Introduction to the Software Tools Text Editor

T. Allen Akin
Terrell L. Countryman
Perry B. Flinn
Daniel H. Forsyth, Jr.
Jeanette T. Myers
Arnold D. Robbins
Peter N. Wan

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

September, 1984

TABLE OF CONTENTS

Foreword

'Ed' is an interactive program that can be used for the creation and modification of "text." "Text" may be any collection of character data, such as a report, a program, or data to be used by a program.

This document is intended to provide the beginning user of 'ed' with a tutorial, an aid to becoming familiar with editing. It does not attempt to cover the editor in full; only the most frequently used aspects are mentioned. For details on advanced uses, a careful reading of _Software Tools_ and the _Software Tools Subsystem Reference Manual_ is recommended.

How To Use This Guide

This tutorial includes a step-by-step journey through an editing session. You should be sitting at a terminal and running the Software Tools Subsystem, so that you can perform the suggested exercises as you go.

Throughout the text of this guide are sample editing commands, which you can execute on your terminal to get a feel for their actual effect. If at any time your terminal session produces results different from those shown in the text, carefully re-check what you have typed, or consult someone in charge of your installation.

Tutorial

Starting an Editing Session

We assume that you have successfully logged in to your computer and are running the Software Tools Subsystem. If you need assistance, see the _Software Tools Subsystem Tutorial_. We further assume that you know how to use the character erase and line delete characters, so that you will have no trouble correcting typographical errors, and that you have some idea of what a "file" is.

Since you are in the Subsystem, the command interpreter should have just printed the prompt "]". To enter the text editor, type

```
          ] ed     (followed by a newline)
```

(Throughout this guide, boldface is used to indicate information that you should type in. Things typed by 'ed' are shown in the regular font.) You are now in the editor, ready to go. Note that 'ed' does not print any prompting information; this quiet behavior is preferred by experienced users. (If you would like a prompt, it can be provided; try the command "op/prompt/".)

At this point, 'ed' is waiting for instructions from you. You can instruct 'ed' by using "commands," which are single letters (occasionally accompanied by other information, which you will see shortly).

Entering Text - the Append Command

The first thing that you need is text to edit. Working with 'ed' is like working with a blank sheet of paper; you write on the paper, alter or add to what you have written, and either file the paper away for further use or throw it away. In 'ed's terminology, the blank sheet of paper you start with is called a "buffer." The buffer is empty when you start editing. All editing operations take place in the buffer; nothing you do can affect any file unless you make an explicit request to transfer the contents of the buffer to a file.

So the first problem reduces to finding a way to put text into the buffer. The "append" command is used to do this:

```
          a
```

This command appends (adds) text lines to the buffer, as they are typed in.

To put text into the buffer, simply type it in, terminating each line with a newline:

```
          The quick brown fox
              jumps over
             the lazy dog.
          .
```

To stop entering text, you must enter a line containing only a period, immediately followed by a newline, as in the last line above. This tells 'ed' that you are finished writing on the buffer, and are ready to do some editing.

The buffer now contains:

```
          The quick brown fox
              jumps over
             the lazy dog.
```

Neither the append command nor the final period are included in the buffer -- just the text you typed in between them.

Writing text on a file - the Write command

Now that you have some text in the buffer, you need to know how to save it. The write command "w" is used for this purpose. It is used like this:

```
          w file
```

where "file" is the name of the file used to store what you just typed in. The write command copies the contents of the buffer to the named file, destroying whatever was previously in the file. The buffer, however, remains intact; whatever you typed in is still there. To indicate that the transfer of data was successful, 'ed' types out the number of lines written. In this example, 'ed' would type:

```
          3
```

It is advisable to write the contents of the buffer out to a file periodically, to insure that you have an up-to-date version in case of some terrible catastrophe (like a system crash).

Finishing up - the Quit command

Now that you have saved your text in a file, you may wish to leave the editor. The "quit" command "q" is provided for this:

```
          q
```

The next thing you see should be the "]" prompt from the Subsystem command interpreter. If you did not write out the contents of the buffer, the editor would respond:

```
          ?
          (not saved)
```

This is to remind you to write out the buffer, so that the results of your editing session are not lost. If you intended that the buffer be discarded, just enter "q" again and 'ed' will throw away the buffer and terminate.

When you receive the "]" prompt from the Subsystem command interpreter, the buffer has been thrown away; there is absolutely no way to recover it. If you wrote the contents of the buffer to a file, then this is of no concern; if you did not, it may mean disaster.

To check if the text you typed in is really in the file you wrote it to, try the following command:

```
] cat file
```

where "file" is the name of the file given with the "w" command. ("Cat" is a Subsystem command that can be used to print files on the terminal. If, for example, you wished to print your file on the line printer, you could say:

```
] pr file
```

and the contents of "file" would be queued for printing.)

Reading files - the Enter command

Of course, most of the time you will not be entering text into the buffer for the first time. You need a way to fill the buffer with the contents of some file that already exists, so that you can modify it. This is the purpose of the "enter" command "e"; it enters the contents of a file into the buffer. To try out "enter," you must first get back into the editor:

```
] ed
```

"Enter" is used like this:

```
e file
```

"File" is the name of a file to be read into the buffer.

Note that you are not restricted to editing files in the current directory; you may also edit files belonging to other users (provided they have given you permission). Files belonging to other users must be identified by their full "pathname" (discussed fully in _User's Guide to the Primos File System_). For example, to edit a file named "document" belonging to user "tom," you would enter the following command:

```
e //tom/document
```

After the file's contents are copied into the buffer, 'ed' prints the number of lines it read. In our example, the buffer would now contain:

```
          The quick brown fox
              jumps over
             the lazy dog.
```

If anything at all is present in the buffer, the "e" command destroys it before reading the named file.

As a matter of convenience, 'ed' remembers the file name specified on the last "e" command, so you do not have to specify a file name on the "w" command. With these provisions, a common editing session looks like

```
          ] ed
          e file
          {editing}
          w
          q
```

The "file" command ("f") is available for finding out the remembered file name. To print out the name, just type:

```
          f
          file
```

You might also want to check that

```
          ] ed file
```

is exactly the same as

```
          ] ed
          e file
```

That is, 'ed' performs an "e" command for you if you give it a file name on the command line.

Errors - the Query command

Occasionally, an error of some kind is encountered. Usually, these are caused by misspelled file names, although there are other possibilities. Whenever an error occurs, 'ed' types

```
          ?
```

Although this is rather cryptic, it is usually clear what caused the problem. If you need further explanation, just enter "?" and 'ed' responds with a one-line explanation of the error. For example, if the last command you typed was an "e" command, 'ed' is probably saying that it could not find the file you asked for. You can find out for sure by entering "?":

```
          e myfile
          ?
          ?
          I can't open the file to read
```

Except for the messages in response to "?", 'ed' rarely gives other, more verbose error messages; if you should see one of these, the best course of action is to report it to whoever maintains the editor at your installation.

Printing text - the Print command

You are likely to need to print the text you have typed to check it for accuracy. The "print" command "p" is available to do this. "P" is different from the commands seen thus far; "e", "w", and "a" have been seen to work on the whole buffer at once. For a small file, it might be easiest to print the entire buffer just to check on some few lines, but for very large files this is clearly impractical. The "p" command therefore accepts "line numbers" that indicate which lines to print. Try the following experiment:

```
          ] ed file
          3
          1p
          The quick brown fox
          3p
             the lazy dog.
          1,2p
          The quick brown fox
              jumps over
          1,3p
          The quick brown fox
              jumps over
             the lazy dog.
```

"1p" tells 'ed' to print line 1 ("The quick brown fox"). "3p" says to print the third line ("the lazy dog."). "1,2p" tells 'ed' to print the first through the second lines, and "1,3p" says to print the first through the third lines.

Suppose we want to print the last line in the buffer, but we don't know what its number is. 'Ed' provides an abbreviation to specify the last line in the buffer:

```
          $p
             the lazy dog.
```

The dollar sign can be used just like a number. To print everything in the buffer, we could type:

```
          1,$p
          The quick brown fox
              jumps over
             the lazy dog.
```

If for some reason you want to stop the printing before it is done, press the BREAK key on your terminal. If you receive no response from BREAK, 'ed' is waiting for you to enter a command. Otherwise, 'ed' responds with

```
          ?
```

and waits for your next command.

