A FULL SCREEN EDITOR IMPLEMENTED IN FASCAL

by

THEODORE JOHN SOCLOPSKY

B.S., KANSAS STATE UNIVERSITY, 1980

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1981

Approved by:

Mycro A. Alhum
Major Professor
ACKNOWLEDGEMENTS

I would like to thank Professor Ed Basham for revising the introduction, and Nassrin Tavakoli for paraphrasing parts of chapter 2 of "Pascal Microengine Computer Pascal Operator's Manual" (release A0 December, 1979) for sections of chapter 2 in this document. I thank Kim Janne for his valuable hints.
### TABLE OF CONTENTS

**INTERNAL IMPLEMENTATION**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Modes of SEDIT</td>
<td>1</td>
</tr>
<tr>
<td>3. Organization of SEDIT</td>
<td>2</td>
</tr>
<tr>
<td>3.1 Command Execution Module</td>
<td>3</td>
</tr>
<tr>
<td>3.2 Added Command Definition Module</td>
<td>3</td>
</tr>
<tr>
<td>3.3 Macro Execution Module</td>
<td>3</td>
</tr>
<tr>
<td>3.4 Console And Keyboard Translators</td>
<td>4</td>
</tr>
<tr>
<td>3.5 Virtual File System</td>
<td>5</td>
</tr>
<tr>
<td>4. Terminal I/O</td>
<td>8</td>
</tr>
<tr>
<td>5. File I/O</td>
<td>8</td>
</tr>
<tr>
<td>6. File Type</td>
<td>8</td>
</tr>
<tr>
<td>7. Work File</td>
<td>9</td>
</tr>
<tr>
<td>8. Descriptor File</td>
<td>9</td>
</tr>
<tr>
<td>9. Status Variables</td>
<td>9</td>
</tr>
</tbody>
</table>

**USER'S MANUAL**

1. ENTERING THE EDITOR.................................................. 13
2. EXITING THE EDITOR.................................................... 15
3. PRIMITIVE COMMANDS.................................................... 16
   3.1 Moving Commands................................................. 17
       jump............................................................... 18
       page............................................................ 18
       find............................................................ 18
3.2 Modifying Commands ........................................ 19
   insert .................................................. 19
   delete .................................................. 20
   copy .................................................... 21
   exchange ............................................... 21
   replace ................................................ 22
   display hexadecimal .................................. 23
   examine spelling ...................................... 23
   margin ................................................. 24
   adjust .................................................. 24

4. SETTING FORMAT AND MACROS ................................. 25
   4.1 SET format control .................................. 25
      Auto-Indent ......................................... 26
      Fill .................................................. 26
      Margins ............................................. 26
      Word ................................................ 26
   4.2 SET Macro Definations ............................... 27

5. MISCELLANEOUS COMMANDS ..................................... 27
   verify ............................................... 27

PROGRAM DOCUMENTATION

1. Introduction .............................................. 28
2. Input Output ............................................ 29
3. Data Structure ......................................... 30
4. Initialization .......................................... 32
5. File names.................................................. 33

appendix

1. Command Summary........................................ 35
2. Pascal Prefix.............................................. 37
3. Class Types............................................... 42
4. Sedit1..................................................... 52
1. INTRODUCTION

SEEDIT (for S creen EDIT or) is a full screen editor. This editor provides a CRT terminal user an efficient means of editing information displayed on the screen. This editor, in contrast to most editors, provides the user a visual machine interactive means to move, correct, change and manipulate displayed text files. Accordingly, full screen editors are much more efficient and effective than other usual types of editors.

SEEDIT is written in PASCAL language and implemented on Perkin-Elmer OS/32 MTM operating system. PASCAL was chosen because of its portability to other machines and operating systems. PASCAL is also a program language that is easy to maintain and modify. SEEDIT is designed to operate on a variety of terminals, including both dumb and smart CRT terminals. This versatility is achieved through a modular design such that selected modules can be resident in smart terminals thus reliving the central processor of part of the work. Other modules can be modified such that SEEDIT is blind to vendors differences in CRT terminal characteristics.

The following paragraphs provide a detailed description of SEEDIT.

2. MODES OF SEEDIT

SEEDIT operates in two modes: Command and Modify mode. When in
command are ignored. The modify mode is used to change or correct an existing text file. The modify mode too has a set of valid acceptable inputs. The terminal operator can easily change from one mode to the other by means of the terminals escape key or control-C key.

3. **ORGANIZATION OF SEDIT**

SEDIT is organized around a disk file and a set of modules and procedures as shown at figure 1. A discussion of the pertinent modules follows:

![Diagram of SEDIT organization](organizational_diagram.png)
3.1 Command Execution Module

The main part of SEDIT is the Command Execution Module. It is a set of procedures that are the primitive operations that can be performed on a file. The Command Execution Module is not concerned with file management or details of screen update. Its purpose is to execute the command. Figure 1 shows the relation of the Command Execution Module to the rest of SEDIT.

3.2 Added Command Definition Module

The use and scope of SEDIT can be changed by adding new commands. The definition of these new commands are, in effect, production rules of a grammar which translates the new commands (nonterminals) into the primitive commands (terminals) found in the Command Execution Module. Each production rule is expressed as a block of Pascal code found in the Added Command Definition Module. See Figure 1 for the relation of the Added Command Definition Module to SEDIT. The system programmer can thus modify SEDIT with the full power of Pascal. The programmer will have access to global variables in SEDIT and will not be concerned with file management or details of screen update. The programmer will only have to be aware of useful global variables and procedure names (the primitive commands) from the Command Execution Module.

By the selection of added commands, the system programmer can cause SEDIT to act like another editor. Thus SEDIT can be tailored to look like an editor that is familiar to the terminal operator.

