) to the rescue 15
- More about the dot (.) and buffer end ($) 16
- Moving around in the buffer (+ and −) 16
- Changing lines (c) 17

Session 4 18
- Making commands global (g) 18
- More about searching and substituting 19
- Special characters 19
- Issuing UNIX commands from the editor 20
- Filenames and file manipulation 20
- The file (f) command 20
- Reading additional files (r) 21
- Writing parts of the buffer 21
- Recovering files 21
- Other recovery techniques 21
- Further reading and other information 22
- Using *ex* 22

Index 23

Introduction

Text editing using a terminal connected to a computer allows you to create, modify, and print text easily. A *text editor* is a program that assists you as you create and modify text. The text editor you will learn here is named *edit*. Creating text using edit is as easy as typing it on an electric typewriter. Modifying text involves telling the text editor what you want to add, change, or delete. You can review your text by typing a command to print the file contents as they were entered by you. Another program, a text formatter, rearranges your text for you into ''finished form.'' This document does not discuss the use of a text formatter.

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions which lead you through the fundamental steps of creating and revising text. After scanning each lesson and before beginning the next, you should practice the examples at a terminal to get a feeling for the actual process of text editing. If you set aside some time for experimentation, 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 ''Communicating with UNIX'' 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 (1) your terminal and its special keys, (2) the login procedure, (3) 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 the computer performs in order to accomplish a specific task. The tasks must be specific, such as balancing your checkbook or editing your text. A general task, such as working for world peace, is something we can do, but not something we can write programs to do.

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

edit — *edit* is the name of the UNIX text editor you will be learning to use, and 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 named *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, end the session, 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, yet 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, which you will learn in Session 1.

filename — Filenames are used to distinguish one file from another, serving the same purpose as the labels of manila folders in a file cabinet. In order to write or access information in a file, you use the name of that file in a UNIX command, and the system will automatically locate 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, and information is recorded on it.

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

Session 1

Making contact with UNIX

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 for the two ways you can make contact: on a terminal that is directly linked to the computer, or over a telephone line where the computer answers your call.

Directly-linked terminals

Turn on your terminal and press the RETURN key. You are now ready to login.

Dial-up terminals

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. You are now ready to login.

Logging in

The message inviting you to login is:

:login:

Type your login name, which identifies you to UNIX, on the same line as the login message, and press 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 the RETURN key)*

(In the examples, input you would type appears in **bold face** to distinguish it from the responses from UNIX.)

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

Password: *(type your password and press RETURN)*

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

Login incorrect.
:login:

in which case you should start the login process anew. Assuming that you have successfully logged in, UNIX will print the message of the day and eventually will present you with a % at the beginning of a fresh line. The % 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 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 the RETURN key and wait for edit's response:

% **edit text** *(followed by a RETURN)*
"text" No such file or directory
:

If you typed the command correctly, you will now be 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. It was unable to find such a file, since ‘‘text’’ is a new file we are about to create. Edit confirms this with the line:

"text" No such file or directory

On the next line appears edit’s prompt ‘‘:’’, announcing that you are in *command mode* and edit expects a command from you. You may now begin to create the new file.

The ‘‘Command not found’’ message

If you misspelled edit by typing, say, ‘‘editor’’, your request would be handled as follows:

% **editor**
editor: Command 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, and you may then enter the correct command.

A summary

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

:login: **susan**
Password:
... A Message of General Interest ...
% **edit text**
"text" No such file or directory
:

Entering text

You may now begin entering text into the buffer. This is done by *appending* (or adding) 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, since you are adding text to nothing you are creating text. Most edit commands have two forms: a word that suggests 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 at first, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is ‘‘append’’, and it may be abbreviated ‘‘a’’. Type **append** and press the RETURN key.

% **edit text**
: **append**

Messages from *edit*

If you make a mistake in entering a command and type something that edit does not recognize, edit will respond 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 will 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 stops sending you a prompt. You will not receive any prompts or error messages while in text input mode. 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, and *when you wish to stop entering text lines you should type a period as the only character on the line and press the RETURN key.* When you type the period and press RETURN, you signal that you want to stop appending text, and edit responds by allowing you to exit text input mode and reenter command mode. Edit will again prompt 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 will believe you want to remain in append mode and will not let you out. As this can be very frustrating, be sure to type **only** the period and the RETURN key.

This is a good place 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 RETURN that gets you out of append mode.

Making corrections

If you have read a general introduction to UNIX, such as ‘‘Communicating with UNIX’’, you will recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character as many times as there are characters you want to erase.

The usual erase character is the backspace (control-H), and you can correct typing errors in the line you are typing by holding down the CTRL key and typing the ‘‘H’’ key. If you try typing control-H you will notice that the terminal backspaces in the line you are on. You can backspace over your error, and then type what you want to be the rest of the line.

If you make a bad start in a line and would like to begin again, you can either backspace to the beginning of the line or you can use the at-sign ‘‘@’’ to erase everything on the line:

```
Text edtiing is strange, but@
Text editing is strange, but nice.
```

When you type the at-sign (@), you erase the entire line typed so far and are given a fresh line to type on. You may immediately begin to retype the line. This, unfortunately, does not help after you type the line and press RETURN. To make corrections in lines that have been completed, it is necessary to use the editing commands covered in the next session and those that follow.

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 safe-keeping 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 will last only until the end of the editing session. 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 will copy the contents of the buffer to a disk file. If the file does not yet exist, a new file will be created automatically and the presence of a ''[New file]'' will be noted. The newly-created file will be given the name specified when you entered the editor, in this case ''text''. To confirm that the disk file has been successfully written, edit will repeat the filename and give 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 will still be 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 will print

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 ready to quit the session with edit. To do this we type ''quit'' (or ''q'') and press 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 the command

%**logout**

This will end your session with UNIX, and will ready the terminal for the next user. It is always important to type **logout** 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)
... A Message of General Interest ...
%
```

When you indicate you want to edit, you can specify the name of the file you worked on last time. This will start edit working, and it will fetch 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 will repeat its name and tell you the number of lines and characters it 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, and indicates this by its prompt character, the colon (:). 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 more to the end of your text you may do so by using the append command to enter text input mode. When ‘‘append’’ is the first command of your editing session, the lines you enter are placed at the end of the buffer. 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.
.
```

You may recall that once you enter append mode using the ‘‘a’’ (or ‘‘append’’) command, you need to type a line containing only a period (.) to exit append mode.

Interrupt

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

```
Interrupt
:
```

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

Making corrections

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. Refer either to the last few pages of Session 1 or to ‘‘Communicating with UNIX’’ if you need to review the procedures for making a correction. The most important idea to remember is that erasing a character or cancelling a line must be done before you press the RETURN key.

Listing what's in the buffer (p)

Having appended text to what you wrote in Session 1, you might want to see all the lines 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 ''$'' is a special symbol designating the last line of the buffer, and ''p'' (or **print**) is the command to print from line 1 to the end of the buffer. The command ''1,$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 accidentally type a character that can't be printed, which can be done by striking a key while the CTRL key is pressed. 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 pressing the CTRL key while typing ''a''. This can happen on many terminals because the CTRL key and the ''A'' key are beside each other. If your finger presses between the two keys, control-A results. When asked to print the contents of the buffer, edit would display

it does illustr^Ate the editor.

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 which 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, a deliberate error so we can learn to make corrections. 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. The way to tell edit to search for something is to type it inside slash marks:

:**/thiss/**

By typing **/thiss/** and pressing RETURN, you instruct edit to search for ''thiss''. If you ask edit to look for a pattern of characters which it cannot find in the buffer, it will respond ''Pattern not found''. When edit finds the characters ''thiss'', it will print the line of text for your inspection:

And thiss is some more text.

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

---

†The numeral ''one'' is the top left-most key, and should not be confused with the letter ''el''.

The current line

Edit keeps track of the line in the buffer where it is located at all times during an editing session. In general, the line that has been most recently printed, entered, or changed is the current location in the buffer. The editor is prepared to make changes at the current location in the buffer, unless you direct it to another location.

In particular, when you bring a file into the buffer, you will be located at the last line in the file, where the editor left off copying the lines from the file to the buffer. If your first editing command is ‘‘append’’, the lines you enter are added to the end of the file, after the current line — the last line in the file.

You can refer to your current location 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, you can type **.=** and press RETURN, and edit will respond with the line number:

```
:.=
2
```

If you type the number of any line and press RETURN, edit will position you at that line and print its contents:

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

You should experiment with these commands to gain experience in using them to make changes.

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.
```

Note 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 their corresponding line numbers.

Substitute command (s)

Now that you have found the misspelled word, you can 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 will use the abbreviation ‘‘s’’ to reduce the chance of mistyping the substitute command. This command will instruct 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 edit to make a 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 it to* /

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

Substitute pattern match failed

indicating that your instructions could not be carried out. When edit does find the characters that you want to change, it will make the substitution and automatically print 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) will be searched for the characters ‘‘thiss’’, and when the first exact match is found, ‘‘thiss’’ will be 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 will assume that we mean to change the line where we are currently located (‘‘.’’). In this case, the command without a line number would have produced the same result because we were already located at the line we wished to change.

For another illustration of the substitute command, let us choose the line:

Text editing is strange, but nice.

You can make this line a bit more positive by taking out the characters ‘‘strange, but ’’ so the line reads:

Text editing is nice.

A command that will first position edit at the desired line and then make the substitution is:

: **/strange/s/strange, but //**

What we have done here is combine our search with our substitution. Such combinations are perfectly legal, and speed up editing quite a bit once you get used to them. That is, you do not necessarily have to use line numbers to identify a line to edit. Instead, you may identify the line you 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 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 that the space is to be taken out, your line will read:

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 may 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 will start with line 1 and continue 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, type the command

: **z**

If no starting line number is given for the z command, printing will start 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 (:quit! overrides)
:
```

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 did during the editing session since you typed the latest write command. Because in this lesson we have not written to disk at all, everything we have done would have been lost if edit had obeyed the **q** command. If you did not want to save the work done during this editing session, you would have to type ‘‘q!’’ or (‘‘quit!’’) to confirm that you indeed wanted to end the session immediately, leaving the file as it was after the most recent ‘‘write’’ command. However, since you want to save what you have edited, you need to type:

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

and then follow with the commands to quit and logout:

```
:q
% logout
```

and hang up the phone or turn off the terminal when UNIX asks for a name. Terminals connected to the port selector will stop after the logout command, and pressing keys on the keyboard will do nothing.