More Complicated Line Numbers

'Ed' has several ways to specify lines other than just numbers and "$". Try the following command:

```
          p
             the lazy dog.
```

'Ed' prints the last line. Does 'ed' always print the last line when it is given an unadorned "p" command? No. The "p" command by itself prints the "current" line. The "current" line is the last line you have edited in any way. (As a matter of fact, the last thing we did was to print all the lines in the buffer, so the last line was edited by being printed.) 'Ed' allows you to use the symbol "." (read "dot") to represent the current line. Thus

```
          .p
             the lazy dog.
```

is the same as

```
          .,.p
             the lazy dog.
```

which is the same as just

```
          p
             the lazy dog.
```

"." can be used in many ways. For example,

```
          1,2p
          The quick brown fox
              jumps over
          1,.p
          The quick brown fox
              jumps over
          .,$p
              jumps over
             the lazy dog.
```

This example shows how to print all the lines up to the current line (1,.p) or all the lines from the current line to the end of the buffer (.,$p). If for some reason you would like to know the number of the current line, you can type

```
          .=
          3
```

and 'ed' displays the number. (Note that the last thing we did was to print the last line, so the current line became line 3.)

"." is not particularly useful when used alone. It becomes much more important when used in "line-number expressions." Try this experiment:

```
          .-1p
              jumps over
```

".-1" means "the line that is one line before the current line."

```
          .+1p
             the lazy dog.
```

".+1" means "the line that is one line after the current line."

```
          .-2,.-1p
          The quick brown fox
              jumps over
```

".-2,.-1p" means "print the lines from two lines before to one line before the current line."

You can also use "$" in line-number expressions:

```
          $-1p
              jumps over
```

"$-1p" means "print the line that is one line before the last line in the buffer, i.e., the next to the last line."

Some abbreviations are available to help reduce the amount of typing you have to do. Typing a newline by itself is equivalent to typing ".+1p"; typing a caret, "^", or a single minus sign, "-", followed by a newline is equivalent to typing ".-1p"; and typing a line-number expression followed by a newline is equivalent to typing that line-number expression followed by "p". Examples:

```
          {type a newline by itself}
             the lazy dog.
          ^
              jumps over
          -
          The quick brown fox
          1
          The quick brown fox
```

It might be worthwhile to note here that almost all commands expect line numbers of one form or another. If none are supplied, 'ed' uses default values. Thus,

```
          w file
```

is equivalent to

```
          1,$w file
```

and

```
          a
```

is equivalent to

```
          .a
```

(which means, append text <u>after</u> the current line.)

Deleting Lines

As yet, you have seen no way of removing lines that are no longer wanted or needed. To do this, use the "delete" command "d":

```
          1,2d
```

deletes the first through the second lines. "D" expects line numbers that work in the same way as those specified for "p", deleting one line or any range of lines.

```
          d
```

deletes only the current line. It is the same as ".d" or ".,.d".

After a deletion, the current line pointer is left pointing to the first line <u>after</u> the group of deleted lines, unless the last line in the buffer was deleted. In this case, the current line is the last line <u>before</u> the group of deleted lines.

Text Patterns

Frequently it is desirable to be able to find a particular "pattern" in a piece of text. For example, suppose that after proofreading a report you have typed in using 'ed' you find a spelling error. There must be an easy way to find the misspelled word in the file so it can be corrected. One way to do this is to count all the lines up to the line containing the error, so that you can give the line number of the offending line to 'ed'. Obviously, this way is not very fast or efficient. 'Ed' allows you to "search" for patterns of text (like words) by enclosing the pattern in slashes:

```
          /jumps/
              jumps over
```

'Ed' looks for the pattern you specified, and moves to the first line which contains the pattern. Note that if we had typed

```
          /jumped/
          ?
```

'ed' would inform us that it could not find the pattern we wanted.

'Ed' searches forward from the current line when it attempts to find the pattern you specified. If 'ed' reaches the last line without seeing the pattern, it "wraps around" to the first line in the file and continues searching until it either finds the pattern or gets back to the line where it started (line "."). This procedure ensures that you get the "next" occurrence of the pattern you were looking for, and that you don't miss any occurrences because of your current position in the file.

Suppose, however, that you do not wish to find the "next" occurrence of a word, but the previous one instead. Very few text editors provide this capability; however, 'ed' makes it simple. Just surround the pattern with backslashes:

```
          \quick\
          The quick brown fox
```

Remember: backslashes search backward. The backward search (or backscan, as it is sometimes called) wraps around the file in a manner similar to the forward search (or scan). The search begins at the line before the current line, proceeds until the first line of the file is seen, then begins at the last line of the file and searches up until the current line is encountered. Once again, this is to ensure that you do not miss any occurrences of a pattern due to your current position in the file.

In pattern searches, and in other commands which we will get to later, 'ed' allows you to leave off the trailing the delimiter. I.e., instead of typing

```
          /jumps/
```

you can type

```
          /jumps
```

to search forward for the first occurrence of the pattern "jumps". Similarly, to search backwards, you may type

```
          \quick
```

instead of

```
          \quick\
```

This feature can save considerable time and frustration when you are doing some involved editing, and accidentally leave off the trailing delimiter ("/" or "\"). The rest of this guide will continue to use examples with the trailing delimiter, but you do not have to in your actual editing.

'Ed' also provides more powerful pattern matching services than simply looking for a given string of characters. (A note to beginning users: this section may seem fairly complicated at first, and indeed you do not really need to understand it completely for effective use of the editor. However, the results you might get from some patterns would be mystifying if you were not provided with some explanation, so look this over once and move on.)

The pattern that may appear within slashes (or backslashes) is called a "regular expression." It contains characters to look for and special characters used to perform other operations. The following characters

```
          %  ?  $  [  *  @  {
```

have special meaning to 'ed':

%   Beginning of line. The "%" character appearing as the first element in a pattern matches the beginning of a line. It is most frequently used to locate lines with some string at the very beginning; for example,

```
          /%The/
```

finds the next line that begins with the word "The". The percent sign has its special meaning _only if it is the first element of the pattern_; otherwise, it is treated as a literal percent sign.

?   Any character. The question mark "?" in a regular expression matches _any_ character (except a beginning-of-line or a newline). It can be used like this:

```
          /a?b/
```

to find strings like

```
          a+b
          a-b
          a b
          arbitrary
```

However, "?" is most often used with the "closure" operator "*" (see below).

$ End of line. The dollar sign appearing as the last element of a pattern matches the newline character at the end of a line. Thus,

```
          /today$/
```

can be used to find a line with the word "today" at the very end. Like the percent sign, the dollar sign has no special meaning in positions other than the end of a pattern.

[] Character classes. The square brackets are used to match "classes" of characters. For example,

```
          /[A-Z]/
```

finds the next line containing a capital letter,

```
          /%[abcxyz]/
```

finds the next line beginning with an a, b, c, x, y, or z, and

```
          /[~0-9]/
```

finds the next line which contains a non-digit. Character classes are also frequently used with the "closure" operator "*".

* Closure. The asterisk is used to mean "any number of repetitions (including zero) of the previous pattern element (one character or a character class in brackets)." Thus,

```
          /a?*b/
```

finds lines containing an "a" followed by any number of characters and a "b". For example, the following lines are matched:

```
          ab
          abnormal
          Recording Media, by Dr. Joseph P. Gunchy
```

As another example,

```
          /%=*$/
```

matches only those lines containing all equal-signs (or nothing at all). If you wish to ensure that only non-empty lines are matched, use

```
          /%==*$/
```

Always remember that "*" (closure) matches <u>zero</u> or more repetitions of an element.

@ Escape. The "at" sign has special meaning to 'ed'. It is the "escape" character, which is used to prevent interpretation of a special character which follows. Suppose you wish to locate a line containing the string "a * b". You may use the following command:

```
/a @* b/
```

The "at" sign "turns off" the special meaning of the asterisk, so it can be used as an ordinary text character. You may have occasion to escape any of the regular expression metacharacters (%, ?, $, [, *, @, or {) or the slash itself. For example, suppose you wished to find the next occurrence of the string "1/2". The command you need is:

```
/1@/2/
```

{} Pattern tags. As seen in the next section, it is sometimes useful to remember what part of a line was actually matched by a pattern. By default, the string matched by the entire pattern is remembered. It is also possible to remember a string that was matched by only a part of a pattern by enclosing that part of the pattern in braces. Hence to find the next line that contains a quoted string and remember the text between the quotes, we might use

```
/"{?*}"/
```

If the line thus located looked like this

```
This is a line containing a "quoted string".
```

then the text remembered as matching the tagged part of the pattern would be

```
quoted string
```

The last important thing you need to know about patterns is the use of the "default" pattern. 'Ed' remembers the last pattern used in any command, to save you the trouble of retyping it. To access the remembered pattern, simply use an "empty" string. For example, the following sequence of commands could be used to step through a file, looking for each occurrence of the string "ICS":

```
/ICS/
//
//
(and so on)
```

One last comment before leaving pattern searching. The constructs

```
          /pattern/
          \pattern\
```

are not separate commands; they are components of line number expressions. Thus, to print the line after the next line containing "tape", you could say

```
          /tape/+1p
```

Or, to print a range of lines from one before to one after a line with a given pattern, you could use

```
          /pattern/-1,/pattern/+1p
```

Making Substitutions - the Substitute command

This is one of the most used editor commands. The "substitute" command "s" is used to make small changes within lines, without retyping them completely. It is used like this:

```
          starting-line,ending-line s [/pattern/new-stuff[/]]
```

For instance, suppose our buffer looks like this:

```
          1,$p
          The quick brown fox
              jumps over
             the lazy dog.
```

To change "jumps" to "jumped,"

```
          2s/jumps/jumped/p
              jumped over
```

Note the use of the trailing "p" to print the result. If the "p" had been omitted, the change would have been performed (in the buffer) but the changed line would not have been printed out.