3.3 Macro Execution Module

SEEDIT contains a powerful macro definition facility. At any time
during the execution of SEDIT the user can define and use commands called macros. These operations are processed by the Macro Execution Module (see Figure 1). The definitions include the name of the macro, may include repeat factors, and a series of other commands. These macro definitions further extend the flexibility of SEDIT and the ability to configure SEDIT to look like another editor. Once a Macro is defined, it is permanent for that file, and would be lost only upon EXIT of SEDIT. This is because of the capability of a Descriptive File associated to each file. The Descriptive Files are automatically created by SEDIT for each file that the user creates. More explanations are given in later sections. The name of the macro is not something like 'M1' but is whatever the user defines. The macro can be composed of any of the commands that are previously defined. It can contain nested parentheses and repeat factors. The macro can also contain variables to express a character string that is defined when the macro is executed.

3.4 Console And Keyboard Translators

SEEDIT can be configured to different terminals with the Console And Keyboard Translators, (see Figure 1).

When SEDIT operates in full screen mode, ASCII code control characters are sent to the console to move the cursor on the screen. Because these control characters are not standard, SEDIT must account for the brand and kind of terminal being used. This is accomplished by means of a table of terminal characteristics. Then the proper file may be read to configure SEDIT output to the terminal. The table will have to be maintained by the system programmer. The file will contain
information about both input from the keyboard and necessary output to the console screen.

3.5 Virtual File System

Another feature of SEDIT is the hidden file manipulation routines. See the Figure 1. The user can page through a file of any size both forward and backward without having to use any commands to read or write to disk. This manipulation of the file should look to the user as if all of the user's file were in memory. The use of memory for buffer space can be minimized and the manipulation of large files is not different from the manipulation of small files. This is accomplished most simply by the following procedure:

a) SEDIT first copies the file to be edited (if there is one) into a file called the Right File Stack. This file is written, a page at a time in reverse order. The last page of the user file is then the first page of the Right File Stack. See Figure 2.

b) Another file is allocated that is called the Left File Stack. This file is written a page at a time, but the pages are stored in first-to-last sequence.

c) There are also two buffers in memory called the Left Buffer Stack and Right Buffer Stack. Each buffer is a string of characters and the Right Buffer Stack contains the string in reverse order. The character at which the cursor is pointing is called the Current Character. The cursor position, or point of operation of commands, is always at the current character between the buffers.

d) When the user adds text, or moves the cursor down or to the left, characters are added to the Left Buffer Stack. When the Left Buffer
Stack is full, half of the buffer will be emptied by writing to the Left File Stack. This operation is hidden from the user. When the user is moving the cursor up or left, characters are taken from the Left Buffer Stack and placed into the Right Buffer Stack. When the Right Buffer Stack is full, it will be half emptied by writing to the Right File Stack. If at any time the Left or Right Buffer Stacks are emptied, they will be half filled by reading from the correct File Stack. This process will keep all disk I/O to a minimum and it will be done with efficient page oriented I/O.
CURRENT CURSOR POSITION
AND
CURRENT CHARACTER

array of char in memory

array of char in memory in reverse order

file of 512 byte pages

file of 512 byte pages in reverse order

RUN TIME EDIT BUFFERS
4. **TERMINAL I/O**

Output to the terminal is done with SVC1 calls that reside in the procedure SHIP_OUT which will output a line of known length. The output is done in Image mode.

Input is done with an SVC1 call in the Procedure INPUT. The call is Image mode Read. The echo back to the terminal is automatic and done by the programmable I/O device called a PASLA (Programmable Asynchronous Serial Line Adapter). This automatic echo is not desirable in this editor because it would prohibit the redefinition of characters. The function of echoing the character back is called ECHOLEX. This function must be turned off when the editor is in use and turned on when done. The calls to control the echoplex should reside in the CSS file that evokes SEDIT. Then if SEDIT crashes (which is very unlikely) the execution of the CSS file will continue and the echoplex function will be enabled again.

The redefinition of the characters can occur in macro definitions, consol translator or the keyboard translator.

5. **FILE I/O**

File I/O will be done with the prefix procedures OPEN, CLOSE, PUT, and GET. These procedures support page I/O.

6. **FILE TYPES**

SEDIT will support clean files. This means the files do not contain nonprinting characters, such as control characters, nor special characters such as paragraph indicators, that might be used for special
purposes in some editors. The files are the same as files currently supported with other programs such as Script, PPF, and the Pascal compilers.

7. **WORK FILE**

   SEDIT will support a workfile. A workfile will store the default values of status variables. It will keep the name of the file the user was last working on. The user will be asked if he wants SEDIT to read in the file that he last edited. If the user replies with a yes, the file will be read and its descriptor file will be read. If there is not a descriptor file associated with the indicated file, the default values of the workfile will be used.

8. **DESCRIPTOR FILE**

   With each file the user uses, there will be an associated file called the file descriptor. It will have the same name as the file but a different extension. The default file descriptor is called the workfile. Like the workfile, the descriptor contains information about the SEDIT status variables that determine the environment of SEDIT and also contains information that pertains to that file.

