An Introduction to Display Editing with Vi

William Joy

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.

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.*

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

1.1. Specifying terminal type

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

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

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

% **setenv TERM** 2621

This command works with the *csh* shell. If you are using the standard Bourne shell *sh* 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.‡

‡ 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

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 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.

: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.

* 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.

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 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.*

---

† 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.

† 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.)

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 **ˆG**. 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 ``.

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. 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 **ˆH**) key which moves left one character. The key **h** works as **ˆH** 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
ˆB	backwards to previous page
ˆD	scrolls down in the file

ˆE	exposes another line at the bottom
ˆF	forward to next page
ˆG	tell what is going on
ˆH	backspace the cursor
ˆN	next line, same column
ˆP	previous line, same column
ˆU	scrolls up in the file
ˆY	exposes another line at the top
+	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 **ˆH** or **#**) to backspace over the last character which you typed, and the character which you use to kill input lines (usually **@**, **ˆX**, or **ˆU**) to erase the input you have typed on the current line.† The character **ˆW** 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.

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 **ˆH** 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.

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

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.

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.

---

† 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.

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 (**ˆI**) 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.

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 ‘ˆ’. On the screen non-printing characters resemble a ‘ˆ’ 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 **ˆV** before the control character. The **ˆV** 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 **ˆL** 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.

---

* 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.

† 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.

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 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 **ˆD**, the editor will clear the screen and continue.

On systems which support it, **ˆZ** 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 **ˆL**, 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 **ˆR** to cause the editor to retype the screen, closing up these holes.

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

6. Special topics

6.1. Editing on slow terminals

When you are on a slow terminal, 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 **ˆL**; 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**, ˆ↑, **!**
ignorecase	noic	Ignore case in searching
lisp	nolisp	**({) }** commands deal with S-expressions
list	nolist	Tabs print as ˆI; 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 **ˆD** and **ˆT**
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

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

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.

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. 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) This string should be placed in the variable EXINIT in your environment. If you use the shell *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 shell *sh*, put these lines in the file *.profile* in your home directory:

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

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

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

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.†

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 **ˆD** key to backtab over the supplied indentation.

Each time you type **ˆD** 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]]**.

† 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.

6.7. Filtering portions of the buffer

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

6.8. Commands for editing LISP

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

The *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 **ˆV**. (It may be necessary to double the **ˆV** 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**ˆVˆV**CR CR

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

Macros can be deleted with

unmap lhs

If the *lhs* of a macro is ''#0'' through ''#9'', this maps the particular function key instead of the 2 character ''#'' sequence. So that terminals without function keys can access such definitions, the form

‘‘#x’’ will mean function key *x* on all terminals (and need not be typed within one second.) The character ‘‘#’’ can be changed by using a macro in the usual way:

:map **ˆVˆVˆI** #

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 **ˆT** to be the same as 4 spaces in input mode, you can type:

:map **ˆT ˆV**b̸b̸b̸b̸

where **b̸** is a blank. The **ˆV** 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 **ˆR** 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 **ˆI** 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.

† 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	**ˆD ˆU**
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 **ˆD**. The window will revert to the last specified size the next time it is cleared and refilled.†

The scroll commands **ˆD** and **ˆU** 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 **ˆR**), 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 **ˆG**, and giving these numbers after the **:** and before the **w**, separated by **,**'s. You can also mark these

† But not by a **ˆL** 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.

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.

ˆH	deletes the last input character
ˆW	deletes the last input word, defined as by **b**
erase	your erase character, same as **ˆH**
kill	your kill character, deletes the input on this line
\	escapes a following **ˆH** and your erase and kill
ESC	ends an insertion
DEL	interrupts an insertion, terminating it abnormally
CR	starts a new line
ˆD	backtabs over *autoindent*
0ˆD	kills all the *autoindent*
↑**ˆD**	same as **0ˆD**, but restores indent next line
ˆV	quotes the next non-printing character into the file

The most usual way of making corrections to input is by typing **ˆH** to correct a single character, or by typing one or more **ˆW**'s to back over incorrect words. If you use **#** as your erase character in the normal system, it will work like **ˆH**.

Your system kill character, normally **@**, **ˆX** or **ˆU**, 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 **ˆV**. The **ˆV** 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 **ˆD**. 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 (**ˆ@**) character to appear in files. Also the LF (linefeed or **ˆJ**) 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 **ˆS** or **ˆQ**, 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 **ˆD**. 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 **ˆD** 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 **ˆR**. 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 **ˆR** 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.

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.

ˆ@ 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 **ˆ@** cannot be part of the file due to the editor implementation (7.5f).

ˆA Unused.

ˆB Backward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).

ˆC Unused.

ˆD 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 **ˆD** and **ˆU** 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.

ˆE Exposes one more line below the current screen in the file, leaving the cursor where it is if possible. (Version 3 only.)

ˆF Forward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).

ˆG 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.

ˆH (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).

ˆI (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).

ˆJ (LF**)** Same as **down arrow** (see **j**).

ˆK Unused.

ˆL 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).

ˆM (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).

ˆN Same as **down arrow** (see **j**).

ˆO Unused.

ˆP Same as **up arrow** (see **k**).

ˆQ Not a command character. In input mode, **ˆQ** quotes the next character, the same as **ˆV**, except that some teletype drivers will eat the **ˆQ** so that the editor never sees it.

ˆR 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 **ˆL**. 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 (**ˆH**) 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 **ˆJ** (linefeed) and **ˆN**.

k **Up arrow**. Moves the cursor one line up. **ˆP** 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)

.