If the last string specified in the substitute command is empty, then the text matching the pattern is deleted:

```
          s/jumped//p
               over
          s/% */    jumps /p
              jumps over
```

Recalling that a missing pattern means "use the last pattern specified," try to explain what the following commands do:

```
          s///p
          jumps over
          s//    /p
              jumps over
```

(Note that, like many other commands, the substitute command assumes you want to work on the current line if you do not specify any line numbers.)

What if you want to change "over" into "over and over"? You might use

```
          s/over/over and over/p
              jumps over and over
```

to accomplish this. There is a shorthand notation for this kind of substitution that was alluded to briefly in the last section. (Recall the discussion of "tagged" patterns.) By default, the part of a line that was matched by the whole pattern is remembered. This string can then be included in the replacement string by typing an ampersand ("&") in the desired position. So, instead of the command in the last example,

```
          s/over/& and &/
```

could have been used to get the same result. If a portion of the pattern had been tagged, the text matched by the tagged part in the replacement could be reused by typing "@1":

```
          s/jump{?*}/vault@1/p
              vaults over and over
```

It is possible to tag up to nine parts of a pattern using braces. The text matched by each tagged part may then be used in a replacement string by typing

```
          @n
```

where n corresponds to the nth "{" in the pattern. What does the following command do?

```
          s/{[~ ]*} {?*}/@2 @1/
```

Some more words on substitute: the slashes are known as "delimiters" and may be replaced by any other character except a newline, as long as the same character is used consistently throughout the command. Thus,

```
          s#vaults#vaulted#p
              vaulted over and over
```

is legal. Also, note that substitute changes only the first occurrence of the pattern that it finds; if you wish to change all occurrences on a line, you may append a "g" (for "global") to the command, like this:

```
          s/ /*/gp
          ****vaulted*over*and*over
```

In the replacement part of a substitute command, the character "&", as the only character in the pattern, means "the replacement part of the previous substitute command". (This allows an empty replacement pattern as well.) Thus, to step through the buffer, and change selected occurrences of one pattern into another, you might do the following:

```
          /pat1/
          Line containing pat1.
          s/pat1/stuff1/p
          Line containing stuff1.
          //
          Another line with pat1.
          //
          Yet another line with pat1.
          s//&/p
          Yet another line with stuff1.
```

You may leave off the trailing delimiter in the substitute command. This will cause 'ed' to print out the changed line. I.e., "s/stuff/junk" is the same as "s/stuff/junk/p".

```
          /quick/
          The quick brown fox
          s/quick/really fast
          The really fast brown fox
```

If you wish to delete an occurrence of a pattern, you may leave it off. 'Ed' will delete the pattern, and then print the line. In other words, "s/stuff" is the same as "s/stuff//p".

```
          p
          The quick brown fox
          s/quick
          The  brown fox
```

Finally, you may leave off the search pattern and replacement string entirely. If you do, 'ed' will behave as though you had typed "s//&/p", in other words, substitute the previous replacement pattern for the previous search pattern, and print.

```
          1,$d
          a
          line 1
          line 2
          .
          1s/line/this is &/p
          this is line 1
          2s
          this is line 2
```

This can save considerable typing.

Line Changes, Insertions, and Concatenations

Two "abbreviation" commands are available to shorten common operations applying to changes of entire lines. These are the "change" command "c" and the "insert" command "i".

The change command is a combination of delete and append. Its format is

```
          starting-line,ending-line c
```

This command deletes the given range of lines, and then goes into append mode to obtain text to replace them. Append mode works exactly the same way as it does for the "a" command; input is terminated by a period standing alone on a line. Examine the following editing session to see how change might be used:

```
          1,$c
          Ed is an interactive program used for
          the creation and modification of "text.
          .
          c
          the creation and modification of "text."
          "Text" may be any collection of character
          data.
          .
```

As you can see, the current line is set to the last line entered in append mode.

The other abbreviation command is "i". "I" is very closely related to "a"; in fact, the following relation holds:

```
          starting-line i
```

is the same as

```
          starting-line - 1  a
```

In short, "i" inserts text <u>before</u> the specified line, whereas "a" inserts text <u>after</u> the specified line.

The join command "j" can be used to put two or more lines together into a single line. It works like this:

```
          starting-line,ending-line j[/string[/]]
```

The defaults for starting-line and ending-line are "^" and "." respectively, that is, "join the line before the current line to the current line". You may specify in "string" what is to replace the newline(s) which currently separate the lines which are to be joined. If you do not specify any string, 'ed' will replace the newline with a single blank. If you do specify a string, you may leave off the trailing delimiter (which can be any character), and 'ed' will print out the resulting joined line. An extended example should make this clear:

```
          1,$p
          The quick brown fox
              jumps over
             the lazy dog.
          2,$s/%  *//
          1,$p
          The quick brown fox
          jumps over
          the lazy dog.
          1,2j
          The quick brown fox jumps over
          1,2j/ the back of /p
          The quick brown fox jumps over the back of the lazy dog.
```

Moving Text

Throughout this guide, we have concentrated on what may be called "in-place" editing. The other type of editing commonly used is often called "cut-and-paste" editing. The move command "m" is provided to facilitate this kind of editing, and works like this:

```
          starting-line,ending-line m after-this-line
```

If you wanted to move the last fifty lines of a file to a point after the third line, the command would be

```
          $-49,$m3
```

Any of the line numbers may, of course, be full expressions with search strings, arithmetic, etc.

You may, if you like, append a "p" to the move command to cause it to print the last line moved. The current line is set to the last line moved.

Global Commands

The "global" command "g" is used to perform an editing command on all lines in the buffer that match a certain pattern. For example, to print all the lines containing the word "editor", you could type

```
          g/editor/p
```

If you wanted to correct some common spelling error, you would use

```
          g/old-stuff/s//new-stuff/gp
```

which makes the change in all appropriate lines and prints the resulting lines. Another example: deleting all lines that begin with an asterisk could be done this way:

```
          g/%@*/d
```

"G" has a companion command "x" (for "exclude") that performs an operation on all lines in the buffer that do _not_ match a given pattern. For example, to delete all lines that do _not_ begin with an asterisk, use

```
          x/%@*/d
```

"G" and "x" are very powerful commands that are essential for advanced usage, but are usually not necessary for beginners. Concentrate on other aspects of 'ed' before you move on to tackle global commands.

Marking Lines

During some types of editing, especially when moving blocks of text, it is often necessary to refer to a line in the buffer that is far away from the current line. For instance, say you want to move a subroutine near the beginning of a file to somewhere near the end, but you aren't sure that you can specify patterns to properly locate the subroutine. One way to solve this problem is to find the first line of the subroutine, then use the command ".=":

```
          /subroutine/
             subroutine think
          .=
          47
```

and write down (or remember) line 47. Then find the end of the subroutine and do the same thing:

```
          /end/
             end
          .=
          71
```

Now you move to where you want to place the subroutine and enter the command

```
          47,71m.
```

which does exactly what you want.

The problem here is that absolute line numbers are easily forgotten, easily mistyped, and difficult to find in the first place. It is much easier to have 'ed' remember a short "name" along with each line, and allow you to reference a line by its name. In practice, it seems convenient to restrict names to a single character, such as "b" or "e" (for "beginning" or "end"). It is not necessary for a given name to be uniquely associated with one line; many lines may bear the same name. In fact, at the beginning of the editing session, all lines are marked with the same name: a single space.

To return to our example, using the 'k' command, we can mark the beginning and ending lines of the subroutine quite easily:

```
          /subroutine/
             subroutine think
          kb
          /end/
             end
          ke
```

We have now marked the first line in the subroutine with "b" and the second line with "e".