9. **STATUS VARIABLES**

   The status variables are the global variables in SEDIT that describe the current status of the editor. The command execution module and added command definition module will have access to these variables. Some commands will directly modify the status variables and some indirectly.
A list of the Status Variables follows:

Current screen column cursor position : integer

Current screen row cursor position : integer

Current line cursor position : integer

Current column in line cursor position : integer

Current character at cursor : char

eofc : boolean , if eofc is true the cursor is pointing at the last character in the file, end of file char.

tofc : boolean , if tofc is true the cursor is pointing at the first character in the file, top of file char.

eofl : boolean , if eofl is true the cursor is in the last line in the file, end of file line.

tofl : boolean , if tofl is true the cursor is in the first line in the file, top of file line.

left margin : integer , column number

right margin : integer , column number.
automatic indentation : boolean, if true and in insert mode the next
    line will begin at the same column as the last one.

last line starting column : integer

script : boolean, if true the text will automatically be adjusted in
    the left and right margins.

token definition : boolean, if true the search argument applies to
    words, if false it applies to character strings.

current date : integer

last date : integer, last data the file was updated.

current search argument : char string

current change string : char string

paragraph margin : integer, column number.

tab sets : array of integer, column numbers.

name of current file : char string

starting anchor : integer, absolute position in the file.
ending anchor : integer, absolute position in the file.

current direction : char

line mode : boolean, if true line numbers will be shown.
1. ENTERING SEDIT

If the user types SEDIT [filename][.ext], where filename and the extension are optional, the full screen editor will be run. If the user types the filename and not the extension, .PAS will be tried first. If the file is not found then .TXT will be used. If the file is not found the following menu will be presented to the user.

Input [filename][.ext] file not found.
PAS and TXT was searched for.
T)ype new file name.
c)reate new file.

The [filename][.ext] is where the name typed by the user is echoed. If 'T' is typed, SEDIT will ask the user to enter the new file name. If 'C' is typed, the current file name and extention becomes the new file. If the extension was typed by the user the above menu will appear without the second line and the extension will be echoed.
If the user does not type in the file name, SEDIT will automatically read the default descriptor file called the work file. The file name is SEDIT.DES, and a copy exists on each account. The work file and descriptor files contain values necessary to configure SEDIT. The name of the last file the user edited with SEDIT is stored in this work file.
SEDIT then gives the user the following menu.

The file last edited was
filename.ext
touch return to use this file,
else type in the new file name.

SEDIT responds with the same search as described above, where the user
does not have to type in the extension.

SEDIT will then search for the proper descriptor file which has the name
filename.DES. If the file is found, it is read and the environment of
SEDIT is set the way it was when SEDIT was last used with this file. If
the file is not found, the default values from SEDIT.DES will be used
and the user will be informed of this action. Once the file to be
edited is specified or found, the following prompt line will be
displayed:

>SEDIT: I nsert D(elete Q uit R( place J(ump F(ind P(age A(djust ?(more

The '>' on the left side of the prompt line indicates the current status
of direction. If the user types '?' an extension of the prompt line
will appear as follows;

>SEDIT: X(change M(argin C(opy S(et H(examine V(erify
2. **EXITING THE EDITOR**

The user may exit the editor by typing 'Q' for QUIT. On entering the quit mode, the following prompt will be displayed.

>Quit:

U(update - the current file and leave
E(xit - without updating
Return - to the editor without updating
W(rite - to a new file
P(rint - to an output device

One of the five options must be selected by typing the appropriate letter. The options are described below:

U(update - The Editor will replace the old file with the file just modified and output the file's descriptor file.
E(xit - The Editor leaves without making any changes to the file nor to the descriptor file.
Return - The Editor returns without updating, the cursor is returned to the exact place in the file it occupied when 'Q' was typed.
W(rite - With this option, a further prompt appears:

>Quit: 'name of output file', <return>

The modified file will be written to the indicated file name and the descriptor file will be output. If it is an existing file, the modified file will replace it. SEDIT, however, warns the user of the existing file

>File xxx.xxx already existed, do you want to replace the file?
"N" will let the user to enter a different file name
"Y" will replace the file

The command can be aborted by entering a <return> instead of the
file name; return will be to the Editor.

After the file has been written to the disk the prompt line will
be:

>Quit:

Writing . . . .

Your file is nnnn bytes long.

Do you want to E(xit from of Return to the Editor?

Typing "E" exits from the Editor. Typing "R" returns the cursor to
the same position in the file as when the "Q" was typed.

Although not discussed above, the user will also be given the
option to output a file with line numbers, and the option to output
a file to a printer.

P(rint - A prompt will appear and the user may type in the name of
the output device to print to. The user can have the lines
numbered if she/he wishes.

3. PRIMITIVE COMMANDS

These commands are used for cursor movement, or modifying text.
REPEAT FACTORS are allowed by some of these commands to repeat the
action of the command as many times as indicated by the immediately
proceeding number. For example, entering

2 <down-arrow>

will cause the <down-arrow> command to be repeated twice, moving the
cursor down two lines.
A slash (/) typed before the command indicates an infinite number of
repeats.
Some commands are directional. If their direction is forward, they
operate forward through the file; if backwards, they operate in the
reverse.

3.1 MOVING COMMANDS

These commands move the cursor from one location to another, to
position it for the next editing function.
The REPEAT FACTORS may be used with all of these commands. They
include:

<ctrl>H, <left arrow> moves cursor left
<ctrl>L, <right arrow> moves cursor right
<ctrl>K, <up arrow> moves cursor up
<ctrl>J, <down arrow> moves cursor down
"<" changes the direction to backward
">" changes the direction to forward
<space> moves direction
<back-space> moves left
<return> moves to the beginning of the
next line

Direction is always indicated by an arrow ( > or < ) in front of the
prompt line. The direction is forward when the Editor is entered, but
can be changed by typing the appropriate command whenever the "Edit:"
prompt line is present.
Commands that do not have special function keys on the CRT keyboard are as follows:

3.1.1 JUMP
3.1.2 PAGE
3.1.3 FIND

3.1.1 JUMP

JUMP mode is entered by typing "J" for J(ump). When the JUMP mode is entered, the following prompt line will be displayed:

JUMP: B(eginning E(nd, <+n>, <-n>, <d>
"B" moves the cursor to the beginning of the file.
"E" moves the cursor to the end of the file
"<+n>" moves the cursor n lines forward
"<-n>" moves the cursor n lines backwards
"d" moves the cursor to line number d Note that n must be an integer, whereas d may be an integer or real number

3.1.2 PAGE

The PAGE command is executed in response to typing "P". PAGE moves the cursor one whole page up or down, depending on the direction of the arrow at the beginning of the prompt line. The cursor moves to the start of the top line. To move several pages at one time, the repeat factor may be used with this command.