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. You should try to login without looking at the notes, but if you must then by all means do.

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

% **edit text**

or simply

% **edit**

Both ways get you in contact with edit, but the first way will bring 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 typing:

:**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 will 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. The first two examples are for illustration only, though after you have read this Session you are welcome to return to them for practice. The command

:**2,4m$**

directs edit to move lines 2, 3, and 4 to the end of the buffer ($). The format for the move command is that you 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. So,

:**1,3m6**

would instruct edit to move lines 1 through 3 (inclusive) to a location after line 6 in the buffer. To move only one line, say, line 4, to a location in the buffer after line 5, the command would be ‘‘4m5’’.

Let’s move some text using the command:

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

After executing a command that moves more than one line of the buffer, edit tells how many lines were affected by the move and prints the last moved line for your inspection. If you want to see more than just the last line, you can then 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.

You 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/**

(Do not type both examples here!) The problem with combining context searching with the move command is that your chance of making a typing error in such a long command is greater than if you type 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:

:**2,5copy $**

makes a copy of lines 2 through 5, 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, which is ''This is text added in Session 2.'' if you typed the commands suggested so far.

:**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 line containing the indicated text, starting its search at the current line and moving line by line until it finds the text. Once you are sure that you have correctly specified the line you want to delete, you can enter the delete (d) command. In this case it is not necessary 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.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
It doesn't mean much here, but

To delete both lines 2 and 3:

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

you type

:**2,3d**
2 lines deleted

which specifies the range of lines from 2 to 3, and the operation on those lines — ‘‘d’’ for delete. If you delete more than one line you will receive a message telling you the number of lines deleted, as indicated in the example above.

The previous example assumes 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:

:**/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 will search 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 RETURN to send the command on its way.

Undo (u) to the rescue

The **undo (u)** command has the ability to reverse the effects of the last command that changed the buffer. 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 which 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 that do not change the buffer, such as print. Most importantly, the **only** command that can be reversed by undo is the last ‘‘undo-able’’ command you typed. You can use control-H and @ to change commands while you are typing them, and undo to reverse the effect of the commands after you have typed them and pressed RETURN.

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. Typing undo at this moment will reverse 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; we type dot (and only a dot) on a line and press 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:

:**.=**

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 ‘‘$’’, then, 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 –)

When you are editing 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. You can move forward in the buffer similarly:

+2p

instructs edit to print the line that is 2 ahead of your 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 will be placed after the last line in the buffer ($), and the original lines referred to by ‘‘–1’’ and ‘‘+2’’ remain where they are.

Try typing only ‘‘–’’; you will 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 RETURN alone on a line is the equivalent of typing ‘‘+1p’’; it will move you one line ahead in the buffer and print 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 will remind you that you are at the end of the buffer:

At end-of-file

or

Not that many lines in buffer

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

Nonzero address required on this command

or

Negative address – first buffer line is 1

The number associated with a buffer line is the line’s ‘‘address’’, in that it can be used to locate the line.

Changing lines (c)

You can also 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 to accept the text that will replace them. Let's say you 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 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 you type RETURN to end the change command, edit notifies you if more than one line will be changed and places you in text input mode. Any text typed on the following lines will be inserted into the position where lines were deleted by the change command. **You will 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. The final section suggests sources of further information.

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 for the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word ‘‘text’’ to the word ‘‘material’’ the command would be 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* will be changed (the normal result of the substitute command). The ‘‘g’’ at the end of the command is independent of the ‘‘g’’ at the beginning. You may give a command such as:

:**5s/text/material/g**

to change every instance of ‘‘text’’ in line 5 alone. Further, neither command will change ‘‘text’’ to ‘‘material’’ if ‘‘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**

You should be careful 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

will delete 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 will print 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 file 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 ‘‘text’’ to ‘‘texts’’ we may type either

:**/text/s/text/texts/**

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

: **/text/s//texts/**

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.

(You should note that the search command found the characters ‘‘does’’ in the word ‘‘doesn’t’’ in the first search request.) 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 characters you are searching for.

It is also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute command, using the same search and replacement patterns. After altering the current line by typing

: **s/text/texts/**

you type

: **/text/&**

or simply

: **//&**

to make the same change on the next line in the buffer containing the characters ‘‘text’’.

Special characters

Two characters have special meanings when used in specifying searches: ‘‘$’’ and ‘‘^’’. ‘‘$’’ is taken by the editor to mean ‘‘end of the line’’ and is used to identify strings that occur at the end of a line.

: **g/text.$/s//material./p**

tells the editor to search for all lines ending in ‘‘text.’’ (and nothing else, not even a blank space), to change each final ‘‘text.’’ to ‘‘material.’’, 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 lose temporarily 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 rather than the character ‘‘$’’. The backslash retains its special significance unless it is preceded by another backslash.

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 shell command. If the buffer contents have not been written since the last change, a warning will be printed before the command is executed:

```
[No write since last change]
```

The editor prints a ‘‘!’’ when the command is completed. The tutorial ‘‘Communicating with UNIX’’ 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. If you are editing a file named ‘‘draft3’’ having 283 lines in it, you can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
"chapter3" [new file] 283 lines, 8698 characters
```

The current filename remembered by the editor *will not be changed as a result of the write command.* Thus, if the next write command does not specify a name, edit will write onto the current file (‘‘draft3’’) 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, the number of lines in the buffer, and the percent of the distance through the file your current location is.

```
: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 will tell you that the file has been ‘‘[Modified]’’. After you save the changes by writing onto a disk file, the buffer will no longer be 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 at a specified location, essentially copying new lines between two existing lines. To use it, specify the line after which the new

text will be placed, the **read (r)** command, and then the name of the file. If you have a file named ‘‘example’’, the command

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

reads the file ‘‘example’’ and adds it to the buffer after the last line. The current filename is not changed by the read command.

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. Your original file is unaffected by your command to write part of the buffer to another file. Edit still remembers whether you have saved changes to the buffer in your original file or not.

Recovering files

Although it does not happen very often, there are times UNIX stops working because of some malfunction. This situation is known as a *crash*. Under most circumstances, edit’s crash recovery feature is able to save work to within a few lines of changes before a crash (or an accidental phone hang up). If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login that gives the name of the recovered file. To recover the file, enter the editor and type the command **recover** (**rec**), followed by the name of the lost file. For example, to recover the buffer for an edit session involving the file ‘‘chap6’’, the command is:

```
: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. For best results, write the buffer to a new file temporarily so you can examine it without risk to the original file. Unfortunately, you cannot use the recover command to retrieve a file you removed using the shell command **rm**.

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 wait for the reply,

