SCIENCE AND ENGINEERING RESEARCH COUNCIL
RUTHERFORD APPLETON LABORATORY
COMPUTING DIVISION

Starlink Project
Starlink User Note 24.1

Modified 02-FEB-81

STE - The Software Tools Editor

Software Tools is an excellent book written by B. W. Kernighan and P. J. Plauger, published by Addison-Wesley. In it the Authors discuss how to write programs that make good tools, and how to program well in the process. One of the tools they develop is a fairly powerful editor, written in Ratfor (a structured form of FORTRAN IV). This program has been implemeted on the UCL Starlink VAX (with a few modifications and extensions) and is recommended as the editor to use on the VAX.

This note gives a brief introduction to, and description of, the editor which has been abstracted from the book (which you are recommended to buy). There are some short command summary sections at the end of this note, which are also kept in files in the directory [USERDOC.LOCAL.SWT]. After reading this note you may like to print these short files and use them for reference when using the editor.

Introduction

STE is modeled closely after the latest in a long family of conversational text editors that have achieved wide acceptance. Concern for human engineering dominates the design - STE tries to be concise, regular and powerful. Because STE is primarily intended for interactive use, it is streamlined and terse, but easy to use. This is especially important for a text editor: for most users it is the primary interface to the system. STE is not confined to conversational editing, however. It can be driven from prepared scripts and from other programs.

An editor, like a programming language, is so heavily used that it should be really good, so that you don't spend all your time fighting its deficiences. Accordingly, STE provides a relatively rich set of facilities, much more than the bare minimum.

This section contains a synopsis of STE, enough to give you a feeling for what commands are available. Individual commands will be described in detail later.

To get started, you type

```
        STE
```

or

```
        STE file
```

where file is a normal VAX file name.
In the latter case, if the file already exists, it's assumed that you want to access its contents, so they are copied into an internal buffer. In any case, text is modified in the buffer and perhaps eventually written back to some external file. Files are never modified except by explicit command. This proves to be a safer procedure than working on a file in place, for if you botch an edit you can always read in a fresh copy and start anew.

STE is basically "line oriented" in that most editing commands operate on groups of one or more lines in the buffer. This is a natural organisation, since text intrinsically comes in lines. Other units might be selected - characters, words, sentences, or arbitrary strings - but lines seem to be most suitable for a wide variety of applications.
It is certainly possible to access parts of lines as well; we'll get to that in a moment.

STE tries to be concise and regular. ALL editing commands consist of a single letter, which may be optionally preceded by one or two "line numbers" which specify the inclusive range of lines in the buffer over which the command is to act. Thus the command

```
      1p
```

calls for the printing of the first line, and

```
      1,3p
```

prints lines 1,2, and 3.

The delete command d is analogous to p ; ir deletes the lines in the specified range:

```
      1,3d
```

deletes the first three lines from the buffer. It is always an error to refer to a line that doesn't exist; STE complains when you do.

Line numbers are relative to the beginning of the buffer. After the first three lines have been deleted, the first remaining line (the old line 4) becomes the new line 1, and all the other lines as renumbered correspondingly. This behaviour may be unfamiliar if you're used to an editor where "line numbers" have a physical existance as part of the text lines themselves. STE line numbers are not part of the file, and indeed have no physical representation anywhere; they are just the relative positions of lines in the buffer. As you will see shortly, this organisation gives invaluable flexibility in specifying and rearranging lines.

Although it is possible to edit entirely in terms of line numbers, be they relative or absolute, it's often an unwieldy nuisance, so STE lets you specify the lines in which you're interested in several other ways. For instance, the editor always keeps track of the CURRENT LINE, typically the most recent line affected by the previous command. The current line is specified by the character . (period or "dot"), which you can use anywhere you would have used an integer line number. The LAST LINE in the buffer is also known; it is called $. So

```
      .,$p
```

would print the current line and any subsequent lines through to the end of the buffer;

```
      1,$p
```

prints everything; and

```
      1,$d
```

deletes everything.

Dot is altered by many commands. In particular, it is set to the last line pointed after a p command and to the next undeleted line after d, except that it never moves past $. Thus a single

```
      d
```

deletes the current line, and leaves dot pointing to the next line, while

```
      .,$d
      p
```

deletes all lines from here to the end, and prints the new last line.