3.1.3 FIND

FIND mode is entered by typing "F". On entering FIND mode, one of the prompt lines below will appear:
>Find[n]: S(tring <target>=)
>Find[n]: W(ord <target>=)

FIND mode finds repeat factor [n] number of occurrences of the <target> string. Text is scanned in the direction of the arrow on the prompt line. The target pattern can be found only if it appears in that section of the text.

In STRING mode, the Editor will look for any occurrences of the target string, in WORD mode, it looks for an isolated occurrence. Isolation means a string is surrounded by any combination of delimiters. To use the STRING mode, type "S" after the prompt line and before the target line, to use WORD, type "W". The default value found in the Environment may be overridden by typing "S" or "W".

The repeat factor can be used, and it must be typed before typing "F".

3.2 MODIFYING COMMANDS

The following operations are used to change the text of a file.

3.2.1 INSERT
3.2.2 DELETE
3.2.3 COPY
3.2.4 EXCHANGE
3.2.5 REPLACE
3.2.6 DISPLAY HEX
3.2.7 EXamine SPELLING
3.2.8 MARGIN
3.2.9 ADJUST

3.2.1 INSERT
INSERT mode is entered by typing "I" for I(nsert). The following prompt line will be displayed:

Insert: <bs> a char, <del> a line, <esc>, <ctrl>C

The user can simply start typing and the text will be inserted into the file at the point where the cursor is. The text to the right side of the cursor will shift to the right or to the next line.

"<bs> a char" a character is deleted by backspacing
"<del> a line" an entire line is deleted
"<esc>" all inserted information will be deleted
"<ctrl>C" all inserted information is put into the file, and put in the copy buffer. all previous information is the copy buffer is lost

3.2.2 DELETE

DELETE mode is entered by typing "D". The following prompt line will be displayed:

>Delete: <moving commands>, <esc>, <ctrl>C

When entering the delete mode, the cursor is positioned at the first character to be deleted. This position is called starting anchor. The user can move cursor around using the moving commands. The point at which the cursor is when the delete mode is exited is called the ending anchor.

<esc> exits edit mode without any changes to the file
<ctrl>C exits edit mode, deletes the text between the starting and ending anchors
3.2.3 COPY

COPY mode is entered by typing "C" for Copy. When entering the COPY mode, the following prompt line appears:

>COPY: B(uffer F(ile <esc>

"B" copy text in the copy buffer into the file at the location of the cursor when "C" was typed.

On completion of the COPY, the cursor returns to a position immediately preceding the text that was copied. The use of the COPY does not change the contents of the copy buffer.

"F" copy text from another file.

The following prompt line will appear:

> COPY: ENTER THE FILE TO COPY FROM OR "END"

"file name"

> COPY: ENTER FIRST LINE OR "END"

"first line"

> COPY: ENTER LAST LINE OR "ALL"

"last line"

> COPY: ENTER THE FILE TO COPY FROM OR "END"

"end"

The "F" command allows the user to copy several files or different sections of different files.

The use of COPY does not change the contents of the file being copied.

3.2.4 EXCHANGE

EXCHANGE mode is entered by typing "X". On entering the EXCHANGE mode, the following prompt line appears:

EXCHANGE: TEXT <bs> a char, <esc>, <ctrl>C
EXCHANGE replaces one character in a file for each character of text entered. Backspacing one character will cause the original character in that position to reappear. Typing <esc> leaves the EXCHANGE mode without making any of the changes indicated since entering the mode. Typing <ctrl>C accepts the changes as part of the file.

3.2.5 REPLACE

REPLACE mode is entered by typing "R". On entering the REPLACE mode, one of the following prompt lines will appear.

>Replace[n]: String V(fy <target> <sub> =>

>Replace[n]: W(ord V(fy <target> <sub> =>

If repeat factor is used, it should be typed before "R". REPLACE mode finds the repeat factor [n] occurrence of the target string and replaces it with the substitute string.

The verify option V(fy allows examination of the target string (to the limit set by the repeat factor) to decide if it is to be replaced. Whenever REPLACE has found the target pattern in the file and verification has been requested, the following prompt line appears:

>Replace: <esc> aborts, 'r' replaces, ' ' doesn't

Typing an "R" will cause replacement, typing a space will cause a continuation of a search for the next occurrence if the limit of the repeat factor has not yet been reached. The repeat factor counts the number of times an occurrence is found, not the number of times "R" is entered. If a slash(\) is used as the repeat factor, every occurrence of the target string will be replaced.

If the Editor cannot find the target string, the error message appears:

ERROR: Pattern not in the file. Press <space bar> to continue.
Example:

>REPLACE [2]: S(string V(fy <target> <sub>) => old-string/new-string/

>REPLACE [2]: S(string V(fy <target> <sub> => W/old/new/

Note that the slash (/) is used here as a delimiter, you may use any other character as a delimiter besides "S", "W", and "V".

3.2.6 DISPLAY HEXDECIMAL

The command display hexadecimal is executed by the user typing 'H' for hex. The current line (which the cursor is in) is displayed. Followed by a line of the first digit of the hexadecimal number for each character. Followed by a line of the second digit of the hexadecimal number for each character.