To refer to names, we need more line number expression elements: ">" and "<". Both work in line number expressions just like "$" or "/pattern/". The symbol ">" followed by a single character mark name means "the line number of the first line with this name when you search forward". The symbol "<" followed by a single character mark name means "the line number of the first line with this name when you search backward". (Just remember that '<' points backward and '>' points forward.)

Now in our example, once we locate the new destination of the subroutine, we can use "<b" and "<e" to refer to lines 47 and 71, respectively (remember, we marked them). The "move" command would then be

```
          <b,<em.
```

Several other features pertaining to mark names are important. First, the 'k' command does not change the current line '.'. You can say

```
          $kx
```

(which marks the last line with "x") and "." will not be changed. If you want to mark a range of lines, the 'k' command accepts two line numbers. For instance,

```
          5,10ka
```

marks lines 5 through 10 with "a" (i.e., gives each of lines 5 through 10 the markname "a").

The 'n', '!' and apostrophe commands also deal with marks. The 'n' command performs two functions. If it is invoked without a mark name following it, like

```
          $n
```

it prints the mark name of the line. In this case, it would print the mark name of the last line in the file. If the 'n' command is followed by a mark name, like

```
4nq
```

it marks the line with that mark name, and erases the marks on any other lines with that name. In this case, line 4 is marked with "q" and it is guaranteed that no other line in the file is marked with "q".

The '!' and apostrophe commands are both global commands that deal with mark names. The apostrophe command works very much like the 'g' command: the apostrophe is followed by a mark name and another command; the command is performed on every line marked with that name. For instance,

```
'as/fox/rabbit/
```

changes the first "fox" to "rabbit" on every line that is named "a". The '!' command works in the same manner, except that it performs the command on those lines that _are_ _not_ marked with the specified name. For example, to delete all lines not named "k", you could type

```
!kd
```

Undoing Things -- the Undo Command

Unfortunately, Murphy's Law guarantees that if you make a mistake, it will happen at the worst possible time and cause the greatest possible amount of damage. 'Ed' attempts to prevent mistakes by doing such things as working with a copy of your file (rather than the file itself) and checking commands for their plausibility. However, if you type

```
d
```

when you really meant to type

```
a
```

'ed' must take its input at face value and do what you say. It is at this point that the "undo" command 'u' becomes useful. "Undo" allows you to "undelete" the last group of lines that was deleted from the buffer. In the last example, some inconvenience could be avoided by typing

```
^ud
```

which restores the deleted line. (By default "undo" _replaces_ the specified line by the last group of lines deleted. Specifying the "d", as in "ud", causes the group to be inserted _after_ the specified line instead.)

The problem that arises with "undo" is the answer to the question: "What was the last group of lines deleted?" This answer is very dependent on the implementation of 'ed' and in some cases is subject to change. After many commands, the last group of lines deleted is well-defined, but unspecified. It is not a good idea to use the "undo" command after anything other than 'c', 'd', or 's'. After a 'c' or 'd' command,

```
          ud
```

places the last group of deleted lines <u>after</u> <u>the</u> <u>current</u> <u>line</u>. After an 's' command (which by the way, deletes the old line, replacing it by the changed line),

```
          u
```

deletes the current line and replaces it by the last line deleted -- it exactly undoes the effects of the 's' command. But beware! If the 's' command covered a range of lines, 'u' can only restore the last of the lines in which a substitution was made; the others are gone forever.

You should be warned that while "undo" works nicely for repairing a single 'c', 'd', or 's' command, it cannot repair the damage done by one of these commands under the control of a global prefix ('g', 'x', '!' and apostrophe). Since the global prefixes cause their command to be performed many times, only the very last command performed by a global prefix can be repaired.

More Line Number Syntax

So far, the commands that you have seen can be given either no line numbers elements (the command tries to make an intelligent assumption about the line(s) on which to perform an operation), one line number element (the command acts only on that line), or two line numbers separated by a comma (the command acts on the given range of lines). There is one more way to specify line number elements, and that is to separate them by a semicolon. When line number elements are separated by semicolons, each line number element encountered sets the "current line" marker before the next line number element is evaluated. This is especially useful when using patterns as line number elements; some examples will illustrate what we mean.

Suppose that you wanted to print all the lines which lie between two lines, each containing the string "fred". An initial effort might yield the following command line:

```
          /fred/,/fred/p
```

This, however, will only print out the first line which contains "fred" after the current line. This is because both patterns will start their search after the current line where the command was executed, instead of the second one starting where the first pattern was found. To correct this, we would issue the following:

```
          /fred/;/fred/p
```

When the first occurrence of "fred" is found, the "current line" is set to that line, and the second occurrence of "fred" will be found starting at this new line. This will print the lines between two succeeding occurrences of "fred" from the current line.

As a final example, suppose that we wanted to print the lines between the second and third occurrence of "fred" after the current line; to do this, we would do:

```
          /fred/;//;//p
```

The first pattern search would find "fred", the next two null strings will cause the previous pattern ("fred") to be searched for again, each time resetting the "current line" marker. Of course, the command "p" may be replaced by any command you wish.

For both comma-separated and semicolon-separated line number elements, you may specify more than two such elements, as the above example shows; only the last two such elements will be used as the range for the given command. In general, using more than two line number elements separated by commas is not too useful, because the "current line" is not modified for any of the line number expression evaluations. Also, using integer line numbers means that multiple expressions (more than two) are not useful, since the equivalent behavior can be obtained by specifying only the last two line numbers.

Escaping to the Shell

With Version 9 of Software Tools and Revision 19.2 or later of PRIMOS, it is now possible to call the Software Tools Subsystem command interpreter (the shell) from within a program.

'Ed' provides access to this facility with the shell escape "˜" command. It works like this:

```
          ˜[<Software Tools Command>]
```

If present, the <Software Tools Command> is passed to the shell to be executed. Otherwise, an interactive shell is created. After either the command or the shell exits, 'ed' prints a "˜" to indicate that the shell escape has completed. If the first character of the <Software Tools Command> is a "!", then the "!" is replaced with the text of the previous shell command. An unescaped "%" in the <Software Tools Command> will be replaced with the current saved file name. If the shell command is expanded, 'ed' will echo it first, and then execute it.

This feature is useful when you want to temporarily stop editing and do something else, or find something out, without having write your file and leave the editor.

```
          {editing session}
          ~lf -l %
          lf -l file
          sam  a/r  06/17/84  16:25:08        19463  sys  file
          ~
```

For a deeper discussion of using the shell from within a program, see the help on the 'shell' subroutine. In particular, due to operating system constraints, you _must_ _not_ run another instance of the editor from the new shell, or you will end up clobbering your current edit buffer.

WARNING: Until Prime supports EPFs, and the editor is reloaded in EPF format, you _must_ _not_ run any external commands (like 'lf') from a shell started from 'ed'. If you do, the new program will load _over_ 'ed', and wipe out your current editing session. You can use commands which are internal to the shell (like 'cd'), without any ill effect. This restriction, for various arcane reasons, does _not_ apply to the Subsystem screen editor, 'se'.

In essence, this feature is provided in the editor with an eye to the future.

Summary

This concludes our tour through the world of text editing. In the section that follows, you will find a brief introduction to the Software Tools Subsystem screen editor 'se', which supports all of the line-oriented commands of 'ed' as well as full screen editing capabilities, while giving you a "window" into your edit buffer. Following that, we have included for your convenience a short summary of all available line editing commands supported by 'ed' and 'se', many of which were not discussed in this introduction, but which you will undoubtedly find useful.

The Subsystem Screen Editor

The screen editor, 'se', is an extended version of the Subsystem line editor, 'ed'. Although 'se' contains a number of additional features, it accepts all 'ed' commands (almost without exception), and is therefore easily used by anyone familiar with 'ed'. This section outlines the differences between 'ed' and 'se'.

The screen editor has a built-in "help" facility, which documents all the commands and options. When in doubt, type "help", and the help screens should guide you to further information on what you need to know.

Invoking the Screen Editor

You can invoke the screen editor with either of the following commands:

```
] se
```

or

```
] se myfile
```

'Se' will automatically fetch your terminal type from the Subsystem. If you never told the Subsystem your terminal type or set an unknown terminal type with the 'term' command, 'se' will prompt you for another terminal type; if you type a '?', 'se' will give you a list of possible terminal types and prompt you again for yours.

'Se' can also be invoked by the command 'e'. 'E' remembers the name of the last file you edited, so if you don't specify a file, 'e' will enter the last file you edited.