```
File preserved.
```

If you do not receive this reply, seek help immediately. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. If the reply is ‘‘File preserved.’’ you can leave the editor (or logout) to remedy the situation. After a preserve, you can use the recover command once the problem has been corrected, or the **–r** option of the edit command if you leave the editor and want to return.

If you make an undesirable change to the buffer and type a write command before discovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the undo command. After fixing the damaged buffer, you can again write the file to disk.

Further reading and other information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex*. These lessons are intended to introduce you to the editor and its more commonly-used commands. We have not covered all of the editor's commands, but a selection of commands that should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in the *Ex Reference Manual*, which is applicable to both *ex* and *edit*. The manual is available from the Computing Services Library, 218 Evans Hall. One way to become familiar with the manual is to begin by reading the description of commands that you already know.

Using *ex*

As you become more experienced with using the editor, you may still find that edit continues to meet your needs. However, should you become interested in using ex, it is easy to switch. To begin an editing session with ex, use the name **ex** in your command instead of **edit.**

Edit commands work the same way in ex, but the editing environment is somewhat different. You should be aware of a few differences that exist between the two versions of the editor. In edit, only the characters ''^'', ''$'', and ''\'' have special meanings in searching the buffer or indicating characters to be changed by a substitute command. Several additional characters have special meanings in ex, as described in the *Ex Reference Manual.* Another feature of the edit environment prevents users from accidently entering two alternative modes of editing, *open* and *visual*, in which the editor behaves quite differently from normal command mode. If you are using ex and the editor behaves strangely, you may have accidently entered open mode by typing ''o''. Type the ESC key and then a ''Q'' to get out of open or visual mode and back into the regular editor command mode. The document *An Introduction to Display Editing with Vi* provides a full discussion of visual mode.

Index

addressing, *see* line numbers
ampersand, 20
append mode, 6-7
append (a) command, 6, 7, 9
‘‘At end of file’’ (message), 18
backslash (\), 21
buffer, 3
caret (^), 10, 20
change (c) command, 18
command mode, 5-6
‘‘Command not found’’ (message), 6
context search, 10-12, 19-21
control characters (‘‘^’’ notation), 10
control-H, 7
copy (co) command, 15
corrections, 7, 16
current filename, 21
current line (.), 11, 17
delete (d) command, 15-16
dial-up, 5
disk, 3
documentation, 3, 23
dollar ($), 10, 11, 17, 20-21
dot (**.**) 11, 17
edit (text editor), 3, 5, 23
edit (e) command, 5, 9, 14
editing commands:
    append (a), 6, 7, 9
    change (c), 18
    copy (co), 15
    delete (d), 15-16
    edit (text editor), 3, 5, 23
    edit (e), 5, 9, 14
    file (f), 21-22
    global (g), 19
    move (m), 14-15
    number (nu), 11
    preserve (pre), 22-23
    print (p), 10
    quit (q), 8, 13
    read (r), 22
    recover (rec), 22, 23
    substitute (s), 11-12, 19, 20
    undo (u), 16-17, 23
    write (w), 8, 13, 21, 22
    z, 12-13
    ! (shell escape), 21
    $=, 17
    +, 17
    –, 17
    //, 12, 20
    ??, 20
    ., 11, 17
    .=, 11, 17
entering text, 3, 6-7
erasing
    characters (^H), 7
    lines (@), 7
error corrections, 7, 16
ex (text editor), 23
Ex Reference Manual, 23
exclamation (!), 21
file, 3
file (f) command, 21-22
file recovery, 22-23
filename, 3, 21
global (g) command, 19
input mode, 6-7
Interrupt (message), 9
line numbers, *see also* current line
    dollar sign ($), 10, 11, 17
    dot (.), 11, 17
    relative (+ and –), 17
list, 10
logging in, 4-6
logging out, 8
‘‘Login incorrect’’ (message), 5
minus (–), 17
move (m) command, 14-15
‘‘Negative address—first buffer line is 1’’ (message), 18
‘‘No current filename’’ (message), 8
‘‘No such file or directory’’ (message), 5, 6
‘‘No write since last change’’ (message), 21
non-printing characters, 10
‘‘Nonzero address required’’ (message), 18
‘‘Not an editor command’’ (message), 6
‘‘Not that many lines in buffer’’ (message), 18
number (nu) command, 11
password, 5
period (.), 11, 17
plus (+), 17
preserve (pre) command, 22-23
print (p) command, 10
program, 3
prompts
    % (UNIX), 5
    : (edit), 5, 6, 7
      (append), 7
question (?), 20
quit (q) command, 8, 13
read (r) command, 22
recover (rec) command, 22, 23

recovery, *see* file recovery
references, 3, 23
remove (rm) command, 21, 22
reverse command effects (undo), 16-17, 23
searching, 10-12, 19-21
shell, 21
shell escape (!), 21
slash (/), 11-12, 20
special characters (^, $, \), 10, 11, 17, 20-21
substitute (s) command, 11-12, 19, 20
terminals, 4-5
text input mode, 7
undo (u) command, 16-17, 23
UNIX, 3
write (w) command, 8, 13, 21, 22
z command, 12-13

Index

addressing, *see* line numbers
append mode, 4
backslash (\), 18
buffer, 2
command mode, 4
context search, 8, 10, 13, 18
control characters (‘‘^’’ notation), 8
control-d, 6
current filename, 19, 20
current line (.), 9, 15
diagnostic messages, 4
disk, 2
documentation, 21
edit (to begin editing session), 3, 7
editing commands:
append (a), 4, 7
change (c), 16
copy (co), 13
delete (d), 13-14
edit (e), 12
file (f), 19
global (g), 18-19
move (m), 12-13
number (nu), 9
preserve (pre), 20-21
print (p), 8
quit (q), 5, 11
quit! (q!), 11
read (r), 20
recover (rec), 20
substitute (s), 9-10, 17, 18
undo (u), 14, 17
write (w), 5-6, 11, 19-20
z, 11

! (shell escape), 19
$= , 15
+, 15
–, 15
//, 8, 18
??, 18
., 9, 15
.=, 9, 15
erasing
characters (#), 8
lines (@), 8
ex (text editor), 21
Ex Reference Manual, 21
file, 1
file recovery, 20
filename, 2
Interrupt (message), 7
line numbers, *see also* current line
dollar sign ($), 8, 12-13, 15
dot (.), 9, 15
relative (+ and –), 15, 16
logging out, 6
login procedure, 2
‘‘magic’’ characters, 21
non-printing characters, 8
‘‘not found’’ (message), 3
program, 1
recovery *see* file recovery
shell, 18
shell escape (!), 19
special characters (^, $, \), 18
text input mode, 4
UNIX, 1

An Introduction to Display Editing with Vi

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

ABSTRACT

Vi (visual) is a display oriented interactive text editor. When using *vi* the screen of your terminal acts as a window into the file which you are editing. Changes which you make to the file are reflected in what you see.

Using *vi* you can insert new text any place in the file quite easily. Most of the commands to *vi* move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences and paragraphs. A small set of operators, like **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 to use.

Vi will work on a large number of display terminals, and new terminals are easily driven after editing a terminal description file. While it is advantageous to have an intelligent terminal which can locally insert and delete lines and characters from the display, the editor will function quite 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.

It is also possible to use the command set of *vi* on hardcopy terminals, storage tubes and ‘‘glass tty’s’’ 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;* it is quite simple to switch between the two modes of editing.

July 20, 1987

An Introduction to Display Editing with Vi

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

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.

There is also a short appendix here, which gives for each character the special meanings which this character has in *vi*. Attached to this document should be a quick reference card. This card summarizes the commands of *vi* in a very compact format. You should have the card handy while you are learning *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

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

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 both version 6 and 7 systems. If you are using the standard version 7 shell then you should 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. 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=`tse**t – –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 which kind of terminal you have, you should make a copy of a file you are familiar with, and run *vi* on this file, giving 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. If something else happens refer to the footnote.‡

1.3. The editor's copy: the buffer

The editor does not directly modify the file which 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 which must be typed as is will be presented in **bold face**. Text which should be replaced with appropriate input will be given in *italics*. We will 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 will 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 labelled with arrows

‡ If you gave the system an incorrect terminal type code then the editor may have just made a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. In this case 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.

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

If the editor doesn't seem to respond to the commands which you type here, try sending an interrupt to it by hitting the DEL or RUB key on your terminal, and then hitting the **:q** command again followed by a carriage return.

on an *adm3a).**

(Particular note for the HP2621: on this terminal the function keys must be *shifted* (ick) to send to the machine, otherwise they only act locally. Unshifted use will leave 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 labelled ESC or ALT. It should be near the upper left corner of your terminal. Try hitting this key a few times. The editor will ring the bell to indicate that it is in a quiescent state.‡ Partially formed commands are cancelled 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 will write the contents of the editor's buffer back into the file you are editing, if you made any changes, and then quit from the editor. You can also end an editor session by giving the command **:q!**CR;† this is a dangerous but occasionally essential command which 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 labelled '^' 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 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.

‡ On smart terminals where it is possible, the editor will quietly flash the screen rather than ringing the bell.

* Backspacing over the '/' will also cancel 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 which 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 labelled '↑'; in any case these characters are one and the same.

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.

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

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 will leave 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 will position the cursor at the next occurrence of this string. Try hitting **n** to then go to the next occurrence of this string. The character **?** will search 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 will print a diagnostic on the last line of the screen, and the cursor will be 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 **$**. Thus **/↑search**CR will search for the word 'search' at the beginning of a line, and **/last$**CR searches for the word 'last' at the end of a line.*

The command **G**, when preceded by a number will position the cursor at that line in the file. Thus **1G** will move the cursor to the first line of the file. If you give **G** no count, then 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 will place only the character '~' on each remaining line. This indicates that the last line in the file is on the screen; that is, the '~' lines are past the end of the file.

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

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

---

‡‡ Version 3 only.

‡ Not available in all v2 editors due to memory constraints.

† These searches will normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction 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 doing **:se nomagic**CR; by putting this command in EXINIT in your environment, you can have this always be in effect (more about *EXINIT* later.)

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 using v2 editors, 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 will scroll down (and up if possible) to bring a line at a time into view. The RETURN key has the same effect as the **+** key.

Vi also has commands to take you to the top, middle and bottom of the screen. **H** will take you to the top (home) line on the screen. Try preceding it with a number as in **3H**. This will take 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 line on the screen. **L** also takes counts, thus **5L** will take 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 will advance the cursor to the next word on the line. Try hitting the **b** key to back up words in the line. Also 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** will move to the right.)

If the line had punctuation in it you may have noticed that that the **w** and **b** keys stopped at each group of punctuation. You can also go back and forwards 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 will set the *readonly* option which will prevent 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 will seem, for a minute, that some of the characters in your line have been overwritten, but they will 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 ‘sESC’ to terminate the textual insert. This sequence of commands can be used to easily pluralize a word.

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.

It is often the case that you want to add new lines to the file you are editing, 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, text you type up to an ESC is inserted on the new line.

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, 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 will be 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 will let you type over the existing screen lines. This avoids the lengthy delay which would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen will be fixed up, and the missing lines will reappear, when you hit ESC.

While you are inserting new text, you can use the characters you normally use at the system command level (usually **^H** or **#**) to backspace over the last character which you typed, and the character which you use to kill input lines (usually **@**, **^X**, or **^U**) to erase the input you have typed on the current line.† The character **^W** will erase a whole word and leave 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; if you want to get rid of them immediately, hit an ESC and then **a** again.

---

‡ Not available in all v2 editors due to memory constraints.

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

Notice also that you can't erase characters which you didn't insert, 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.

3.2. Making small corrections

You can make small corrections in existing text quite easily. Find a single character which 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 which 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 which is incorrect should be replaced by more than one character, give the command **s** which substitutes a string of characters, ending with ESC, for it. If there are a small number of characters which 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 which 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 which 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

It is often the case that you want to operate on lines. Find a line which you want to delete, and type **dd**, the **d** operator twice. This will delete 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 will change 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 will also always tell you when a change you make affects text which 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.

* One subtle point here involves using the / search after a **d**. This will normally delete characters from the current position to the point of the match. If what is desired is 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 which you made was incorrect; you could use the insert, delete and append commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, the editor provides a **u** (undo) command to reverse the last change which 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 which 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 where there is 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. There is also a **F** command, which is like **f**, but searches backward. The **;** command repeats **F** also.

When you are operating on the text in a line it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try **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 **$** moves it to the end of the line. Thus **$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 which represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

---

* This is settable by a command of the form **:se ts=***x*CR, where *x* is 4 to set tabstops every four columns. This has effect on the screen representation within the editor.

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 spacing or backspacing over the character will reveal that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes discards control characters, depending on the character and the setting of the *beautify* option, if you attempt to insert them in your file. You can get a control character in 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

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

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

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

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 away text is saved, and a set of named buffers **a**–**z** which you can use to save copies of text and to move text around in your file and between files.

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

If the text which you yank forms a part of a line, or is an object such as a sentence which partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if

---

† 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.

you use **P**). If the yanked text forms whole lines, they will be put back as whole lines, without changing the current line. In this case, the put 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** will also make a copy of the current line, and place it after the current line. You can give **Y** a count of lines to yank, 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. You can precede a delete operation by the name of a buffer in which the text is to be stored as in **"a5dd** deleting 5 lines into the named buffer *a*. You can then move the cursor to the eventual resting place of the these lines and do a **"ap** or **"aP** to put them back. In fact, you can switch and edit another file before you put the lines back, by giving a command of the form **:e** *name*CR where *name* is the name of the other file you want to edit. You will have to write back the contents of the current editor buffer (or discard them) if you have made changes before the editor will let you switch to the other file. An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files, so to move text from one file to another you should use an unnamed buffer.

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, either because you messed up the file or decided that the changes are not an improvement to the file, 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 it is not possible to recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving the editor by giving the command **:e** *name*CR. 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 will run the single command *cmd* and when the command finishes, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN and the editor will clear the screen and redraw it. You can then continue editing. You can also give another **:** command when it asks you for a RETURN; in this case the screen will not be redrawn.

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

On systems which support it, **^Z** will suspend the editor and return to the (top level) shell. When the editor is resumed, the screen will be 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 will return to the place which you marked. Marks last only until you edit another file.

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

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. (**z.**, **z-**, and **z+** are not available on all v2 editors.)

6. Special topics

6.1. Editing on slow terminals

When you are on a slow terminal, it is important to limit the amount of output which is generated to your screen so that you will not suffer long delays, waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how the editor 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. 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 editing in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor

can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window which is redrawn each time the screen is cleared by giving window sizes as argument to the commands which 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 will draw three line windows around each instance of the string which 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 RUB 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.

You can get a list of all options which you have changed by the command **:set**CR, or the value of a single option by the command **:set** *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.

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

Options set by the **set** command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands† which are to be run every time you start up *ex*, *edit*, or *vi*. A typical list includes a **set** command, and possibly a few **map** commands (on v3 editors). Since it is advisable to get these commands on one line, they can be separated with the | character, for example:

set ai aw terse|**map** @ dd|**map** # x

which sets the options *autoindent*, *autowrite*, *terse*, (the **set** command), makes @ delete a line, (the first **map**), and makes # delete a character, (the second **map**). (See section 6.9 for a description of the **map** command, which only works in version 3.) 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|**map** @ dd|**map** # x´

If you use the standard v7 shell, put these lines in the file *.profile* in your home directory:

EXINIT=´**set** ai aw terse|**map** @ dd|**map** # x´
export EXINIT

On a version 6 system, the concept of environments is not present. In this case, put the line in the file *.exrc* in your home directory.

set ai aw terse|**map** @ dd|**map** # x

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

6.3. Recovering lost lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. Despair not, the editor saves the last 9 deleted blocks of text in a set of numbered registers 1–9. You can get the *n*'th 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 try (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this and then **.** (period) to repeat the put command. In general the **.** command will repeat the last change you made. As a special case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. Thus a sequence of the form

"1pu.u.u.

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

6.4. Recovering lost files

If the system crashes, you can recover the work you were doing to within a few changes. You will normally receive mail when you next login giving you the name of the file which has been saved for you. You should then change to the directory where you were when the system crashed and give a command of the form:

% **vi** –**r** *name*

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

---

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

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

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

% **vi –r**

If there is more than one instance of a particular file saved, the editor gives you the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

For this feature to work, *vi* must be correctly installed by a super user on your system, and the *mail* program must exist to receive mail. The invocation ‘‘*vi -r*’’ will not always list all saved files, but they can be recovered 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=10**CR. This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.*

If the editor breaks an input line 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. You can 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. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure to the body of the program. The editor has a *autoindent* facility for helping 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* which you can 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, there are operators **<** and **>.** These shift the lines you specify right or left by one *shiftwidth.* Try **<<** and **>>** which shift one line left or right, and **<L** and **>L** shifting the rest of the display left and right.

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

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

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.

---

* This feature is not available on some v2 editors. In v2 editors where it is available, the break can only occur to the right of the specified boundary instead of to the left.

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 *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.

There is another option which is useful for typing in LISP, the *showmatch* option. 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,, the **[[** 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 flavors of macros:

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

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

:map *lhs rhs*CR

mapping *lhs* into *rhs.* There are restrictions: *lhs* should be one keystroke (either 1 character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and *vi* will wait for you to finish it before it echoes anything). The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs* you should escape them with a **^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**, it will be as though you had typed the four characters **:wq**CR. A **^V**'s is needed because without it the CR would end the **:** command, rather than becoming part of the *map* definition. There are two **^V**'s 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:

---

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

‡ The macro feature is available only in version 3 editors.

:map **^V^V^I** #

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

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

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

:map **^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. There is no need for an abbreviation to be a single keystroke, as it should be with a macro.

7.1. Abbreviations

The editor has a number of short commands which abbreviate longer commands which we have 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 which advance lines advance logical lines and will skip over all the segments of a line in one motion. The command | moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try **80|** on a line which is more than 80 columns long.†

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

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

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

---

‡‡ Version 3 only.

† You can make long lines very easily by using **J** to join together short lines.

8.2. Counts

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

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

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

This size is the size used when the editor clears and refills the screen after a search or other motion moves far from the edge of the current window. The commands which take a new window size as count all often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a – or similar command or off the bottom with a command such as RETURN or **^D**. The window will revert 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, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus **10a+**———ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands which ignore any counts (such as **^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. A very useful instance of a count as a repetition is a count given to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you will delete first one and then three words. You can then delete two more words with **2.**.

8.3. More file manipulation commands

The following table lists the file manipulation commands which 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 will end with a **ZZ** command. If you are editing for a long period of time you can give **:w** commands occasionally after major amounts of editing, and then finish with a **ZZ**. When you edit more than one file, you can finish with 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, then you must give an **!** after the command you would otherwise use; this forces the editor to discard any changes you have made. Use this carefully.

The **:e** command can be given a **+** argument to start at the end of the file, or a **+***n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, 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. Thus if you try to do a **:e** and get a diagnostic that you haven't written the file, you can give a **:w** command and then a **:e #** command to redo the previous **:e**.

You can write part of the buffer to a file by 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. You can also mark these

† 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*

lines with **m** and then use an address of the form *´x,´y* on the **w** command here.

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

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command **:n**. It is also possible to 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 on the initial *vi* command.

If you are editing large programs, you will find the **:ta** command very useful. It utilizes a data base of function names and their locations, which can be created by programs such as *ctags*, to quickly find a function whose name you give. If the **:ta** command will require the editor to switch files, then you must **:w** or abandon any changes before switching. You can repeat the **:ta** command without any arguments to look for the same tag again. (The tag feature is not available in some v2 editors.)

8.4. More about searching for strings

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

You can have the editor ignore the case of words in 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 an extended pattern matching facility. It is also necessary to use a \ before a **/** in a forward scan or a **?** in a backward scan, in any case. The following table gives the extended forms when **magic** is set.

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

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

8.5. More about input mode

There are a number of characters which you can use to 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 to input is by typing **^H** to correct a single character, or by typing one or more **^W**'s to back over incorrect words. If you use **#** as your erase character in the normal system, it will work like **^H**.

Your system kill character, normally **@**, **^X** or **^U**, will erase all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters which 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 to continue. 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.*

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

---

* 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 appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ↑ before you type the character. In fact, the editor will treat a following letter as a request for the corresponding control character. This is the only way to type **^S** or **^Q**, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

When you are using *autoindent* you may wish to place a label at the left margin of a line. The way to do this easily is to type ↑ and then **^D**. The editor will move 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** if you wish 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 which 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 of the **:** commands which were 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 will get a diagnostic and be 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 **:** which *ex* prompts you with, or you can reenter *vi* by giving *ex* a *vi* command.

There are a number of things which you can do more easily 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 a lot 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 tty’s’’ ‡

If you are on a hardcopy terminal or a terminal which does not have a cursor which 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 will tell 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 will retype 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 which are deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals which can support *vi* in the full screen mode. You can do this by entering *ex* and using an *open* command.

Acknowledgements

Bruce Englar encouraged the early development of this display editor. Peter Kessler helped bring sanity to version 2’s command layout. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

---

‡ The \ character you give will not echo until you type another key.

‡ Not available in all v2 editors due to memory constraints.

Appendix: character functions

This appendix gives the uses the editor makes of each character. 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 a meaning it has as a command and any meaning it has during an insert. If it has only meaning as a command, then only this is discussed. Section numbers in parentheses indicate 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 specifies repetition. 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 Exposes one more line below the current screen in the file, leaving the cursor where it is if possible. (Version 3 only.)

^F Forward window. A count specifies repetition. 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 and 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 character it rests at the last of the spaces which 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 **^Q**is

^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, inverting **^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 Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; however, it is next to **^U** which scrolls up a bunch.) (Version 3 only.)

^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 will be inserted correctly whether or not you were in insert mode when you started (1.5, 3.1, 7.5).

^ Unused.

^] Searches for the word which 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 which 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, which 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 **!**. Thus **2!}***fmt*CR reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the command **!%***grind*CR,* given at the beginning of a function, will run the text of the function through the LISP grinder (6.7, 7.3). 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 and named buffers **a**–**z** into which you can place text (4.3, 6.3)

The macro character which, when followed by a number, will substitute for a function key on terminals without function keys (6.9). In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a

*Both *fmt* and *grind* are Berkeley programs and may not be present at all installations.

\ 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 **{ }** which 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 which 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 **?** which is followed by either the end of a line or by two spaces. Any number of closing **)]** " and ´ characters may appear after the **. !** or **?**, 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, the screen is scrolled, or cleared and redrawn if this is not possible. 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 which 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; an returns to command state without ever searching. 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 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 speeds 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 will have to preceed the characters **. [***

and ~ in the search pattern with a \ to get them to work as you would naively expect (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, and the command 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 which used **f F t** or **T**. A count iterates the basic scan (4.1).

< An operator which 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 line for LISP, as though they were typed in with *lisp* and *autoindent* set (6.8).

> An operator which 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).

@ A macro character (6.9). If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line (3.1, 3.4, 7.5).

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 that many times (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 with the new current line in the center if necessary (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, buffers **a**–**z** are available for general use (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. May be preceded by a buffer name to put lines in that buffer (7.4).

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 which which 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.

]] 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 which 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 which 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 which is changed away is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away 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 which 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.

Arrow keys **h**, **j**, **k**, **l**, and **H**.

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 which is specified by the next character **a**–**z**. Return to this position or use with an operator using ` 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 usually useful iteration of **r** (3.2).

s Changes the single character under the cursor to the text which 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 upto 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, will alternate between these two states, thus is its own inverse. When used after an insert which 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. If preceded by a named buffer specification, "*x*, the text is placed in that buffer also. 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 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)

Ex Quick Reference

Entering/leaving ex

% ex *name*	edit *name*, start at end
% ex +*n name*	... at line *n*
% ex –**t** *tag*	start at *tag*
% ex –**r**	list saved files
% ex –**r** *name*	recover file *name*
% ex *name* **...**	edit first; rest via **:n**
% ex –**R** *name*	read only mode
: x	exit, saving changes
: q!	exit, discarding changes

Ex states

Command	Normal and initial state. Input prompted for by **:**. Your kill character cancels partial command.
Insert	Entered by **a i** and **c**. Arbitrary text then terminates with line having only **.** character on it or abnormally with interrupt.
Open/visual	Entered by **open** or **vi**, terminates with **Q** or ^\.

Ex commands

abbrev	**ab**	next	**n**	unabbrev	**una**
append	**a**	number	**nu**	undo	**u**
args	**ar**	open	**o**	unmap	**unm**
change	**c**	preserve	**pre**	version	**ve**
copy	**co**	print	**p**	visual	**vi**
delete	**d**	put	**pu**	write	**w**
edit	**e**	quit	**q**	xit	**x**
file	**f**	read	**re**	yank	**ya**
global	**g**	recover	**rec**	*window*	**z**
insert	**i**	rewind	**rew**	*escape*	**!**
join	**j**	set	**se**	*lshift*	<
list	**l**	shell	**sh**	*print next*	CR
map		source	**so**	*resubst*	**&**
mark	**ma**	stop	**st**	*rshift*	>
move	**m**	substitute	**s**	*scroll*	**^D**

Ex command addresses

n	line *n*	*/pat*	next with *pat*
.	current	**?***pat*	previous with *pat*
$	last	*x*-*n*	*n* before *x*
+	next	*x*,*y*	*x* through *y*
–	previous	´*x*	marked with *x*
+*n*	*n* forward	´´	previous context
%	1,$		

Specifying terminal type

% setenv TERM *type*	*csh* and all version 6
$ TERM=*type***; export TERM**	*sh* in Version 7

See also *tset*(1)

Some terminal types

2621	43	adm31	dw1	h19
2645	733	adm3a	dw2	i100
300s	745	c100	gt40	mime
33	act4	dm1520	gt42	owl
37	act5	dm2500	h1500	t1061
4014	adm3	dm3025	h1510	vt52

Initializing options

EXINIT	place **set**'s here in environment var.
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set all	show all options
set *x***?**	show value of option *x*

Useful options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp		**( ) { }** are s-exp's
list		print ^I for tab, $ at end
magic		**. [*** special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input **^D**
showmatch	sm	to **)** and **}** as typed
slowopen	slow	choke updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer?
wrapmargin	wm	automatic line splitting

Scanning pattern formation

↑	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str***]**	any char in *str*
[↑*str***]**	... not in *str*
[*x***–***y***]**	... between *x* and *y*
*****	any number of preceding

Vi Quick Reference

Entering/leaving vi

% vi *name*	edit *name* at top
% vi +*n name*	... at line *n*
% vi + *name*	... at end
% vi –r	list saved files
% vi –r *name*	recover file *name*
% vi *name* **...**	edit first; rest via **:n**
% vi –t *tag*	start at *tag*
% vi +/*pat name*	search for *pat*
% view *name*	read only mode
ZZ	exit from vi, saving changes
^Z	stop vi for later resumption

The display

Last line	Error messages, echoing input to **: / ?** and **!**, feedback about i/o and large changes.
@ lines	On screen only, not in file.
~ lines	Lines past end of file.
^*x*	Control characters, ^? is delete.
tabs	Expand to spaces, cursor at last.

Vi states

Command	Normal and initial state. Others return here. ESC (escape) cancels partial command.
Insert	Entered by **a i A I o O c C s S R**. Arbitrary text then terminates with ESC character, or abnormally with interrupt.
Last line	Reading input for **: / ?** or **!**; terminate with ESC or CR to execute, interrupt to cancel.

Counts before vi commands

line/column number	**z G |**
scroll amount	**^D ^U**
replicate insert	**a i A I**
repeat effect	most rest

Simple commands

dw	delete a word
de	... leaving punctuation
dd	delete a line
3dd	... 3 lines
i*text*ESC	insert text *abc*
cw*new*ESC	change word to *new*
easESC	pluralize word
xp	transpose characters

--

--

Interrupting, cancelling

ESC	end insert or incomplete cmd
^?	(delete or rubout) interrupts
^L	reprint screen if **^?** scrambles it

File manipulation

:w	write back changes
:wq	write and quit
:q	quit
:q!	quit, discard changes
:e *name*	edit file *name*
:e!	reedit, discard changes
:e + *name*	edit, starting at end
:e +*n*	edit starting at line *n*
:e #	edit alternate file
^↑	synonym for **:e #**
:w *name*	write file *name*
:w! *name*	overwrite file *name*
:sh	run shell, then return
:!*cmd*	run *cmd*, then return
:n	edit next file in arglist
:n *args*	specify new arglist
:f	show current file and line
^G	synonym for **:f**
:ta *tag*	to tag file entry *tag*
^]	**:ta**, following word is *tag*

Positioning within file

^F	forward screenfull
^B	backward screenfull
^D	scroll down half screen
^U	scroll up half screen
G	goto line (end default)
/*pat*	next line matching *pat*
?*pat*	prev line matching *pat*
n	repeat last / or **?**
N	reverse last / or **?**
/*pat*/+*n*	n’th line after *pat*
?*pat***?**–*n*	n’th line before *pat*
]]	next section/function
[[	previous section/function
%	find matching **( ) {** or **}**

Adjusting the screen

^L	clear and redraw
^R	retype, eliminate @ lines
zCR	redraw, current at window top
z–	... at bottom
z .	... at center
/*pat*/**z**–	*pat* line at bottom
z*n* **.**	use *n* line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

Marking and returning