An example follows.

Contents of SEDIT
2246676672662544450
003F745E430F0035494A

3.2.7 EXAMIN SPELLING

SEDIT will contain a facility for the user to check spelling. When the user types 'E' for Examine Spelling the spelling of all the words from the current cursor position to the end of the current line will be checked. The repeat factor when used will cause more lines to be checked or the rest of the file. A word is defined as a character string delimited by spaces, where a sequence of hyphen-linefeed is ignored and a period is treated as a following delimiter. A word is said to be spelled correctly if it occurs in the dictionary. The dictionary is a file (DICTION.TXT) that SEDIT has access to that is just a list of words spelled correctly. If the word does not exist in the dictionary, or is spelled incorrectly the cursor is placed immediately
following that word, and the user is queried with the prompt line.

Spelling error. C)orrect spelling, <ctrl>C continue, <esc> escape
If the user types 'C', this indicates that the word is spelled correctly
and it is placed in the dictionary. If the user types the <esc> key, he
then exits from the spelling mode and returns to SEDIT mode where he can
correct the spelling. If the user types the control 'C' key, the
spelling mode is continued and the next word is checked.

3.2.8 MARGIN

The MARGIN command is executed by typing "M". This command does
not appear on the prompt line, MARGIN is Environment dependent and
cannot be executed except when FILLING is set to true and AUTO-INDENT is
set to false. See SET command to set the Environment.
Three parameters are used by margin: right-, left- and paragraph-
margin. MARGIN deals with only one paragraph at a time. It realigns
the text to compress it as much as possible without violating the three
margins. To set the margin values, see SET mode.
For purposes of formatting, a paragraph is defined as the lines of text
occurring between two blank lines. To MARGIN a paragraph, move the
cursor anywhere in the paragraph and type "M".
Any given line of text can be protected from being MARGINed if the
command character appears as the first non-blank character on the line.
The MARGIN treats that line as though it were entirely blank.

3.2.9 ADJUST

ADJUST mode is entered by typing "A". On entering the ADJUST mode,
the following prompt line appears:
>Adjust: L(eftright)C(enter <left,right,up,down-arrow> <ctrl>C

ADJUST mode adjusts indentation on a line-by-line basis. On any line, the right-arrow and left-arrow commands move the whole line one space to the right or left, respectively, each time the arrow is typed. Type <ctrl>C when indentation is adjusted.

To adjust a sequence of lines, adjust one, then use the up-arrow and down-arrow commands to adjust the line above or below, respectively. Repeat factors can be used before any of the arrows.

"L" and "R" are used to left- and right-justify lines to margins set in the Environment. "C" will center the line between the set margins. Typing an up- or down-arrow will justify or center the line above or below to the same specification.

4. SETTING FORMAT AND MACROS

SEDIT allows for formatting of text. A SET command is used to set the Environment for each of the formatting commands.

The macro definitions are entered with the set command.

SET mode is entered by typing "S". This command does not appear on the prompt line. On entering the SET mode, the following prompt line appears:

>Set: E(nvironment, M(acos, <esc>

4.1 FORMAT CONTROL

Through SET mode, SEDIT enables the user to set the Environment needs of the editing to be done. When "E" for E(nvironment is typed,
the following prompt appears:

```
>Environment: <options> <ctrl>C

Auto indent    true
Filling        false
Left margin    0
Right margin   79
Para margin    5
Word def       true
```

By typing the appropriate letter, any or all of the options may be changed. Each option is described below.

4.1.1 AUTO INDENT

AUTO ANDENT is set to true by typing "AT" and to false by typing "AF". Affects only the INSERT mode.

4.1.2 FILLING

FILLING is set to true by typing "FT" and to false by "FF". Affects the INSERT mode and allows the MARGIN command to function.

4.1.3 MARGINS

MARGINS are set by typing the appropriate letter ("L", "R" or "P") followed by a positive integer of less than 4 digits and a <space>. When Filling is true, the margins set here are the ones that affect INSERT and MARGIN and the center and justify commands in ADJUST.

4.1.4 WORD

WORD is set to true by typing "WT" and false by "WF". If true,
word is the default, if false string is the default. Affects FIND and REPLACE.

4.2 MACRO DEFINITIONS

Through SET mode, SEDIT enables the user to define the macros. When "M" is typed for Macro, the macros will be listed on the screen and the user will be given the option to change them if he wishes.

5. MISCELLANEOUS COMMANDS

5.1 VERIFY

The VERIFY command is executed in response to typing "V". The text is verified by redisplaying the text on the screen. The Editor attempts to adjust the window so the cursor is at the center of the screen.
1. INTRODUCTION

This editor is not a copy of the UCSD Full Screen L2 editor. It is intended to have a subset of commands equal to the UCSD editor. It also works, whereas the Reverse File Paging function of the UCSD Full Screen editor does not work. The interval structure of the UCSD Full Screen editor is unknown. Any similarities between the SEDIT program and the UCSD program are strictly coincidental.

The actual program to date is a working subset of the ideal described previously. This subset includes the following commands:

movement commands (up, down, left, right direction control, page, return, jump to beginning, and jump to end); verify command.

The following modes exist:

Insert, Delete, Exchange.

An additional mode has been added and is called Fix. Fix mode will let the user remove nonprinting characters (except for line feeds and tab) from his file.

The excessive use of global variables is considered to be a poor trait of large programs. This attitude was taken very seriously in the construction of SEDIT. In the 2700 line program, there are 7 global variables. The former sections of this document list many more global variables than were ever implemented. This eliminates the necessity of
maintaining the value of these at all times, but makes it necessary to
calculate the value when it is needed.