Using 'Se'

'Se' first clears the screen, draws in its margins, and executes the commands in the file "=home=/.serc", if it exists. It then processes the command line, obeying the options given there, and begins reading your file (if you specified one). The screen it draws looks something like this. (The parenthesized numerals are not part of the screen layout, but are there to aid in the following discussion.)

```
          (1) (2)              (3)
          A      |
          B     *|   integer a
          C      |
          .  ->  |   for (a = 1; a <= 12; a = a + 1)
          E      |      call putch (NEWLINE, STDOUT)
          F      |   stop
          $      |   end
          cmd>   _  (4)
          11:39  myfile ....(5)................................
```

The display is divided into five parts: (1) the line number area, (2) the mark name area, (3) the text area, (4) the command line, and (5) the status line. The current line (remember ".") is indicated by the symbol "." in the line number area of the screen. In addition, a rocket ("->") is displayed to make the current line more obvious. The current mark name of each line is shown in the markname area just to the left of the vertical bar. Other information, such as the number of lines read in, the name of the file, and the time of day, are displayed in the status line.

The cursor is positioned at the beginning of the command line, showing you that 'se' awaits your command. You may now enter any of the 'ed' commands and 'se' will perform them, while making sure that the current line is always displayed on the screen. There are only a few other things that you need know to successfully use 'se'.

. 'Se' always recognizes BS (control-h) and DEL as the erase and kill characters, regardless of your Subsystem erase and kill character settings.

. If you make an error, 'se' automatically displays an error message in the status line. It also leaves your command line intact so that you may change it using in-line editing commands (we'll get to this a little later). If you don't want to bother with changing the command, just hit DEL and 'se' will erase it.

. The "p" command has a different meaning than in 'ed'. When used with line numbers, it displays as many of the lines in the specified range as possible (always including the last line). When used without line numbers, "p" displays the previous page.

. The ":" command positions a specified line at the top of the screen (e.g., "12:" positions the screen so that line 12 is at the top). If no line number is specified, ":" displays the next page.

. The "v" command can be used to modify an entire line rather than just add to the end of the line. Also, if you use "v" over a range of lines and find that you want to terminate the command before all lines have been considered, the control-f key is used instead of a period.

. If a file name is specified in the "w" command and the file already exists, 'se' will display "file already exists"; entering the command again (by typing a NEWLINE) will cause the file to be overwritten. Given the command "w! <file>", 'se' will never warn about the destruction of an existing file.

Keeping these few differences in mind, you will see that 'se' can perform all of the functions of 'ed', while giving the advantage of a "window" into the edit buffer.

Extended Line Numbers

'Se' has a number of features that take advantage of the window display to minimize keystrokes and speed editing. In the line number area of the screen, 'se' always displays for each line a string that may be used in a command to refer to that line. Normally, it displays a capital letter for each line, but in "absolute line number" mode (controlled by the "oa" command; see the section on options for more details), it displays the ordinal number of the line in the buffer.

The line number letters displayed by 'se' may be used in any context requiring a line number. For instance, in the above example, a change to the first line on the screen could be specified as

```
          As/%/# my new program/
```

You could delete the line before the first line on the screen by typing

```
          A-1d
```

Finally, 'se' accepts "#" as a line number element; it always refers to the first line on the screen; like the line number letters, it may be used in any context which requires a line number element or expression.

Case Conversion

When 'se' is displaying upper-case letters for line numbers, it accepts command letters only in lower case. For those who edit predominantly upper-case text this is somewhat inconvenient; for those with upper-case only terminals this is a disaster. For this reason, 'se' offers several options to alleviate this situation.

First of all, typing a control-z causes 'se' to invert the case of all letters (just like the alpha-lock key on some terminals). Upper-case letters are converted to lower-case, lower-case letters are converted to upper-case, and all other characters are unchanged. You can type control-z at any time to toggle the case conversion mode. When case inversion is in effect, 'se' displays the word "CASE" in the status line.

One drawback to this feature is that 'se' still expects line numbers in upper case and commands in lower case, so you must shift to type the command letter -- just the reverse of what you're used to. A more satisfactory solution is to specify the "c" option. Just type

```
oc
```

on the command line and 'se' toggles the case conversion mode, and completely reverses its interpretation of upper and lower case letters. In this mode, 'se' displays the line number letters in lower case and expects its command letters in upper case. Unshifted letters from the terminal are converted to upper case and shifted letters to lower case.

Tabs

In the absence of tabs, program indentation is very costly in keystrokes. So 'se' gives you the ability to set arbitrary tab stops using the "ot" command. By default, 'se' places a stop at column 1 and every third column thereafter. Tabs corresponding to the default can be set by enumerating the column positions for the stops:

```
ot 1 4 7 10 13 16 19 22 25 28 31 34  ...
```

This is almost as bad as typing the blanks on each line. For this reason, there is also a shorthand for such repetitive specifications.

```
ot +3
```

sets a tab stop at column 1 and at every third column thereafter. Fortran programmers may prefer the specification

```
ot 7 +3
```

to set a stop at column 7 and at every third thereafter.

Once the tab stops are set, the control-i and control-e keys can be used to move the cursor from its current position forward or backward to the nearest stop, respectively.

Full-Screen Editing

Full screen editing with 'se' is accomplished through the use of control characters for editing functions. A few, such as control-h, control-i, and control-e have already been mentioned. Since 'se' supports such a large number of control functions, the mnemonic value of control character assignments has dwindled to almost zero. About the only thing mnemonic is that most symmetric functions have been assigned to opposing keys on the keyboard (e.g., forward and backward tab to control-i and control-e, forward and backward space to control-g and control-h, skip right and left to control-o and control-w, and so on). We feel pangs of conscience about this, but can find no more satisfactory alternative. If you feel the control character assignments are terrible and you can find a better way, you may change them by modifying the definitions in 'se' and recompiling.

Except for a few special purpose ones, control characters can be used anywhere, even on the command line. (This is why erroneous commands are not erased -- you may want to edit them.) Most of the functions work on a single line, but in overlay mode (controlled by the "v" command), the cursor may be positioned anywhere in the buffer.

Horizontal Cursor Motion

There are quite a few functions for moving the cursor. You've probably used at least one (control-h) to backspace over errors. None of the cursor motion functions erase characters, so you may move forward and backward over a line without destroying it. Here are several of the more frequently used cursor motion characters:

control-g	Move forward one column.
control-h	Move backward one column.
control-i	Move forward to the next tab stop.
control-e	Move backward to the previous tab stop.
control-o	Move to the first column beyond the end of the line.
control-w	Move to column 1.

Vertical Cursor Motion

'Se' provides two control keys, control-d and control-k, to move the cursor up and down, respectively, from line to line through the edit buffer. The exact function of each depends on 'se's current mode: in command mode they simply move the current line pointer without affecting the cursor position or the contents of the command line; in overlay mode (viz. the "v" command) they actually move the cursor up or down one line within the same column; finally, in append move, these keys are ignored. Regardless of the mode, the screen is adjusted when necessary to insure that the current line is displayed.

control-d	Move the cursor up one line.
control-k	Move the cursor down one line.

Character Insertion

Of course the next question is: "Now that I've moved the cursor, how do I change things?" If you want to retype a character, just position the cursor over it, and type the desired character; the old one is replaced. You may also insert characters at the current cursor position instead of merely overwriting what's already there. Typing a control-c inserts a single blank before the character under the cursor and moves the remainder of the line one column to the right; the cursor remains in the same column over the newly-inserted blank. Typing a control-x inserts enough blanks at the current cursor position to move the character that was there to the next tab stop. This can be handy for aligning items in a table, for example. As with control-c, the cursor remains in the same column.

A more general way of handling insertions is to type control-a. This toggles "insert mode" -- the word "INSERT" appears on the status line, and all characters typed from this point are inserted in the line (and characters to the right are moved over). Typing control-a again turns insert mode off. Here is a summary of these control characters:

control-a	Toggle insert mode.
control-c	Insert a blank to the left of the cursor.
control-x	Insert blanks to the next tab stop.
control-_	Insert a newline.

Character Deletion

There are many ways to do away with characters. The most drastic is to type DEL; 'se' erases the current line and leaves the cursor in column 1. Typing control-t causes 'se' to delete the character under the cursor and all those to its right. The cursor is left in the same column which is now just beyond the new end of the line. Similarly, control-y deletes all the characters to the left of the cursor (not including the one under it). The remainder of the line is moved to the left, leaving the cursor over the same character, but now in column 1. Control-r deletes the character under the cursor and closes the gap from the right, while control-u does the same thing after first moving the cursor one column to the left. These last two are most commonly used to eat characters out of the middle of a line.

DEL	Erase the entire line.
control-t	Erase the characters under and to the right of the cursor.
control-y	Erase the characters to the left of the cursor.
control-r	Erase the character under the cursor.
control-u	Erase the character immediately to left of the cursor.