``` `` ```	previous context
´´	... at first non-white in line
m*x*	mark position with letter *x*
`*x*	to mark *x*
´*x*	... at first non-white in line

Line positioning

H	home window line
L	last window line
M	middle window line
+	next line, at first non-white
–	previous line, at first non-white
CR	return, same as +
↓ or **j**	next line, same column
↑ or **k**	previous line, same column

Character positioning

↑	first non white
0	beginning of line
$	end of line
h or →	forward
l or ←	backwards
^H	same as ←
space	same as →
f*x*	find *x* forward
F*x*	**f** backward
t*x*	upto *x* forward
T*x*	back upto *x*
;	repeat last **f F t** or **T**
,	inverse of **;**
|	to specified column
%	find matching **({)** or **}**

Words, sentences, paragraphs

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back **W**
E	to end of **W**

Commands for LISP

)	Forward s-expression
}	... but don’t stop at atoms
(	Back s-expression
{	... but don’t stop at atoms

Corrections during insert

^H	erase last character
^W	erases last word
erase	your erase, same as **^H**
kill	your kill, erase input this line
\	escapes **^H**, your erase and kill
ESC	ends insertion, back to command
^?	interrupt, terminates insert
^D	backtab over *autoindent*
↑^D	kill *autoindent*, save for next
0^D	... but at margin next also
^V	quote non-printing character

Insert and replace

a	append after cursor
i	insert before
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
r*x*	replace single char with *x*
R	replace characters

Operators (double to affect lines)

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

Miscellaneous operations

C	change rest of line
D	delete rest of line
s	substitute chars
S	substitute lines
J	join lines
x	delete characters
X	... before cursor
Y	yank lines

Yank and put

p	put back lines
P	put before
"*x***p**	put from buffer *x*
"*x***y**	yank to buffer *x*
"*x***d**	delete into buffer *x*

Undo, redo, retrieve

u	undo last change
U	restore current line
.	repeat last change
"*d***p**	retrieve *d*'th last delete

Vi Command & Function Reference

Alan P.W. Hewett

Revised for version 2.12 by Mark Horton

1. Author's Disclaimer

This document does not claim to be 100% complete. There are a few commands listed in the original document that I was unable to test either because I do not speak **lisp**, because they required programs we don't have, or because I wasn't able to make them work. In these cases I left the command out. The commands listed in this document have been tried and are known to work. It is expected that prospective users of this document will read it once to get the flavor of everything that **vi** can do and then use it as a reference document. Experimentation is recommended. If you don't understand a command, try it and see what happens.