2. **INPUT OUTPUT**

A problem with SEDIT is the absence of appropriate IO interfaces. The Pascal routines Accept and Display are well-standardized single-character oriented IO. But the standardization is of the interface between Pascal and the operating system, not between Pascal and the user. On the 8/32 minicomputer under the MTM operating system with Robert Young's Pascal Driver the Accept and Display functions are buffered by the driver as line-oriented IO. This can be overcome with the use of SVC1 calls to the operating system with Image mode single character read and multiple character write. This method is very effective. In other operating systems Accept and Display might be adequate.

The prompts for IO to the console are disabled by the PROMPT command in the SEDIT.CSS file so that they will not interfere with the screen display.

The existing CRT driver will buffer input characters for the Pascal program only when the Pascal program is ready to Accept them. If the computer system is lightly used, SEDIT is almost always waiting for input. If SEDIT has a time-consuming task and is not asking the CRT driver for input, any characters the user types will be buffered as an MTM operating system command. Because the prompts are disabled, the user has no way of knowing when this state exists.

The desirable driver will buffer all input characters for the Pascal
program, and give the user access to the MTM operating system only when he hits break. The driver could use a circular ring buffer or double buffers to implement this feature (called the type ahead feature).

2. DATA STRUCTURES

A common problem with editor data structures is with insertion and deletion. If the user's text is in memory in an array of characters or an array of lines, the inevitable occurs: information will be inserted or deleted. This requires that all of the contents of the array on one side of the operation be shifted. Pointers may be used to keep track of the string of text. Pointers require much manipulation and make a program very hard to read. In effect, they are messy. Shifting the contents of arrays periodically requires large amounts of time.

Another alternative is to implement the user's file as a string of characters that is broken at the current point of action. Each half of the string is started in a stack. A string such as "eat that cat" with the current point of action at the "c" will exist with "eat that" in the left stack and "at" in the right stack. The character "space" will be at the top of the left stack and the "a" will be at the top of the right stack. The character "c" will be in the current character buffer of type CHAR. Deletions can be done by removing information from the top of either stack. Additions can be done by adding characters to the top of either stack. This method greatly increases the ease of insertion and deletion, but also increases the overhead for moving through the user's file.

One of the major good points of SEDIT is that it will work with files
of any length. The only limits are those imposed by the operating system. This is because of the data structures used to contain the user's file. An important point to realize is that the right stack must contain information in reverse order relative to the left stack.

To simplify the interface of the program with this data structure, only 4 procedure calls are used. The main program does not have access to any of the variables in these procedures other than the actual parameter CURRENT_CHAR. The procedures are:

Add(CURRENT_CHAR) which will add current character to the left stack.

Delete(CURRENT_CHAR) which will pop a character from the right stack and assign it to CURRENT_CHAR, destroying the original character.

MOVE.RIGHT(CURRENT_CHAR) which will push the CURRENT_CHAR onto the right stack and pop a character off of the left stack into CURRENT_CHAR.

MOVE.LEFT(CURRENT_CHAR) which will push the CURRENT_CHAR onto the left stack and pop a character off of the right stack into CURRENT_CHAR.

By using these 4 procedures the entire file can be examined and manipulated. The system is analogous to a Turing machine, where the CURRENT_CHAR is the position of the read head and the tape is the character string. When applied to cursor movement on the CRT screen, the current cursor position is analogous to the read head position. A
major source of confusion when thinking about this is the concept of moving in a certain direction. For example, when the string "1234" is in the data structure and the MOVE_LEFT (CURRENT_CHAR) procedure is executed, if the CURRENT_CHAR had been "3" it is now "4". But if the user has "1234" on the screen with the current cursor position on "3" and executes a left arrow, the new CURRENT_CHAR is "2". The Move procedure implies movement of the character string past a certain point. The cursor movement commands imply movement of the point along the character string.

The user's file when in SEDIT is a character string delimited by the following characters: EM NL character string NL EM. When manipulating the file, the procedures in SEDIT detect the starting or ending of the file when CURRENT_CHAR = EM.

The left and right stacks exist in memory; they are arrays of characters. Their size is static. When the user fills a stack, it is half-emptied into a file. The file for the right stack is called the right file, for the left stack the left file. When a stack is emptied it is half-filled from a file. The stacks in memory called buffer stacks are actually decks, implemented as a circular ring buffer that can be popped and pushed from both ends. The emptying and filling of the buffer stacks is done 4 pages at a time by popping or pushing on the bottom of the stacks. Boolean parameters in the procedure calls terminate filling a buffer if the file is empty.

4. **INITIALIZATION**

The initialization of a file involves reading it into memory and
outputing it into the right file so that the pages in the right file stack are in reverse order. Next, the line numbers are added by moving the character left through the CURRENT_CHAR. When an NL character is detected, the line number characters are added. When the EM character is reached, the process is stopped, all of the file is shifted back to the right side of CURRENT_CHAR , and the screen is filled with the Verify procedure.

5. PROGRAM FILE NAMES

Line numbers are present in the working subset. The inhibiting of line number display has been provided for with the global Boolean variable SHOW-LINE.NUMBERS, although this has not been tested.

Line numbers are manipulated and created in a manner like Fedit. To make handling of line numbers simple, they are inserted into the runtime character string which is in the users file. The manipulation of the cursor by the user is limited to the text outside the line number field. This field exists after every character except the last one. Each field is 8 characters wide; four for the integer, one for the decimal point, two for decimal digits and one space.

The current version of SEDIT proves to be an effective means for editing a file. However, the time involved in initializing and terminating the use of a large file decreases the attractiveness to the user. On the existing minicomputer with hard disks the initialization of a 1000-line file may take 60 seconds; stepping from the beginning of the file to the end while editing may take 30 seconds. Of course, small steps are instantaneous.
The SEDIT program exists in 3 files: the PREFIX and SVC types are in SYSMAC.PAS, the data buffer management routines are in MACRO.PAS, the main program is in SEDIT1.PAS. These programs are listed in the appendix.