Terminating a Line

After you have edited a line, there are two ways of terminating it. The most commonly used is the control-v. A newline (or carriage-return) can be used but beware that it deletes all characters over and to the right of the cursor.

control-v	Terminate.
NEWLINE	Erase characters under and to the right of the cursor and terminate.

Non-printing Characters

'Se' displays a non-printing character as a blank (or other user-selectable character; see the description of "ou" in the section on options). Non-printing characters (such as 'se's control characters), or any others for that matter, may be entered by hitting the ESC key followed immediately by the key to generate the desired character. Note, however, that the character you type is taken literally, exactly as it is generated by your terminal, so case conversion does not apply.

ESC	Accept the literal value of the next character, regardless of its function.

The .serc File

When 'se' starts up, it tries to open the file "=home=/.serc". If that file exists, 'se' reads it, one line at a time, and executes each line as a command. If a line has "#" as the <u>first</u> character on the line, or if the line is empty, the entire line is treated as a comment, otherwise it is executed. Here is a sample ".serc" file:

```
# turn on unix mode, tabs every 8 columns, auto indent
opu
ot+8
oia
```

The ".serc" file is useful for setting up personalized options, without having to type them on the command line every time, and without using a special shell file in your bin. In particular, it is useful for automatically turning on UNIX mode for Software Tools users who are familiar with the UNIX system.

Command line options are processed _after_ commands in the ".serc" file, so, in effect, command line options can be used to over-ride the defaults in your ".serc" file.

NOTE: Commands in the ".serc" file do _not_ go through that part of 'se' which processes the special control characters (see above), so _do_ _not_ use them in your ".serc" file.

Screen Editor Options

Options for 'se' can be specified in two ways: with the "o" command or on the Subsystem command line that invokes 'se'. To specify an option with the "o" command, just enter "o" followed immediately by the option letter and its parameters. To specify an option on the command line, just use "-" followed by the option letter and its parameters. With this second method, if there are imbedded spaces in the parameter list, the entire option should be enclosed in quotes. For example, to specify the "a" (absolute line number) option and tab stops at column 8 and every fourth thereafter with the "o" command, just enter

```
oa
ot 8 +4
```

when 'se' is waiting for a command. To enter the same options on the invoking command line, you might use

```
se -t regent myfile -a "-t 8 +4"
```

The following table summarizes the available 'se' options:

Option	Action
a	causes absolute line numbers to be displayed in the line number area of the screen. The default behavior is to display upper-case letters with the letter "A" corresponding to the first line in the window.
c	inverts the case of all letters you type (i.e., converts upper-case to lower-case and vice versa). This option causes commands to be recognized only in upper-case and alphabetic line numbers to be displayed and recognized only in lower-case.
d[<dir>]	selects the placement of the current line pointer following a "d" (delete) command. <dir> must be either ">" or "<". If ">" is specified, the default behavior is selected: the line following the deleted lines becomes the new current line. If "<" is specified, the line immediately preceding the deleted lines becomes the new current line. If neither is specified, the current value of <dir> is displayed in the status line.
f	selects Fortran oriented options. This is equivalent to specifying both the "c" and "t7 +3" (see below) options.
g	controls the behavior of the "s" (substitute) command when it is under the control of a "g" (global) command. By default, if a substitute inside a global command fails, 'se' will not continue with the rest of the

lines which might succeed. If "og" is given, then the global substitute will continue, and lines which failed will not be affected. Successive "og" commands will toggle this behavior. An explanatory message is placed in the status line.

h[<baud>] lets the editor know at what baud rate you are receiving characters. Baud rates can range from 50 to 19200; the default is 9600. This option allows the editor to determine how many, if any, delay characters (nulls) will be output when the hardware line insert/delete functions of the terminal are being used (if available). Use of the built-in terminal capabilities to insert/delete lines speeds up editing over slow-speed lines (i.e., dialups). Entering 'oh' without an argument will cause your current baud rate to appear on the status line.

i[a | <indent>] selects indent value for lines inserted with "a", "c" and "i" commands (initially 1). "a" selects auto-indent which sets the indent to the value which equals the indent of the previous line. If <indent> is an integer, then the indent value will be set to that number. If neither "a" nor <indent> are specified, the current value of indent is displayed.

k Indicates whether the current contents of your edit buffer has been saved or not by printing either a "saved" or "not saved" message on your status line.

l[<lop>] sets the line number display option. Under control of this option, 'se' continuously displays the value of one of three symbolic line numbers in the status line. <lop> may be any of the following:

.	display the current line number
#	display the number of the top line on the screen
$	display the number of the last line in the buffer

If <lop> is omitted, the line number display is disabled.

lm[<col>] sets the left margin to <col> which must be a positive integer. This option will shift your entire screen to the left, enabling you to see characters at the end of the line that were previously off the screen; the characters in columns 1 through <col> - 1 will not be visible. You may continue editing in the normal fashion. To reset your screen enter the command 'olm 1'. If <col> is omitted, the current left margin column is displayed in the status line.

m[d] [<user>] displays messages sent to you by other users (via the 'to' command) while you are editing. When a message arrives while you are editing, the word "message" appears on your status line. To send other users messages while inside of the editor, you can insert the text of your message into the edit buffer, and then issue the command "line1,line2om <user>", where "line1" and "line2" are the first and last lines, respectively, of where you appended your message in the edit buffer and "<user>" is the login name or process id of the person to whom you want to send a message. The given lines are sent and deleted from the edit buffer. To prevent the lines from being deleted after they are sent, use the command line "line1,line2omd <user>"

p[s | u] converts to or from UNIX (tm) compatibility mode. The "op" command, by itself, will toggle between normal (Software Tools mode) and UNIX mode. The command "opu" will force 'se' to use UNIX mode, while the command "ops" will force 'se' to use Software Tools mode.

When in UNIX mode, 'se' uses the following for its patterns and commands:

?pattern[?] searches backwards for a pattern.

^ matches the beginning of a line.

. matches any character.

^ is used to negate character classes.

% used by itself in the replacement part of a substitute command represents the replacement part of the previous substitute command.

\(<regular expression>\) tags pieces of a pattern.

\<digit> represents the text matched by the tagged subpattern specified by <digit>.

\ is the escape character, instead of @.

t copies lines.

y transliterates lines.

~ does the global exclude on markname (see the "!" command, in the help on 'ed').

![<Software Tools Command>] will create a new instance of the Software Tools shell, or execute <Software Tools Command> if it is present (see the "~" command, in the help on 'ed').

All other characters and commands are the same for both

UNIX and normal (Software Tools) mode. The help command will always call up documentation appropriate to the current mode. UNIX mode is indicated by the message "UNIX" in the status line.

UNIX mode is available _only_ in 'se'. This extension is not available in 'ed'.

s[pma | ftn | f77 | s | f] sets other options for case, tabs, etc., for one of the three programming languages listed. The option "oss" is the same as "ospma" and the option "osf" is the same thing as "osftn" (the corresponding command line options are "-ss" and "-sf"). If no argument is specified the options effected by this command revert to their default value.

t[<tabs>] sets tab stops according to <tabs>. <tabs> consists of a series of numbers indicating columns in which tab stops are to be set. If a number is preceded by a plus sign ("+"), it indicates that the number is an increment; stops are set at regular intervals separated by that many columns, beginning with the most recently specified absolute column number. If no such number precedes the first increment specification, the stops are set relative to column 1. By default, tab stops are set in every third column starting with column 1, corresponding to a <tabs> specification of "+3". If <tabs> is omitted, the current tab spacing is displayed in the status line.

u[<chr>] selects the character that 'se' displays in place of unprintable characters. <chr> may be any printable character; it is initially set to blank. If <chr> is omitted, 'se' displays the current replacement character on the status line.

v[<col>] sets the default "overlay column". This is the column at which the cursor is initially positioned by the "v" command. <Col> must be a positive integer, or a dollar sign ($) to indicate the end of the line. If <col> is omitted, the current overlay column is displayed in the status line.

w[<col>] sets the "warning threshold" to <col> which must be a positive integer. Whenever the cursor is positioned at or beyond this column, the column number is displayed in the status line and the terminal's bell is sounded. If <col> is omitted, the current warning threshold is displayed in the status line. The default warning threshold is 74, corresponding to the first column beyond the right edge of the screen on an 80 column crt.

-[<lnr>] splits the screen at the line specified by <lnr> which must be a simple line number within the current window. All lines above <lnr> remain frozen on the screen, the line specified by <lnr> is replaced by a row of dashes, and the space below this row becomes the new window on the file. Further editing commands do not affect the lines displayed in the top part of the screen. If <lnr> is omitted, the screen is restored to its full size.