The purpose of . and $ is to reduce the need for specific line numbers. This is further helped by the ability to do line number arithmetic. To print the last few lines of the buffer (perhaps to see how far you got in a previous editing session), say

```
      $-10,$p
```

Or you can say

```
      .-5,.+5p
```

to print a group of lines around where you're working.

Even when augmented by . , $ and arithmetic, line number editing is still clumsy. When you're editing, you want to be able to say, "Find me an occurrence of this string", so that you can work on it without having to know procisely where it occurs. In STE you can do a CONTEXT SEARCH to find a line, simply by writing a pattern between slashes.

```
      /abc/
```

means: Starting with the next line after the current line, scan forward until you find a line which matches the pattern abc . The details of the patterns that can be specified will be dealt with later. The search wraps around from line $ to line 1 if necessary. Thus

```
      /abc/,$p
```

would locate the next line (after the current line) that matches abc , and print from there to the end of the buffer. (If a context search proceeds forward around the ring back to the current line without finding a match, an error is signaled.)

Similarly you can scan backwards by writing a pattern between backslashes. \def\ means: starting with the line right before the current line, scan backward until you find a line which matches the specified pattern (def). Again, the search wraps around from line 1 to line $, and if no line satisfies the search, an error is signalled. Editing a Fortran program, for instance, you might say

```
\subroutine\,/end/p
```

to print the subroutine in which the current line is imbedded.

A lne number standing by itself (i.e. followed only by carriage return) is taken as a request to print that line, so

```
$
```

prints the last line, and the common case of "Find the next line with an abc in it" is

```
/abc/
```

It finds the line, prints it, and sets dot to that line so you can begin work there. As special cases, a carriage return all by itself is a request to print the next line.

It is hard to overstate the importance of context searching. Most of the time you use context searches to get to the next place where you want to do some editing. Even when you know the source line numbers, it's often better to scan. If you've used a identifier in two different ways, for instance, you might overlook an instance or two while correcting the listing. A context search, however, will lead you in turn to every place in the source where the offending identifier is referenced.

Placing line numbers before the command instead of after may seem unnatural at first, but one adapts rapidly. Thus choice lets individual commands use different syntaxes for optional information after the command letter without destroying the regularity with which a range of lines is specified.

The most important of the commands which take further information is the substitute command s , used for changing characters within a line.

```
s/ofrmat/format/
```

changes the FIRST occurence of ofrmat to format on the current line. If there should be more than one on that line, you can say

```
s/ofrmat/format/g
```

to do it GLOBALLY (i.e. everywhere) on the line.

An s command can be preceded by one or two line numbers, to indicate that the substitution is to be done on a range of lines:

```
.+1s/ofrmat/format/
```

fixes the mistake on the next line, and

```
1,$s/ofrmat/format/g
```

does it everywhere on all lines. (This is handy for consistent misspellers.) Dot is left pointing to the last line which was changed.

s is probably the most useful command in the editor, since it permits you to specify changes in a line or lines succinctly.

The character that delimits the pieces of a substitute command need not be slash; any character (except : and carriage return) will do, so

```
      s?/??g
```

deletes all slashes in a line. The last pattern to be used in a context search or substitute is remembered, and can be specified by a null pattern like // or \\. If you say /format/ to find a format statement, and its not the one you want, you can say // to get to the next one, or \\ to get back to the previous one. The remembered pattern eliminates a lot of tedious and error-prone typing. A typical use of remembered patterns is

```
      /ofrmat/s//format/
      //s//format/
      //s//format/
```

etc.

to walk (slowly) through a program, picking up the misspellings one by one. You know you have them all when a search fails. You can change them all at once with

```
      1,$s/ofrmat/format/g
```

The original STE editor is very quiet, i.e. it does not give prompts, or print lines after they have been corrected etc. . To get it to print corrections you can add a p to the end of most commands, which will force a print of the last line corrected. Some people like this style of working whereas others prefer prompting and printing of lines after correction. To satisfy both groups the VAX implementation of STE has a switch to turn on prompts and printing. When entered with the command :

```
          STE         or       STE file
```

STE is in 'verbose' mode, it prompts with > when it is waiting for a command, and always prints lines after correction. If this behaviour is not required you can invoke STE with :

```
       STE  -p  file
```

which will suppress prompts, and printing lines after correction. You can add any text (ended with a blank) after the -p to specify your own promp

```
 e.g.       STE -pthisisaverylongprompt     file
```

will prompt with THISISAVERYLONGPROMPT (NOT recommended !!)