[Note: In revising this document, I have attempted to make it completely reflect version 2.12 of **vi**. It does not attempt to document the VAX version (version 3), but with one or two exceptions (wrapmargin, arrow keys) everything said about 2.12 should apply to 3.1. *Mark Horton*]

2. Notation

[option] is used to denote optional parts of a command. Many **vi** commands have an optional count. **[cnt]** means that an optional number may precede the command to multiply or iterate the command. **{variable item}** is used to denote parts of the command which must appear, but can take a number of different values. **<character [-character]>** means that the character or one of the characters in the range described between the two angle brackets is to be typed. For example **<esc>** means the **escape** key is to be typed. **<a-z>** means that a lower case letter is to be typed. **^<character>** means that the character is to be typed as a **control** character, that is, with the **<cntl>** key held down while simultaneously typing the specified character. In this document control characters will be denoted using the *upper case* character, but ^<uppercase chr> and ^<lowercase chr> are equivalent. That is, for example, **<^D>** is equal to **<^d>**. The most common character abbreviations used in this list are as follows:

<esc> escape, octal 033

<cr> carriage return, ^M, octal 015

<lf> linefeed ^J, octal 012

<nl> newline, ^J, octal 012 (same as linefeed)

<bs> backspace, ^H, octal 010

<tab> tab, ^I, octal 011

<bell> bell, ^G, octal 07

<ff> formfeed, ^L, octal 014

<sp> space, octal 040

<del> delete, octal 0177

3. Basics

To run **vi** the shell variable **TERM** must be defined and exported to your environment. How you do this depends on which shell you are using. You can tell which shell you have by the character it prompts you for commands with. The Bourne shell prompts with '$', and the C shell prompts with '%'. For these examples, we will suppose that you are using an HP 2621 terminal, whose termcap name is ''2621''.

3.1. Bourne Shell

To manually set your terminal type to 2621 you would type:

```
TERM=2621
export TERM
```

There are various ways of having this automatically or semi-automatically done when you log in. Suppose you usually dial in on a 2621. You want to tell this to the machine, but still have it work when you use a hardwired terminal. The recommended way, if you have the **tset** program, is to use the sequence

```
tset –s –d 2621 > tset$$
. tset$$
rm tset$$
```

in your .login (for csh) or the same thing using '.' instead of 'source' in your .profile (for sh). The above line says that if you are dialing in you are on a 2621, but if you are on a hardwired terminal it figures out your terminal type from an on-line list.

3.2. The C Shell

To manually set your terminal type to 2621 you would type:

```
setenv TERM 2621
```

There are various ways of having this automatically or semi-automatically done when you log in. Suppose you usually dial in on a 2621. You want to tell this to the machine, but still have it work when you use a hardwired terminal. The recommended way, if you have the **tset** program, is to use the sequence

```
tset –s –d 2621 > tset$$
source tset$$
rm tset$$
```

in your .login.* The above line says that if you are dialing in you are on a 2621, but if you are on a hardwired terminal it figures out your terminal type from an on-line list.

4. Normal Commands

Vi is a visual editor with a window on the file. What you see on the screen is **vi**'s current notion of what your file will contain, (at this point in the file), when it is written out. Most commands do not cause any change in the screen until the complete command is typed. Should you get confused while typing a command, you can abort the command by typing an <del> character. You will know you are back to command level when you hear a <bell>. Usually typing an <esc> will produce the same result. When **vi** gets an improperly formatted command it rings the <bell>. Following are the **vi** commands broken down by function.

4.1. Entry and Exit

To enter **vi** on a particular *file* , type

---

* On a version 6 system without environments, the invocation of tset is simpler, just add the line ''tset –d 2621'' to your

vi *file*

The file will be read in and the cursor will be placed at the beginning of the first line. The first screenfull of the file will be displayed on the terminal.

To get out of the editor, type

ZZ

If you are in some special mode, such as input mode or the middle of a multi-keystroke command, it may be necessary to type <esc> first.

4.2. Cursor and Page Motion

NOTE: The arrow keys (see the next four commands) on certain kinds of terminals will not work with the PDP-11 version of vi. The control versions or the hjkl versions will work on any terminal. Experienced users prefer the hjkl keys because they are always right under their fingers. Beginners often prefer the arrow keys, since they do not require memorization of which hjkl key is which. The mnemonic value of hjkl is clear from looking at the keyboard of an adm3a.

[cnt]<bs> or [cnt]h or [cnt]←
: Move the cursor to the left one character. Cursor stops at the left margin of the page. If cnt is given, these commands move that many spaces.

[cnt]^N or [cnt]j or [cnt]↓ or [cnt]<lf>
: Move down one line. Moving off the screen scrolls the window to force a new line onto the screen. Mnemonic: **N**ext

[cnt]^P or [cnt]k or [cnt]↑
: Move up one line. Moving off the top of the screen forces new text onto the screen. Mnemonic: **P**revious

[cnt]<sp> or [cnt]l or [cnt]→
: Move to the right one character. Cursor will not go beyond the end of the line.

[cnt]-
: Move the cursor up the screen to the beginning of the next line. Scroll if necessary.

[cnt]+ or [cnt]<cr>
: Move the cursor down the screen to the beginning of the next line. Scroll up if necessary.

[cnt]$
: Move the cursor to the end of the line. If there is a count, move to the end of the line "cnt" lines forward in the file.

^
: Move the cursor to the beginning of the first word on the line.

0
: Move the cursor to the left margin of the current line.

[cnt]|
: Move the cursor to the column specified by the count. The default is column zero.

[cnt]w
: Move the cursor to the beginning of the next word. If there is a count, then move forward that many words and position the cursor at the beginning of the word. Mnemonic: next-**w**ord

[cnt]W
: Move the cursor to the beginning of the next word which follows a "white space" (<sp>,<tab>, or <nl>). Ignore other punctuation.

[cnt]b
: Move the cursor to the preceding word. Mnemonic: **b**ackup-word

[cnt]B
: Move the cursor to the preceding word that is separated from the current word by a "white space" (<sp>,<tab>, or <nl>).

[cnt]e
: Move the cursor to the end of the current word or the end of the "cnt"'th word hence. Mnemonic: **e**nd-of-word

[cnt]E Move the cursor to the end of the current word which is delimited by "white space" (<sp>,<tab>, or <nl>).

[line number]G
Move the cursor to the line specified. Of particular use are the sequences "1G" and "G", which move the cursor to the beginning and the end of the file respectively. Mnemonic: **G**o-to

NOTE: The next four commands (^D, ^U, ^F, ^B) are not true motion commands, in that they cannot be used as the object of commands such as delete or change.

[cnt]^D Move the cursor down in the file by "cnt" lines (or the last "cnt" if a new count isn't given. The initial default is half a page.) The screen is simultaneously scrolled up. Mnemonic: **D**own

[cnt]^U Move the cursor up in the file by "cnt" lines. The screen is simultaneously scrolled down. Mnemonic: **U**p

[cnt]^F Move the cursor to the next page. A count moves that many pages. Two lines of the previous page are kept on the screen for continuity if possible. Mnemonic: **F**orward-a-page

[cnt]^B Move the cursor to the previous page. Two lines of the current page are kept if possible. Mnemonic: **B**ackup-a-page

[cnt](Move the cursor to the beginning of the next sentence. A sentence is defined as ending with a ".", "!", or "?" followed by two spaces or a <nl>.

[cnt]) Move the cursor backwards to the beginning of a sentence.

[cnt]} Move the cursor to the beginning of the next paragraph. This command works best inside **nroff** documents. It understands two sets of **nroff** macros, **–ms** and **–mm**, for which the commands ".IP", ".LP", ".PP", ".QP", "P", as well as the nroff command ".bp" are considered to be paragraph delimiters. A blank line also delimits a paragraph. The **nroff** macros that it accepts as paragraph delimiters is adjustable. See **paragraphs** under the **Set Commands** section.

[cnt]{ Move the cursor backwards to the beginning of a paragraph.

]] Move the cursor to the next "section", where a section is defined by two sets of **nroff** macros, **–ms** and **–mm**, in which ".NH", ".SH", and ".H" delimit a section. A line beginning with a <ff><nl> sequence, or a line beginning with a "{" are also considered to be section delimiters. The last option makes it useful for finding the beginnings of C functions. The **nroff** macros that are used for section delimiters can be adjusted. See **sections** under the **Set Commands** section.

[[Move the cursor backwards to the beginning of a section.

% Move the cursor to the matching parenthesis or brace. This is very useful in C or lisp code. If the cursor is sitting on a **( ) {** or **}** the cursor is moved to the matching character at the other end of the section. If the cursor is not sitting on a brace or a parenthesis, **vi** searches forward until it finds one and then jumps to the match mate.

[cnt]H If there is no count move the cursor to the top left position on the screen. If there is a count, then move the cursor to the beginning of the line "cnt" lines from the top of the screen. Mnemonic: **H**ome

[cnt]L If there is no count move the cursor to the beginning of the last line on the screen. If there is a count, then move the cursor to the beginning of the line "cnt" lines from the bottom of the screen. Mnemonic: **L**ast

M Move the cursor to the beginning of the middle line on the screen. Mnemonic: **M**iddle

m<a-z> This command does not move the cursor, but it **marks** the place in the file and the character "<a-z>" becomes the label for referring to this location in the file. See the next two commands. Mnemonic: **m**ark **NOTE:** The mark command is not a motion, and cannot be used as the target of commands such as delete.

´<a-z> Move the cursor to the beginning of the line that is marked with the label "<a-z>".

`<a-z> Move the cursor to the exact position on the line that was marked with with the label "<a-z>".

´´ Move the cursor back to the beginning of the line where it was before the last "non-relative" move. A "non-relative" move is something such as a search or a jump to a specific line in the file, rather than moving the cursor or scrolling the screen.

`` Move the cursor back to the exact spot on the line where it was located before the last "non-relative" move.

4.3. Searches

The following commands allow you to search for items in a file.

[cnt]f{chr}

Search forward on the line for the next or "cnt"'th occurrence of the character "chr". The cursor is placed **at** the character of interest. Mnemonic: **f**ind character

[cnt]F{chr}

Search backwards on the line for the next or "cnt"'th occurrence of the character "chr". The cursor is placed **at** the character of interest.

[cnt]t{chr}

Search forward on the line for the next or "cnt"'th occurrence of the character "chr". The cursor is placed **just preceding** the character of interest. Mnemonic: move cursor up **t**o character

[cnt]T{chr}

Search backwards on the line for the next or "cnt"'th occurrence of the character "chr". The cursor is placed **just preceding** the character of interest.

[cnt]; Repeat the last "f", "F", "t" or "T" command.

[cnt], Repeat the last "f", "F", "t" or "T" command, but in the opposite search direction. This is useful if you overshoot.