Screen Editor Control Characters

(Files can be edited with control characters only when you are in overlay mode, which you can enter with the ’v’ command. A control-v will exit overlay mode and put you back into command mode. While in command mode you can use these characters to edit your command.)

<u>Character</u>	<u>Action</u>
control-a	Toggle insert mode. The status of the insertion indicator is inverted. Insert mode, when enabled, causes characters typed to be inserted at the current cursor position in the line instead of overwriting the characters that were there previously. When insert mode is in effect, "INSERT" appears in the status line.
control-b	Scan right and erase. The current line is scanned from the current cursor position to the right margin until an occurrence of the next character typed is found. When the character is found, all characters from the current cursor position up to (but not including) the scanned character are deleted and the remainder of the line is moved to the left to close the gap. The cursor is left in the same column which is now occupied by the scanned character. If the line to the right of the cursor does not contain the character being sought, the terminal’s bell is sounded. ’Se’ remembers the last character that was scanned using this or any of the other scanning keys; if control-b is hit twice in a row, this remembered character is used instead of a literal control-b.
control-c	Insert blank. The characters at and to the right of the current cursor position are moved to the right one column and a blank is inserted to fill the gap.
control-d	Cursor up. The effect of this key depends on ’se’s current mode. When in command mode, the current line pointer is moved to the previous line without affecting the contents of the command line. If the current line pointer is at line 1, the last line in the file becomes the new current line. In overlay mode (viz. the "v" command), the cursor is moved up one line while remaining in the same column. In append mode, this key is ignored.
control-e	Tab left. The cursor is moved to the nearest tab stop to the left of its current position.
control-f	"Funny" return. The effect of this key depends on the editor’s current mode. In command mode, the current command line is entered as-is, but is not erased upon completion of the command; in append mode, the current

line is duplicated; in overlay mode (viz. the "v" command), the current line is restored to its original state and command mode is reentered (except if under control of a global prefix).

control-g Cursor right. The cursor is moved one column to the right.

control-h Cursor left. The cursor is moved one column to the left. Note that this <u>does</u> <u>not</u> erase any characters; it simply moves the cursor.

control-i Tab right. The cursor is moved to the next tab stop to the right of its current position.

control-k Cursor down. As with the control-d key, this key's effect depends on the current editing mode. In command mode, the current line pointer is moved to the next line without changing the contents of the command line. If the current line pointer is at the last line in the file, line 1 becomes the new current line. In overlay mode (viz. the "v" command), the cursor is moved down one line while remaining in the same column. In append mode, control-k has no effect.

control-l Scan left. The cursor is positioned according to the character typed immediately after the control-l. In effect, the current line is scanned, starting from the current cursor position and moving left, for the first occurrence of this character. If none is found before the beginning of the line is reached, the scan resumes with the last character in the line. If the line does not contain the character being looked for, the message "NOT FOUND" is printed in the status line. 'Se' remembers the last character that was scanned for using this key; if the control-l is hit twice in a row, this remembered character is searched for instead of a literal control-l. Apart from this, however, the character typed after control-l is taken literally, so 'se's case conversion feature does not apply.

control-m Newline. This key is identical to the NEWLINE key described below.

control-n Scan left and erase. The current line is scanned from the current cursor position to the left margin until an occurrence of the next character typed is found. Then that character and all characters to its right up to (but not including) the character under the cursor are erased. The remainder of the line, as well as the cursor are moved to the left to close the gap. If the line to the left of the cursor does not contain the character being sought, the terminal's bell is sounded. As with the control-b key, if control-n is hit twice in a row, the last character scanned for is used instead of a literal control-n.

control-o	Skip right. The cursor is moved to the first position beyond the current end of line.
control-p	Interrupt. If executing any command except "a", "c", "i" or "v", 'se' aborts the command and reenters command mode. The command line is not erased.
control-q	Fix screen. The screen is reconstructed from 'se's internal representation of the screen.
control-r	Erase right. The character at the current cursor position is erased and all characters to its right are moved left one position.
control-s	Scan right. This key is identical to the control-l key described above, except that the scan proceeds to the right from the current cursor position.
control-t	Kill right. The character at the current cursor position and all those to its right are erased.
control-u	Erase left. The character to the left of the current cursor position is deleted and all characters to its right are moved to the left to fill the gap. The cursor is also moved left one column, leaving it over the same character.
control-v	Skip right and terminate. The cursor is moved to the current end of line and the line is terminated.
control-w	Skip left. The cursor is positioned at column 1.
control-x	Insert tab. The character under the cursor is moved right to the next tab stop; the gap is filled with blanks. The cursor is not moved.
control-y	Kill left. All characters to the left of the cursor are erased; those at and to the right of the cursor are moved to the left to fill the void. The cursor is left in column 1.
control-z	Toggle case conversion mode. The status of the case conversion indicator is inverted; if case inversion was on, it is turned off, and vice versa. Case inversion, when in effect, causes all upper case letters to be converted to lower case, and all lower case letters to be converted to upper case. Note, however, that 'se' continues to recognize alphabetic line numbers in upper case only, in contrast to the "case inversion" option (see the description of options above). When case inversion is on, "CASE" appears in the status line.
control-_	Insert newline. A newline character is inserted before the current cursor position, and the cursor is moved one position to the right. The newline is displayed according to the current non-printing replacement

character (see the "u" option).

control-\	Tab left and erase. Characters are erased starting with the character at the nearest tab stop to the left of the cursor up to but not including the character under the cursor. The rest of the line, including the cursor, is moved to the left to close the gap.
control-^	Tab right and erase. Characters are erased starting with the character under the cursor up to but not including the character at the nearest tab stop to the right of the cursor. The rest of the line is then shifted to the left to close the gap.
NEWLINE	Kill right and terminate. The characters at and to the right of the current cursor position are deleted, and the line is terminated.
DEL	Kill all. The entire line is erased, along with any error message that appears in the status line.
ESC	Escape. The ESC key provides a means for entering 'se's control characters literally as text into the file. In fact, any character that can be generated from the keyboard is taken literally when it immediately follows the ESC key. If the character is non-printing (as are all of 'se's control characters), it appears on the screen as the current non-printing replacement character (normally a blank).

Editor Command Summary

Range	Syntax	Function
.	a[:text]	Append Inserts text after the specified line. Text is inserted until a line containing only a period and a newline is encountered. In 'se', if the command is followed immediately by a colon, then whatever text follows the colon is inserted without entering "append" mode. The current line pointer is left at the last line inserted.
.,.	c[:text]	Change Deletes the lines specified and inserts text to replace them. Text is inserted until a line containing only a period and a newline is encountered. In 'se', if the command is followed immediately by a colon, then whatever text follows the colon is inserted without entering "append" mode. The current line pointer is left at the last line inserted.
.,.	d[p]	Delete Deletes all lines between the specified lines, inclusive. The current line pointer is left at the line after the last one deleted. If the "p" is included, the new current line is printed.
none	e[!] [filename]	Enter Loads the specified file into the buffer and prepares for editing. Automatically invoked if a filename is specified as an argument on the command line used to invoke the editor. The current line pointer is positioned at the first line in the buffer. An error message is generated if the editing buffer contains text that has not been saved. The enter command may be resubmitted after the error message, in which case it will be obeyed. The "enter now" command "e!" may be used to avoid the error message.
none	f [filename]	File Print or change the remembered file name. If a name is given, the remembered file name is set to that value; otherwise, the remembered file name is printed.

.,$	g/pat/command	**Global on pattern** Performs the given command on all lines in the specified range that match a certain pattern.
none	h[stuff]	**Help** In 'se', provides access to online documentation on the screen editor. "Stuff" may be used to select which information is displayed.
.	i[:text]	**Insert** Inserts text before the specified line. Text is inserted until a line containing only a period and a newline is encountered. In 'se', if the command is immediately followed by a colon, then whatever text follows is inserted without entering "append" mode. The current line pointer is left at the last line inserted.
^,.	j[/stuff[/]][p]	**Join** The specified lines are joined into a single line. You may specify in "stuff" what is to replace the newlines that previously separated the lines. The default is a single blank. If you use the default, 'ed' automatically prints out the result. If the "p" option is used, the resulting line (which becomes the new current line) is printed. Thus "j" and "jp" are equivalent to "j/ /p". In general, 'ed' and 'se' will supply trailing delimiters for you. So "j/" is the same as "j//", i.e. replace the newline(s) with nothing (delete them).
.,.	km	**marK** The specified lines are marked with 'm' which may be any single character other than a newline. If 'm' is not present, the lines are marked with the default name of blank. The current line pointer is never changed.
none	l	**Locate** "l" will print the first line of the file =installation=. This is so that one can tell what machine he is using from within the editor. This is particularly useful for installations with many machines that can run the editor, where the user can switch back and forth between them, and become confused as to where he is at a given moment.

