An Abridged Introduction to Display Editing with Vi

(see the unabridged USD:15 document for more information)

William Joy

Mark Horton

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

(Abridged by Corey Satten corey@cac.washington.edu)

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.

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

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

1.1. Editing a file

To begin, you should make a copy of a file you are familiar with, and run *vi* on this file, giving the command

% **vi** *name*

replacing *name* with the name of the copy file you just created. The screen should clear and the text of your file should appear on the screen. If something else happens refer to the footnote.‡

1.2. 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.

1.3. 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.

1.4. Arrow keys

The editor command set is independent of the terminal you are using. Many terminals have arrow keys which can work within the editor, however even if your terminal has them, you may still prefer to use the easier to type **h j k** and **l** keys* for normal cursor motion and bind the arrow keys to other functions.

1.5. Special characters: ESC, CR, interrupt

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.

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

‡ If you gave the system an incorrect terminal type code then the editor may have just made a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. In this case hit the keys **:q** (colon and the q key) and then hit the RETURN key. This should get you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again.

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

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

* 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.

position in the file after a search or running a command to reformat part of the buffer. When this is happening you can stop the editor by sending an interrupt.

1.6. Getting out of the editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **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 'ˆD'. We will use this two character notation for referring to these control keys from now on. You may have a key labelled 'ˆ' on your terminal. This key will be represented as '↑' in this document; 'ˆ' is exclusively used as part of the 'ˆx' notation for control characters.‡

As you know now if you tried hitting **ˆD**, 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 **ˆU**. Many dumb terminals can't scroll up at all, in which case hitting **ˆU** 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 **ˆE** to expose one more line at the bottom of the screen, leaving the cursor where it is. The command **ˆY** (which is hopelessly non-mnemonic, but next to **ˆU** 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 **ˆF** and **ˆB** 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 **ˆD** and **ˆU** if you wish.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting **ˆF** 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 'ˆ' 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

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**.

---

ed(1). If you don't wish to learn about this yet, you can disable this more general facility by doing **:se nomagic**CR; by putting this command in EXINIT in your environment, you can have this always be in effect (more about *EXINIT* later.)

2.5. Summary

SPACE	advance the cursor one position
^B	backwards to previous page
^D	scrolls down in the file
^E	exposes another line at the bottom
^F	forward to next page
^G	tell what is going on
^H	backspace the cursor
^N	next line, same column
^P	previous line, same column
^U	scrolls up in the file
^Y	exposes another line at the top
+	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 **s** and ESC to terminate the textual insert. This sequence of commands can be used to easily pluralize a word.

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

It is often the case that you want to add new lines to the file you are editing, before or after some specific line in the file. Find a line where this makes sense and then give the command **o** to create a new line after the line you are on, or the command **O** to create a new line before the line you are on. After you create a new line in this way, text you type up to an ESC is inserted on the new line.

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

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

While you are inserting new text, you can use the characters you normally use at the system command level (usually **^H** or #) to backspace over the last character which you typed, and the character which you use to kill input lines (usually @, **^X**, or **^U**) to erase the input you have typed on the current line.† The character **^W** will erase a whole word and leave you after the space after the previous word; it is useful for quickly backing up in an insert.

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

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

3.2. Making small corrections

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

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

3.3. More corrections: operators

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

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

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

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

3.4. Operating on lines

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

Try repeating the **c** operator twice; this will change a whole line, erasing its previous contents and replacing them with text you type up to an ESC.†

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

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

† 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.

4. Moving about; rearranging and duplicating text

4.1. Low level character motions

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. **0** moves the cursor to the very beginning of the 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. 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 a double quote and 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 a named buffer.

* 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.

4.3. 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

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

5.5. 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.

5.6. Recovering lost files

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

% **vi –r** *name*

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

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.

5.7. 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

† 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.

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.

5.8. 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.

5.9. 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.

6. 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.

7. Nitty-gritty details

7.1. 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:

scroll amount	**^D ^U**
line/column number	**z G |**
repeat effect	most of the rest

The scroll commands **^D** and **^U** remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus **10a+−−−−**ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

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

7.2. 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

: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*

replaced by the current file name, or the character # which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file. Thus if you try to do a **:e** and get a diagnostic that you haven't written the file, you can give a **:w** command and then a **:e** # command to redo the previous **:e**.

You can write part of the buffer to a file by finding out the lines that bound the range to be written using ^**G**, and giving these numbers after the **:** and before the **w**, separated by **,**'s. You can also mark these 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.

7.3. 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 \.

7.4. 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 **^D**. The editor will move the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a **0** followed immediately by a **^D** if you wish to kill all the indent and not have it come back on the next line.

7.5. 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.

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.