[cnt]/[string]/<nl>

Search forward for the next occurrence of "string". Wrap around at the end of the file does occur. The final **</>** is not required.

[cnt]?[string]?<nl>

Search backwards for the next occurrence of "string". If a count is specified, the count becomes the new window size. Wrap around at the beginning of the file does occur. The final **<?>** is not required.

n Repeat the last /[string]/ or ?[string]? search. Mnemonic: **n**ext occurrence.

N Repeat the last /[string]/ or ?[string]? search, but in the reverse direction.

:g/[string]/[editor command]<nl>

Using the **:** syntax it is possible to do global searches ala the standard UNIX "ed" editor.

4.4. Text Insertion

The following commands allow for the insertion of text. All multicharacter text insertions are terminated with an <esc> character. The last change can always be **undone** by typing a **u**. The text insert in insertion mode can contain newlines.

a{text}<esc>	Insert text immediately following the cursor position. Mnemonic: **a**ppend
A{text}<esc>	Insert text at the end of the current line. Mnemonic: **A**ppend
i{text}<esc>	Insert text immediately preceding the cursor position. Mnemonic: **i**nsert
I{text}<esc>	Insert text at the beginning of the current line.
o{text}<esc>	Insert a new line after the line on which the cursor appears and insert text there. Mnemonic: **o**pen new line
O{text}<esc>	Insert a new line preceding the line on which the cursor appears and insert text there.

4.5. Text Deletion

The following commands allow the user to delete text in various ways. All changes can always be **undone** by typing the **u** command.

[cnt]x	Delete the character or characters starting at the cursor position.
[cnt]X	Delete the character or characters starting at the character preceding the cursor position.
D	Deletes the remainder of the line starting at the cursor. Mnemonic: **D**elete the rest of line
[cnt]d{motion}	Deletes one or more occurrences of the specified motion. Any motion from sections 4.1 and 4.2 can be used here. The d can be stuttered (e.g. [cnt]dd) to delete cnt lines.

4.6. Text Replacement

The following commands allow the user to simultaneously delete and insert new text. All such actions can be **undone** by typing **u** following the command.

r<chr>	Replaces the character at the current cursor position with <chr>. This is a one character replacement. No <esc> is required for termination. Mnemonic: **r**eplace character
R{text}<esc>	Starts overlaying the characters on the screen with whatever you type. It does not stop until an <esc> is typed.
[cnt]s{text}<esc>	Substitute for "cnt" characters beginning at the current cursor position. A "$" will appear at the position in the text where the "cnt"'th character appears so you will know how much you are erasing. Mnemonic: **s**ubstitute
[cnt]S{text}<esc>	Substitute for the entire current line (or lines). If no count is given, a "$" appears at the end of the current line. If a count of more than 1 is given, all the lines to be replaced are deleted before the insertion begins.
[cnt]c{motion}{text}<esc>	Change the specified "motion" by replacing it with the insertion text. A "$" will appear at the end of the last item that is being deleted unless the deletion involves whole lines. Motion's can be any motion from sections 4.1 or 4.2. Stuttering the c (e.g. [cnt]cc) changes cnt lines.

4.7. Moving Text

Vi provides a number of ways of moving chunks of text around. There are nine buffers into which each piece of text which is deleted or "yanked" is put in addition to the "undo" buffer. The most recent deletion or yank is in the "undo" buffer and also usually in buffer 1, the next most recent in buffer 2, and so forth. Each new deletion pushes down all the older deletions. Deletions older than 9 disappear. There is also a set of named registers, a-z, into which text can optionally be placed. If any delete or replacement type command is preceded by **"<a-z>**, that named buffer will contain the text deleted after the command is executed. For example, **"a3dd** will delete three lines starting at the current line and put them in buffer **"a**.*

* Referring to an upper case letter as a buffer name (A-Z) is the same as referring to the lower case letter, except that text placed in such a buffer is appended to it instead of replacing it.

There are two more basic commands and some variations useful in getting and putting text into a file.

["<a-z>][cnt]y{motion}

Yank the specified item or "cnt" items and put in the "undo" buffer or the specified buffer. The variety of "items" that can be yanked is the same as those that can be deleted with the "d" command or changed with the "c" command. In the same way that "dd" means delete the current line and "cc" means replace the current line, "yy" means yank the current line.

["<a-z>][cnt]Y Yank the current line or the "cnt" lines starting from the current line. If no buffer is specified, they will go into the "undo" buffer, like any delete would. It is equivalent to "yy". Mnemonic: **Y**ank

["<a-z>]p Put "undo" buffer or the specified buffer down **after** the cursor. If whole lines were yanked or deleted into the buffer, then they will be put down on the line following the line the cursor is on. If something else was deleted, like a word or sentence, then it will be inserted immediately following the cursor. Mnemonic: **p**ut buffer

It should be noted that text in the named buffers remains there when you start editing a new file with the **:e file<esc>** command. Since this is so, it is possible to copy or delete text from one file and carry it over to another file in the buffers. However, the undo buffer and the ability to undo are lost when changing files.

["<a-z>]P Put "undo" buffer or the specified buffer down **before** the cursor. If whole lines where yanked or deleted into the buffer, then they will be put down on the line preceding the line the cursor is on. If something else was deleted, like a word or sentence, then it will be inserted immediately preceding the cursor.

[cnt]>{motion} The shift operator will right shift all the text from the line on which the cursor is located to the line where the **motion** is located. The text is shifted by one **shiftwidth**. (See section 6.) >> means right shift the current line or lines.

[cnt]<{motion} The shift operator will left shift all the text from the line on which the cursor is located to the line where the **item** is located. The text is shifted by one **shiftwidth**. (See section 6.) << means left shift the current line or lines. Once the line has reached the left margin it is not further affected.

[cnt]={motion} Prettyprints the indicated area according to **lisp** conventions. The area should be a lisp s-expression.

4.8. Miscellaneous Commands

Vi has a number of miscellaneous commands that are very useful. They are:

ZZ This is the normal way to exit from vi. If any changes have been made, the file is written out. Then you are returned to the shell.

^L Redraw the current screen. This is useful if someone "write"s you while you are in "vi" or if for any reason garbage gets onto the screen.

^R On dumb terminals, those not having the "delete line" function (the vt100 is such a terminal), **vi** saves redrawing the screen when you delete a line by just marking the line with an "@" at the beginning and blanking the line. If you want to actually get rid of the lines marked with "@" and see what the page looks like, typing a ^R will do this.

. "Dot" is a particularly useful command. It repeats the last text modifying command. Therefore you can type a command once and then to another place and repeat it by just typing ".".

u Perhaps the most important command in the editor, u undoes the last command that changed the buffer. Mnemonic: **u**ndo

U Undo all the text modifying commands performed on the current line since the last time you moved onto it.

[cnt]J — Join the current line and the following line. The <nl> is deleted and the two lines joined, usually with a space between the end of the first line and the beginning of what was the second line. If the first line ended with a "period", then two spaces are inserted. A count joins the next cnt lines. Mnemonic: **J**oin lines

Q — Switch to **ex** editing mode. In this mode **vi** will behave very much like **ed**. The editor in this mode will operate on single lines normally and will not attempt to keep the "window" up to date. Once in this mode it is also possible to switch to the **open** mode of editing. By entering the command **[line number]open<nl>** you enter this mode. It is similar to the normal visual mode except the window is only **one** line long. Mnemonic: **Q**uit visual mode

^] — An abbreviation for a tag command. The cursor should be positioned at the beginning of a word. That word is taken as a tag name, and the tag with that name is found as if it had been typed in a :tag command.

[cnt]!{motion}{UNIX cmd}<nl> — Any UNIX filter (e.g. command that reads the standard input and outputs something to the standard output) can be sent a section of the current file and have the output of the command replace the original text. Useful examples are programs like **cb**, **sort**, and **nroff**. For instance, using **sort** it would be possible to sort a section of the current file into a new list. Using **!!** means take a line or lines starting at the line the cursor is currently on and pass them to the UNIX command. **NOTE:** To just escape to the shell for one command, use :!{cmd}<nl>, see section 5.

z{cnt}<nl> — This resets the current window size to "cnt" lines and redraws the screen.

4.9. Special Insert Characters

There are some characters that have special meanings during insert modes. They are:

^V — During inserts, typing a ^V allows you to quote control characters into the file. Any character typed after the ^V will be inserted into the file.

[^]^D or [0]^D — <^D> without any argument backs up one **shiftwidth**. This is necessary to remove indentation that was inserted by the **autoindent** feature. ^<^D> temporarily removes all the autoindentation, thus placing the cursor at the left margin. On the next line, the previous indent level will be restored. This is useful for putting "labels" at the left margin. 0<^D> says remove all autoindents and stay that way. Thus the cursor moves to the left margin and stays there on successive lines until <tab>'s are typed. As with the <tab>, the <^D> is only effective before any other "non-autoindent" controlling characters are typed. Mnemonic: **D**elete a shiftwidth

^W — If the cursor is sitting on a word, <^W> moves the cursor back to the beginning of the word, thus erasing the word from the insert. Mnemonic: erase **W**ord

<bs> — The backspace always serves as an erase during insert modes in addition to your normal "erase" character. To insert a <bs> into your file, use the <^V> to quote it.

5. : Commands

Typing a ":" during command mode causes **vi** to put the cursor at the bottom on the screen in preparation for a command. In the ":" mode, **vi** can be given most **ed** commands. It is also from this mode that you exit from **vi** or switch to different files. All commands of this variety are terminated by a <nl>, <cr>, or <esc>.

:w[!] [file] — Causes **vi** to write out the current text to the disk. It is written to the file you are editing unless "file" is supplied. If "file" is supplied, the write is directed to that file instead. If that file already exists, **vi** will not perform the write unless the "!" is supplied indicating you *really* want to destroy the older copy of the file.

:q[!] — Causes **vi** to exit. If you have modified the file you are looking at currently and haven't written it out, **vi** will refuse to exit unless the "!" is supplied.