Some people like to have line numbers against the text when working with an editor, this can be achieved in STE by invoking it with :

```
       STE -n  file
```

NOTE : these line numbers are relative line numbers !

Returning now to operations that affect whole lines of text. Most important is the APPEND command a , to add new lines a text to the buffer. It is the basic mechanism for adding text to a file, or for making a file to begin with.

Since it is used so much, the append mechanism tries to be as unobtrusive as possible. Once it encounters the command a , STE enters a special append mode where everything following is tucked away in the appropriate part of the buffer. Escape sequences and other special characters (described later) lose their special meaning, until a line is encountered that contains only a period at the beginning. This signal, which is easy to type and pretty unlikely to appear in ordinary text, marks the end of the append mode and is not itself copied into the buffer. Subsequent lines are interpreted as commands once again.

So to add text to the buffer, you specify where you want to put it and do an append. To tack stuff on to the end, for instance,

```
$a
anything you want to type
except a line containing only a .
as in the following line
.
```

This adds three lines to the buffer, then resumes looking for commands. Note the period character is only magic only when it stands alone at the beginning of a line. To add something at the beginning of the buffer, you can use the line number zero, as in 0a . If no line number is specified, the text is appended after the current line (dot).

The INSERT command, i , is identical to a , except that it inserts lines before the line named, instead of after it. The CHANGE command, c , replaces one or more lines with a fresh group of zero or more lines :

```
line1,line2 c
stuff
.
```

replaces line1 through line2 with whatéver lines follow the c. If no line numbers are given, dot is used by both i and c.

Clearly if you have a and d, you don't need either c or i. The extra flexibility, however, appears to be worthwhile.

Dot is left at the last line of text appended, changed or inserted, so you can correct errors as you go, as in this sequence:

```
a                               append some text
10   ofrmat(...)                        oops!
.                               stop appending
s/ofr/for/                      fix it
a                               resume appending
...more text..
botched line                            oops again
.                               stop appending
c                               just replace ir entirely
... corrected stuff ...                 and continue typing
```

The behaviour of dot and the default line numbers may seem like a minor concern, but in fact proper choices are crucial for smooth editing. The example above works naturally, without any explicit line numbers, because dot and the default line numbers are "right" each time.

The BROWSE command, b , is useful as it allows you to print a block of lines. The format of the command is :

```
b[.|+|-] [n]
```

```
e.g.    b 15    or    b-      or    b. 30        etc.
```

A b on its own will print a block of lines starting at the current line. After printing, the current line is the last line printed. The number of lines can be set by specifying a value for n , after it has been set the value is used until it is reset. The initial value for n is 23 .
The command b. will print out the block of lines centered about the current line, and b- will print out the block with the last line as the current line (useful for going backwards through the file).

The MOVE command, m , lets you move a block of one or more lines to any place in the buffer, and thus provides for "cut and paste" editing. The command

```
line1,line2mline3
```

moves line1 through line2 inclusive to after line3. Thus

```
.,.+1m$
```

moves the current line and the one following to the end of the buffer and

```
$m0
```

moves the last line to the beginning ("after line 0"). If no line1 or line2 is present, line dot is moved. Dot is left pointing to the last line moved.

The COPY command, k , is identical to m except that the specified lines are copied, rather than moved.

You can add the contents of any file to the buffer with the READ command r :

```
r file
```

reads file, places its contents right after line dot, and sets dot to the last line read in. Lines already in the buffer are not altered. If a line is specified with the r command, the text is read in after that line.

Any part of the buffer can be written onto any file with the WRITE command, w ;

```
\subroutine\,/end/w test.for
```

writes the current subroutine to the file test.for. If no lines are given, a w command writes out the entire contents of the buffer, and if you leave out the file name (a bare w command), it writes a new version of the file name used in the original STE file command. w does not change dot, nor does it alter the buffer.