.,.	m<line>[p]	Move Moves the specified block of lines after <line>. <Line> may not be omitted. The current line pointer is left at the last line moved. If the "p" is specified, the new current line is also printed.
.,.	n[m]	Name If 'm' is present, the last line in the specified range is marked with it and all other lines having that mark name are given the default mark name of blank. In 'ed', if 'm' is not present, the mark name of each line in the range is printed; in 'se' the names of all lines in the range are cleared.
none	o[stuff]	Option Editing options may be queried or set. "Stuff" determines which options are affected. In 'ed', options "d", "g", "k", and "p" are available. Options "d", "g", and "k" are the same as in 'se'. In 'ed', option "p" sets the prompt to be used (useful for the user who is disturbed by 'ed's quiet behavior). The prompt can be set by the command "op/string[/]", which sets the prompt to "string". The trailing delimiter is optional. If no string is given, the prompt is set to "* ". An empty string ("op//") restores 'ed's no prompting behavior. Successive "op" commands will toggle prompting mode. In 'se', the "op" command controls what metacharacters are used for pattern matching.
.,.	p	Print Prints all the lines in the given range. In 'se', as much as possible of the range is displayed, always including the last line; if no range is given, the previous page is displayed. The current line pointer is left at the last line printed.
none	q[!]	Quit Exit from the editor. An error message is generated if the editing buffer contains text that has not been saved. The quit command may be resubmitted after the error message, in which case it will be obeyed. The "quit now" command "q!" may be used to avoid the error message.

.   r [filename]   Read

Insert the contents of the given file after the specified line. The current line pointer is left at the last line read.

.,.   s[/pat/sub[/][g][p]] Substitute

Substitutes "sub" for each occurrence of the pattern "pat". If the optional "g" is specified, all occurrences in each line are changed; otherwise, only the first occurrence is changed. The current line pointer is left at the last line in the range in which a substitution was made. This line is also printed if the "p" is used. In 'ed', if you leave off the trailing slash, the result of the substitute will be printed automatically. Thus "s/junk/stuff" is entirely equivalent to "s/junk/stuff/p". If you type an "s" by itself, without a pattern and replacement string, 'ed' will behave as though you had typed "s//&/p", i.e. substitute the previous replacement pattern for the previous search pattern, and print.

.,.   t[/from/to[/][p]] Transliterate

The range of characters specified by 'from' is transliterated into the range of characters specified by 'to'. The last line on which something was transliterated is printed if the "p" option is used. The last line in the range becomes the new current line. Again, if you leave off the trailing delimiter, 'ed' will print the result of the transliteration. In addition, like the "s" command, both the 'from' and 'to' parts are saved; "t//&/" will perform the same transliteration as the last one, and "t" is the same as "t//&/". The "&" is special if it is the only character in the 'to' part, otherwise it is treated as a literal "&". In Unix mode (for 'se' only), use "%" instead of "&". See <u>Software</u> <u>Tools</u> and the help on 'tlit' for some examples of character transliterations.

.   u[d][p]   Undo

The specified range of lines is replaced by the last range of lines deleted. If the "d" is used, the restored text is inserted after the last line in the specified range. The current line

		pointer is set at the last line that was restored; this line is also printed if the "p" is specified.
.,.	v	oVerlay In 'ed', each line in the given range is printed without its terminating newline and a line of input is read and added to the end of the line. If the first and only character on the input line is a period, no further lines are printed. In 'se', "overlay mode" is entered and the control characters may be used to modify text anywhere in the buffer. A control-v may be used to quit overlay mode. A control-f may be used to restore the current line to its original state and terminate the command.
1,$	w['+'|'!'] [filename]	Write Writes the portion of the buffer specified to the named file. The current line pointer is not changed. If "+" is given, the portion of the buffer is appended to the file; otherwise the portion of the buffer replaces the file. In 'se' only, if "!" is present, an existing file specified in the command is overwritten without comment. If "filename" is not present, the specified lines will be written to the current file name specified on the status line.
1,$	x/pat/command	eXclude on pattern Performs the command on all lines in the given range that do not match the specified pattern.
.,.	y<line>[p]	copY Makes a copy of all the lines in the given range, and inserts the copies after <line>. As with the "m" command, <line> may not be omitted. The current line pointer is set to the new copy of the last line in the range; this line is printed if the "p" is present.
.,.	zb<left>[,<right>][<char>]	draw Box In 'se' only, a box is drawn using the given <char> (blank by default, allowing erasure of a previously-drawn box). Line numbers are used to specify top and bottom row positions of the box. <Left> and <right> specify left and right column positions of the box. If second line number is omitted, the box degenerates to

		a horizontal line. If right-hand column is omitted, the box degenerates to a vertical line.
.	=[p]	Equals The number of the specified line is printed. The line itself is also printed if the "p" option is used. The current line pointer is not changed.
none	?	Query In 'ed' only, a verbose description of the last error encountered is printed.
1,$	!mcommand	Exclude on markname Similar to the 'x' prefix except that 'command' is performed for all lines in the range that do not have the mark name 'm'.
1,$	'mcommand	Global on markname Similar to the 'g' prefix except that 'command' is performed for all lines in the range that have the mark name 'm'.
.	:	Print next page In 'ed', 23 lines beginning with the current line are printed (equivalent to ".,.+23p"). In 'se', the next page of the buffer is displayed and the current line pointer is placed at the top of the window.
none	~[<Software Tools Command>]	Escape to the shell If present, the <Software Tools Command> is passed to the shell to be executed. Otherwise, an interactive shell is created. After either the command or the shell exits, 'ed' prints "~" to indicate that the shell escape has completed. For a command, 'se' asks you to type a newline before redrawing the screen, but for an interactive shell, 'se' will redraw the screen immediately. If the first character of the <Software Tools Command> is a "!", then the "!" is replaced with the text of the previous shell command. An unescaped "%" in the <Software Tools Command> will be replaced with the current saved file name. If the shell command is expanded, both 'ed' and 'se' will echo it first, and then execute it. Until EPFs are supported, when using 'ed', do not use the shell to execute

external commands. Internal commands (like 'cd') are OK. This does not apply to 'se'.

For a deeper discussion of using the shell from within a program, see the help on the 'shell' subroutine.

Elements of Line Number Expressions

Form	Value
integer	value of the integer (e.g., 44).
.	number of the current line in the buffer.
$	number of the last line in the buffer.
^	number of the previous line in the buffer (same as .-1).
-	number of the previous line in the buffer (same as ^).
#	number of the first line on the screen (only in 'se')
/pattern[/]	number of the next line in the buffer that matches the given pattern (e.g., /February/); the search proceeds to the end of the buffer, then wraps around to the beginning and back to the current line. The trailing "/" is optional.
\pattern[\]	number of the previous line in the buffer that matches the given pattern (e.g., \January\); search proceeds in reverse, from the current line to line 1, then from the last line back to the current line. The trailing "\" is optional.
>name	number of the next line having the given markname (search wraps around, like //).
<name	number of the previous line having the given mark-name (search proceeds in reverse, like \\).
expression	any of the above operands may be combined with plus or minus signs to produce a line number expression. Plus signs may be omitted if desired (e.g., /parse/-5, /lexical/+2, /lexical/2, $-5, .+6, .6).

Summary of Pattern Elements

Element	Meaning
%	Matches the null string at the beginning of a line. However, if not the first element of a pattern, is treated as a literal percent sign.
?	Matches any single character other than newline.
$	Matches the newline character at the end of a line. However, if not the last element of a pattern, is treated as a literal dollar sign.
[<ccl>]	Matches any single character that is a member of the set specified by <ccl>. <Ccl> may be composed of single characters or of character ranges of the form <c1>-<c2>. If character ranges are used, <c1> and <c2> must both belong to the digits, the upper case alphabet or the lower case alphabet.
[~<ccl>]	Matches any single character that is not a member of the set specified by <ccl>.
*	In combination with the immediately preceding pattern element, matches zero or more characters that are matched by that element.
@	Turns off the special meaning of the immediately following character. If that character has no special meaning, this is treated as a literal "@".
{<pattern>}	Tags the text actually matched by the sub-pattern specified by <pattern> for use in the replacement part of a substitute command.
&	Appearing in the replacement part of a substitute command, represents the text actually matched by the pattern part of the command. If "&" is the only character in the replacement part, however, then it represents the replacement part used in a previous substitute command.
@<digit>	Appearing in the replacement part of a substitute command, represents the text actually matched by the tagged sub-pattern specified by <digit>.