:e[!] [+[cmd]] [file]

Start editing a new file called "file" or start editing the current file over again. The command ":e!" says "ignore the changes I've made to this file and start over from the beginning". It is useful if you really mess up the file. The optional "+" says instead of starting at the beginning, start at the "end", or, if "cmd" is supplied, execute "cmd" first. Useful cases of this are where cmd is "n" (any integer) which starts at line number n, and "/text", which searches for "text" and starts at the line where it is found.

^^ Switch back to the place you were before your last tag command. If your last tag command stayed within the file, ^^ returns to that tag. If you have no recent tag command, it will return to the same place in the previous file that it was showing when you switched to the current file.

:n[!] Start editing the next file in the argument list. Since **vi** can be called with multiple file names, the ":n" command tells it to stop work on the current file and switch to the next file. If the current file was modifies, it has to be written out before the ":n" will work or else the "!" must be supplied, which says discard the changes I made to the current file.

:n[!] file [file file ...]

Replace the current argument list with a new list of files and start editing the first file in this new list.

:r file Read in a copy of "file" on the line after the cursor.

:r !cmd Execute the "cmd" and take its output and put it into the file after the current line.

:!cmd Execute any UNIX shell command.

:ta[!] tag **Vi** looks in the file named **tags** in the current directory. **Tags** is a file of lines in the format:

tag filename **vi**-search-command

If **vi** finds the tag you specified in the **:ta** command, it stops editing the current file if necessary and if the current file is up to date on the disk and switches to the file specified and uses the search pattern specified to find the "tagged" item of interest. This is particularly useful when editing multi-file C programs such as the operating system. There is a program called **ctags** which will generate an appropriate **tags** file for C and f77 programs so that by saying **:ta function<nl>** you will be switched to that function. It could also be useful when editing multi-file documents, though the **tags** file would have to be generated manually.

6. Special Arrangements for Startup

Vi takes the value of **$TERM** and looks up the characteristics of that terminal in the file **/etc/termcap**. If you don't know **vi**'s name for the terminal you are working on, look in **/etc/termcap**.

When **vi** starts, it attempts to read the variable EXINIT from your environment.* If that exists, it takes the values in it as the default values for certain of its internal constants. See the section on "Set Values" for further details. If EXINIT doesn't exist you will get all the normal defaults.

Should you inadvertently hang up the phone while inside **vi**, or should the computer crash, all may not be lost. Upon returning to the system, type:

vi –r file

This will normally recover the file. If there is more than one temporary file for a specific file name, **vi** recovers the newest one. You can get an older version by recovering the file more than once. The

* On version 6 systems Instead of EXINIT, put the startup commands in the file .exrc in your home directory.

command "vi -r" without a file name gives you the list of files that were saved in the last system crash (but *not* the file just saved when the phone was hung up).

7. Set Commands

Vi has a number of internal variables and switches which can be set to achieve special affects. These options come in three forms, those that are switches, which toggle from off to on and back, those that require a numeric value, and those that require an alphanumeric string value. The toggle options are set by a command of the form:

:set option<nl>

and turned off with the command:

:set nooption<nl>

Commands requiring a value are set with a command of the form:

:set option=value<nl>

To display the value of a specific option type:

:set option?<nl>

To display only those that you have changed type:

:set<nl>

and to display the long table of all the settable parameters and their current values type:

:set all<nl>

Most of the options have a long form and an abbreviation. Both are listed in the following table as well as the normal default value.

To arrange to have values other than the default used every time you enter **vi**, place the appropriate **set** command in EXINIT in your environment, e.g.

```
EXINIT='set ai aw terse sh=/bin/csh'
export EXINIT
```

or

```
setenv EXINIT 'set ai aw terse sh=/bin/csh'
```

for **sh** and **csh**, respectively. These are usually placed in your .profile or .login. If you are running a system without environments (such as version 6) you can place the set command in the file .exrc in your home directory.

autoindent ai — Default: noai Type: toggle
When in autoindent mode, vi helps you indent code by starting each line in the same column as the preceding line. Tabbing to the right with <tab> or <^T> will move this boundary to the right, and it can be moved to the left with <^D>.

autoprint ap — Default: ap Type: toggle
Causes the current line to be printed after each ex text modifying command. This is not of much interest in the normal **vi** visual mode.

autowrite aw — Default: noaw type: toggle
Autowrite causes an automatic write to be done if there are unsaved changes before certain commands which change files or otherwise interact with the outside world. These commands are :!, :tag, :next, :rewind, ^^, and ^].

beautify bf — Default: nobf Type: toggle
Causes all control characters except <tab>, <nl>, and <ff> to be discarded.

directory dir — Default: dir=/tmp Type: string
This is the directory in which **vi** puts its temporary file.

errorbells eb — Default: noeb Type: toggle
Error messages are preceded by a <bell>.

hardtabs ht — Default: hardtabs=8 Type: numeric
This option contains the value of hardware tabs in your terminal, or of software tabs expanded by the Unix system.

ignorecase ic — Default: noic Type: toggle
All upper case characters are mapped to lower case in regular expression matching.

lisp — Default: nolisp Type: toggle
Autoindent for **lisp** code. The commands **( ) [[** and **]]** are modified appropriately to affect s-expressions and functions.

list — Default: nolist Type: toggle
All printed lines have the <tab> and <nl> characters displayed visually.

magic — Default: magic Type: toggle
Enable the metacharacters for matching. These include **. * < > [string] [^string]** and **[<chr>-<chr>]**.

number nu — Default: nonu Type: toggle
Each line is displayed with its line number.

open — Default: open Type: toggle
When set, prevents entering open or visual modes from ex or edit. Not of interest from vi.

optimize opt — Default: opt Type: toggle
Basically of use only when using the **ex** capabilities. This option prevents automatic <cr>s from taking place, and speeds up output of indented lines, at the expense of losing typeahead on some versions of UNIX.

paragraphs para — Default: para=IPLPPPQPP bp Type: string
Each pair of characters in the string indicate **nroff** macros which are to be treated as the beginning of a paragraph for the **{** and **}** commands. The default string is for the **-ms** and **-mm** macros. To indicate one letter **nroff** macros, such as **.P** or **.H**, quote a space in for the second character position. For example:

:set paragraphs=P\ bp<nl>

would cause **vi** to consider **.P** and **.bp** as paragraph delimiters.

prompt — Default: prompt Type: toggle
In **ex** command mode the prompt character **:** will be printed when **ex** is waiting for a command. This is not of interest from vi.

redraw — Default: noredraw Type: toggle
On dumb terminals, force the screen to always be up to date, by sending great amounts of output. Useful only at high speeds.

report — Default: report=5 Type: numeric
This sets the threshold for the number of lines modified. When more than this number of lines are modified, removed, or yanked, **vi** will report the number of lines changed at the bottom of the screen.

scroll — Default: scroll={1/2 window} Type: numeric
This is the number of lines that the screen scrolls up or down when using the <^U> and <^D> commands.

sections — Default: sections=SHNHH HU Type: string
Each two character pair of this string specify **nroff** macro names which are to be

treated as the beginning of a section by the **]]** and **[[** commands. The default string is for the **-ms** and **-mm** macros. To enter one letter **nroff** macros, use a quoted space as the second character. See **paragraphs** for a fuller explanation.

shell sh
Default: sh=from environment SHELL or /bin/sh Type: string
This is the name of the **sh** to be used for "escaped" commands.

shiftwidth sw
Default: sw=8 Type: numeric
This is the number of spaces that a <^T> or <^D> will move over for indenting, and the amount < and > shift by.

showmatch sm
Default: nosm Type: toggle
When a **)** or **}** is typed, show the matching **(** or **{** by moving the cursor to it for one second if it is on the current screen.

slowopen slow
Default: terminal dependent Type: toggle
On terminals that are slow and unintelligent, this option prevents the updating of the screen some of the time to improve speed.

tabstop ts
Default: ts=8 Type: numeric
<tab>s are expanded to boundaries that are multiples of this value.

taglength tl
Default: tl=0 Type: numeric
If nonzero, tag names are only significant to this many characters.

term
Default: (from environment **TERM**, else dumb) Type: string
This is the terminal and controls the visual displays. It cannot be changed when in "visual" mode, you have to Q to command mode, type a set term command, and do ‘‘vi.’’ to get back into visual. Or exit vi, fix $TERM, and reenter. The definitions that drive a particular terminal type are found in the file **/etc/termcap**.

terse
Default: terse Type: toggle
When set, the error diagnostics are short.

warn
Default: warn Type: toggle
The user is warned if she/he tries to escape to the shell without writing out the current changes.

window
Default: window={8 at 600 baud or less, 16 at 1200 baud, and screen size − 1 at 2400 baud or more} Type: numeric
This is the number of lines in the window whenever **vi** must redraw an entire screen. It is useful to make this size smaller if you are on a slow line.

w300, w1200, w9600
These set window, but only within the corresponding speed ranges. They are useful in an EXINIT to fine tune window sizes. For example,

set w300=4 w1200=12

causes a 4 lines window at speed up to 600 baud, a 12 line window at 1200 baud, and a full screen (the default) at over 1200 baud.

wrapscan ws
Default: ws Type: toggle
Searches will wrap around the end of the file when is option is set. When it is off, the search will terminate when it reaches the end or the beginning of the file.

wrapmargin wm
Default: wm=0 Type: numeric
Vi will automatically insert a <nl> when it finds a natural break point (usually a <sp> between words) that occurs within "wm" spaces of the right margin. Therefore with "wm=0" the option is off. Setting it to 10 would mean that any time you are within 10 spaces of the right margin **vi** would be looking for a <sp> or <tab> which it could replace with a <nl>. This is convenient for people who forget to look at the screen while they type. (In version 3, wrapmargin behaves more like nroff, in that the boundary specified by the distance from the right edge of the screen is taken as the rightmost edge of the area where a break is allowed, instead of the leftmost edge.)

writeany wa Default: nowa Type: toggle

Vi normally makes a number of checks before it writes out a file. This prevents the user from inadvertently destroying a file. When the "writeany" option is enabled, **vi** no longer makes these checks.