The file name used in the original STE command is remembered and can be printed out using the f command. The name of the "current" file can be changed by giving the command : f file . This file then becomes the "current file" which will be written to by the bare w command. Thus the following sequences have the same effect

```
     STE fred.dat                              STE fred.dat
     .                     .
     editing commands                          editing commands
     .                     .
     f fred1.dat
     w                     w fred1.dat
```

The QUIT command, q , lets you leave the editor. The file you were working on IS NOT SAVED AUTOMATICALLY - if you want it saved, you have to issue an appropriate w command before the q.

When you have finished editing a file and have written it back to disk with the w command, you may wish to edit another file. To save you quitting and giving another STE command, you can use the e command which flushes the contents of the buffer and reads in the specified file :

```
     w                     w
     q
   $ STE fred2.dat                             e fred2.dat
     .                     .
```

Any editor command except a,c,i, and q can be preceeded by a GLOBAL PREFIX :

```
     g/pattern/ command
```

specifies that command is to be performed for each line in the buffer that contains an instance of pattern . g, like w, has a default range of all lines, but a smaller range can be given. A common use of the global prefix is to print all lines containing an interesting pattern:

```
     g/interesting/p
```

or to delete all lines with an undesirable pattern:

```
     g/undesirable/d
```

You could use

```
     g/ofrmat/s//format/gp
```

to find all ofrmat's, fix them, and print each corrected line as a check. Since the command that follows a global prefix can have a range of lines, we can print all lines near ones that contain an interesting pattern:

```
     g/interesting/.-1,.+1p
```

The g prefix is defintely an advanced feature, not the concern of a first time user, but it is worth learning.

There is also a x command which is identical to g , except that it operates only on those lines that DO NOT contain the pattern (x is for "exclude"):

A semicolon may be used to separate line numbers just as a comma does, but it has the additional effect of setting dot to the latest line number before evaluating the next argument.

```
      /abc/;.+1p
```

scans forward to the next line containing abc , then prints that line and the one following it (.+1) .

A line number expression may be arbitrarily complex, so long as its value lies between 0 and $, inclusive. And there can be any number of expresions, so long as the last one or two are legal for th particular command. Thus

```
      \function\;\\
```

finds the second previous function declaration, and

```
      /C/;//;//;//p
```

prints from the third succeeding line containing C to the forth, inclusive.

You can do a lot of editing without global prefixes, semicolons, and multiple context searches, and indeed there is a seldom call for anything as elaborate as the last example. But as you gain familiarity with the editor, more and more of these things become natural. And when you write scripts to perform complex editing sequences on a series of files, these facilities are invaluable.

A few other commands are :

h	prints a brief page of help information
j	joins two lines together into one
u	undoes the last delete command
l	can be used wherever p can be used, it prints all characters (including the ascii control characters which are printed as, e.g. ^G

Text Patterns

You will have noticed that we have talked about searching for, and substituting for patterns in the discussion of the s and g commands. We now describe what we mean by a "pattern" .

A text pattern can be a simple thing, like the letter a, or a more elaborate construct built up from simple things, like the string format . To build arbitrary text patterns, you need only remember a handful of rules.

Any literal character, like a, is a text pattern that matches that same character in the text being scanned. A sequence of literal characters, like 123 or format, is a pattern that matches any occurance of that sequence of characters in a line of the input.

A pattern is said to match part of a text line if the text line contains an occurence of the pattern. For example the pattern aa matches the line aabc once at position 1, the line aabcaabc in two places and the line aaaaaa in five (overlapping) places. Matching is done on a line-by-line basis; no pattern can match across a line boundary. Text patterns may by concatenated: a text pattern followed by another text pattern forms a new pattern that matches anything matched by the first, followed immediately by anything matched by the second. A sequence of literal characters is an example of concatenated patterns.

Although it is an easy task to look only for literal strings, you will soon find it restrictive. Thus we need the ability to search for patterns that match classes of characters, that match patterns only at particular positions on a line, or that match text of indefinite length.

To be able to express these more general patterns, we need to preempt some characters to represent other types of text patterns, or to delimit them. For example, we will use the character ? as a text pattern that matches any single character except a carriage return. The pattern x?y matches x+y, xay, x?y and similar strings.

The ? and other reserved characters are often called metacharacters. We try to choose characters which will not appear with high frequency in normal text, but there will be occasions where we want to look for a literal occurence of a metacharacter. Thus the special meaning of any metcharacter can be turned off by preceding it with the escape character @. Thus @? matches a literal ? , and @@ matches a literal at-sign.

The metacharacter [signals that the characters following, up to the next], form a character class, that is, a text pattern that matches any single character from the bracketed list. Thus [Aa] matches A or a , [a-z] matches any lower case letter. You can also specify a negated character class, that is, the pattern matches any character not in the specified list. Thus [!a] matches any character except a etc. . The escape convention must be used inside character classes if the class is to include ! or - or @ or] .

Two other metacharacters do not match literal characters, but rather match positions on the input line. % matches the beginning of a line: %abc matches abc only if it occurs as the first three characters on the line. Analogously, $ matches the carriage return at the end of the line: abc$ matches abc only if abc is the last thing on a line. Of course these can work together: %abc$ matches a line that contains only abc; and %$ matches only empty lines (i.e. just A carriage return). A useful command in the editor is :

```
    s/$/some more text/
```

which adds the text on to the end of a line.

Any of the above text patterns that match a single character (everthing but % and $) can be followed by the character * to make a text pattern which matches zero or more successive occurences of the single character pattern. The resulting pattern is called a closure. For example, a* matches zero or more a's; aa* matches one or more a's [a-z]* matches any string of zero or more lower case letters.

A plus sign can also be used to indicate one or more repetitions, this is known as an anchored closure.

Since a closure matches zero or more instances of the pattern, which do we pick if there is a choice? It turns out to be most convenient if the longest possible string is matched, even when a null-string match would be equally valid. Thus [a-zA-Z]* matches an entire word (which may be a null string), [a-zA-Z][a-zA-Z]* matches an entire word (one or more letters, but not a null string), and ?* matches a whole line (which may be a null string). Any ambiguity in deciding which part of a line matches a pattern will be resolved by choosing the match beginning with the leftmost character, then choosing the longest possible match at that point. So [A-Z][A-Z0-9]* matches the leftmost Fortran identifier on a line, (?*) matches anything between parentheses, and ??* matches an entire line of one or more characters (but not a line containing only carriage return).

STE command summary - patterns

A text pattern consists of the following elements :

c	literal character
?	any character except return
%	beginning of line
$	end of line
[..]	character class (any one of these characters)
[!..]	negated character class (all but these characters)
*	closure (zero or more occurences of previous pattern)
+	anchored closure (one or more occurences of the patte
@c	escaped character (e.g. @%, @[, @*)

any special meaning of characters in a text pattern is lost when escaped, inside [..], or for :

%	not at beginning
$	not at end
*	at beginning

A character class consists of zero or more of the following elements, surrounded by [and] :

c	literal character, including [
a-c	range of characters (digits, lower or upper case)
!	negated character class if at beginning
@c	escaped character (@!, @-, @@, @])

Special meaning of characters in a character class is lost when escaped or for

!	not a beginning
-	at beginning ot end

A substitution pattern consists of xero or more of the following elements:

c	literal character
&	ditto, i.e. whatever was matched
@c	escaped character (@&)

An escape sequence consists of the character @ followed by a single char. :

@n	newline
@t	tab
@c	c (including @@)

STE command summary - line numbers

Line numbers are formed of the following components :

17	a decimal number
.	the current line
$	the last line
/pattern/	a forward context search
\pattern\	a backward context search

Components may be combined with + or - :

.+1	sum of . and 1
.-34	difference of dot and 34

Line numbers are separated by commas or colons; a colon sets the current line to the most recent line number before proceeding.

STE command summary - the commands

In alphabetical order, the commands and default line numbers are :

(.)	a	append text after line (text follows)
(.)	b[.|+|-] [n]	print n lines of text.
(.,.)	c	change text (text follows)
(.,.)	dp	delete text
	e file	enter file, discard all previous text
	f file	print file name, remember file name
	h	print help information
(.)	i	insert text before line (text follows)
(.,.+1)	j	joins two lines together
(.,.)	kline3p	kopy text to after line3
(.,.)	l	print text (including control characters)
(.,.)	mline3p	move text to after line3
(.,.)	p	print text
	q	quit
(.)	r file	read file, appending after line
(.,.)	s/pat/new/gp	substitute new for first occurence of pat (g implies repeatedly across line)
	u	undoes the last delete command
(1,$)	w file	write file (leaves current state unaltered)
(.)	=p	print line number
(.+1)	newline	print next line

The trailing p, which is optional, causes the last affected line to be printed.