Class types are used for convenience; they are available with the PASCAL used in this computer science department. The classes can be omitted with the use of an editor and a few hours' time of a knowledgable programmer to gain portability to other operating systems.
APPENDIX

SEDIT Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEDIT [filename][.ext]........</td>
<td>entering SEDIT</td>
</tr>
<tr>
<td>Q..................</td>
<td>exiting SEDIT</td>
</tr>
<tr>
<td>&lt;ctrl&gt;H, &lt;left arrow&gt;........</td>
<td>move cursor left*</td>
</tr>
<tr>
<td>&lt;ctrl&gt;L, &lt;right arrow&gt;........</td>
<td>move cursor right*</td>
</tr>
<tr>
<td>&lt;ctrl&gt;K, &lt;up arrow&gt;........</td>
<td>move cursor up*</td>
</tr>
<tr>
<td>&lt;ctrl&gt;J, &lt;down arrow&gt;........</td>
<td>move cursor down*</td>
</tr>
<tr>
<td>....................</td>
<td>change direction to backwards</td>
</tr>
<tr>
<td>....................</td>
<td>change direction to forwards</td>
</tr>
<tr>
<td>&lt;space&gt;...............</td>
<td>move direction*</td>
</tr>
<tr>
<td>&lt;back-space&gt;...........</td>
<td>move against direction*</td>
</tr>
<tr>
<td>&lt;return&gt;.............</td>
<td>move to next line</td>
</tr>
<tr>
<td>A....................</td>
<td>adjust line in margins</td>
</tr>
<tr>
<td>C....................</td>
<td>copy from buffer or file</td>
</tr>
<tr>
<td>D....................</td>
<td>delete text</td>
</tr>
<tr>
<td>E....................</td>
<td>examine spelling</td>
</tr>
<tr>
<td>F....................</td>
<td>find a string or word*</td>
</tr>
<tr>
<td>H....................</td>
<td>display a line in hexadecimal*</td>
</tr>
<tr>
<td>I....................</td>
<td>insert text</td>
</tr>
<tr>
<td>J....................</td>
<td>jump to a different place in the text</td>
</tr>
<tr>
<td>M....................</td>
<td>margin adjustment of a paragraph</td>
</tr>
</tbody>
</table>
P......................... page thru the text*
R......................... replace a string with another*
S.......................... set format control and macro definitions
V.......................... verify text, fill screen
X.......................... exchange text with what is typed

* These commands may be preceded with a repeat factor.
Pascal Prefix, file name SYSMAC.PAS

"$MACRO FULL_PREFIX"

"$EJECT"
"PER BRINCH HANSEN
INFORMATION SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
UTILITY PROGRAMS FOR
THE SOLO SYSTEM
18 MAY 1975

1 DEC 1976"

"#################
# PREFIX #
################"

CONST NL = '(:10:); FF = '(:12:); CR = '(:13:); EM = '(:25:);
CONST PAGELENGTH = 512;
TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF CHAR;
CONST LINELENGTH = 132;
TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;
CONST IDLENGTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;
TYPE FILE = 1..5;
TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);
TYPE FILEATTR = RECORD
  KIND: FILEKIND;
  ADDR: INTEGER;
  PROTECTED: BOOLEAN;
  NOTUSED: ARRAY (.1..5.) OF INTEGER
END;
TYPE IODEVICE =
  (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, CARDDDEVICE);
TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);
TYPE IOARG = (WRITEEOF, Rewind, UPSpace, BACKSPACE);
TYPE IORESULT =
  (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
   ENDFILE, END_MEDIUM, START_MEDIUM);

TYPE IOPARAM = RECORD
  OPERATION: IOPERATION;
  STATUS: IORESULT;
  ARG: IOARG
END;

TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);

TYPE ARGTAG =
  (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);

TYPE POINTER = BOOLEAN;

TYPE ARGTYPE = RECORD
  CASE TAG: ARGTAG OF
    NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
    INTTYPE: (INT: INTEGER);
    IDTYPE: (ID: IDENTIFIER);
    PTRTYPE: (PTR: POINTER)
  END;

CONST MAXARG = 10;

TYPE ARGLIST = ARRAY (. . . MAXARG.) OF ARGTYPE;

TYPE ARGSEQ = (INP, OUT);

TYPE PROGRESS =
  (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR,
   HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);

PROCEDURE READ(VAR C: CHAR);
PROCEDURE WRITE(C: CHAR);

PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
PROCEDURE CLOSE(F: FILE);
PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
FUNCTION LENGTH(P: FILE): INTEGER;
PROCEDURE MARK(VAR TOP: INTEGER);
PROCEDURE RELEASE(TOP: INTEGER);
PROCEDURE IDENTIFY(HEADER: LINE);
PROCEDURE ACCEPT(VAR C: CHAR);
PROCEDURE DISPLAY(C: CHAR);
PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);
PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);
PROCEDURE READLINE(VAR TEXT: UNIV LINE);
PROCEDURE WRITELINE(TEXT: UNIV LINE);
PROCEDURE READARG(S: ARGSEQ; VAR ARG: ARGTYPE);
PROCEDURE WRITEARG(S: ARGSEQ; ARG: ARGTYPE);
PROCEDURE LOOKUP(ID: IDENTIFIER; VAR ATTR: FILEATTR; VAR FOUND: BOOLEAN);
PROCEDURE IOTRANSFER
  (DEVICE: IODEVICE; VAR PARAM: IOPARAM; VAR BLOCK: UNIV PAGE);
PROCEDURE IOMOVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);
FUNCTION TASK: TASKKIND;
PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;
  VAR LINE: INTEGER; VAR RESULT: PROGRESS);
PROCEDURE RESET( LU : INTEGER);
PROCEDURE BREAKPT ( LN : INTEGER );
"%ENDMACRO FULL_PREFIX"
"%MACRO PREFIX"
CONST NL='(:10:)'; FF='(:12:)'; CR='(:13:)'; EM='(:25:)';
CONST PAGELENGTH=512; LINELENGTH=132;
TYPE PAGE=ARRAY [1..PAGELENGTH] OF CHAR;
  LINE=ARRAY [1..LINELENGTH] OF CHAR;
PROCEDURE READ(VAR C: CHAR);
PROCEDURE WRITE(C: CHAR);
PROCEDURE OPEN(FILE:INTEGER; ID:UNIV LINE; VAR FOUND:BOOLEAN);
PROCEDURE CLOSE(FILE:INTEGER);
PROCEDURE GET(FILE:INTEGER; PAGE_NO:INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(FILE:INTEGER; PAGE_NO:INTEGER; VAR BLOCK: UNIV PAGE);
FUNCTION LENGTH:INTEGER; "NOT IMPLEMENTED"
PROCEDURE MARK(VAR TOP: INTEGER);
PROCEDURE RELEASE(TOP: INTEGER);
PROCEDURE IDENTIFY; "NOT IMPLEMENTED"
PROCEDURE ACCEPT(VAR C: CHAR);
PROCEDURE DISPLAY(C: CHAR);
"%ENDMACRO PREFIX"
"%MACRO SVC_TYPES"
"******************************************************************************
  "
  " OS/32MT3 SVC INTERFACE ROUTINE TYPES
  "
  "******************************************************************************"
"MISCELLANEOUS DATA TYPES"

TYPE CHAR1 = PACKED ARRAY [1..1] OF CHAR;
TYPE CHAR3 = PACKED ARRAY [1..3] OF CHAR;
TYPE CHAR8 = PACKED ARRAY [1..8] OF CHAR;
TYPE CHAR4 = PACKED ARRAY [1..4] OF CHAR;
TYPE CHAR16 = ARRAY [1..16] OF CHAR;
TYPE CHAR28 = ARRAY [1..28] OF CHAR;

"SVC1 PARAMETER BLOCK"

TYPE SVC1_BLOCK = RECORD
  SVC1_FUNC: BYTE;
  SVC1_LU: BYTE;
  SVC1_STAT: BYTE;
  SVC1_DEV_STAT: BYTE;
  SVC1_BUFSTART: INTEGER;
  SVC1_BUFEND: INTEGER;
  SVC1_BUFEND_PLUS_SIZE_BUFFER - 1;
  SVC1_RANDOM_ADDR: INTEGER;
  SVC1_XFER_LEN: INTEGER;
  SVC1_RESERVED: INTEGER;
END;

"SVC 1 FUNCTION CODES"

CONST SVC1_DATA_XFER = #00;
SVC1_COMMAND = #80;
SVC1_READ = #40;
SVC1_WRITE = #20;
SVC1_TESTSET = #60;
SVC1_TESTIO = #00;
SVC1_ASCII = #00;
SVC1_BINARY = #10;
SVC1_PROCEED = #00;
SVC1_WAIT = #08;
SVC1_SEQL = #00;
SVC1_RANDOM = #04;
SVC1_CWAIT = #00;
SVC1_UNC_PROC = #02;
SVC1_FORMAT = #00;
SVC1_IMAGE = #01;
SVC1_REW = #40;
SVC1_BSR = #20;
SVC1_FSR = #10;
SVC1_WFM = #08;
SVC1_FSF = #04;
SVC1_BSF = #02;
SVC1_RESV_FN = #01;

"SVC 1 DEVICE-INDEPENDENT STATUS CODES"

CONST SVC1_OK = #00;
SVC1_ERROR = #80;
SVC1_ILGFM = #40;
SVC1_DU = #20;
SVC1_EOM = #10;
SVC1_EOF = #08;
SVC1_UNRV = #04;
SVC1_RECV = #02;
SVC1_ILGLU = #01;
SVC1_DEVBUSY = #7F;

"FILE_DESCRIPTOR FOR SVC7 REQUESTS"

TYPE FD_TYPE = PACKED RECORD
  VOLN: CHAR4;
  "VOLUME NAME"
  FN: CHAR6;
  "FILE NAME"
  EXTN: CHAR3;
  "EXTENSION"
ACCT: CHAR; "ACCOUNT NUMBER CODE"
END;

"SVC 7 PARAMETER BLOCK"

TYPE SVC7_BLOCK = RECORD
  SVC7_CMD: BYTE; "COMMAND"
  SVC7_MOD: BYTE; "MODIFIER/DEVICE TYPE"
  SVC7_STAT: BYTE; "STATUS"
  SVC7_LU: BYTE; "LOGICAL UNIT NUMBER"
  SVC7_KEYS: SHORTINTEGER; "READ/WRITE KEYS"
  SVC7_RECLEN: SHORTINTEGER; "LOGICAL RECORD LENGTH"
  SVC7_FD: FD_TYPE; "FILE DESCRIPTOR"
  SVC7_SIZE: INTEGER; "FILE(INDEX) SIZE"
END;

"SVC 7 COMMAND CODES"

CONST SVC7_ALLOC = #80; SVC7_ASSIGN = #40;
SVC7_CHAP = #20; SVC7_RENAME = #10;
SVC7_REPROT = #08; SVC7_CLOSE = #04;
SVC7_DELETE = #02; SVC7_CHECKPT = #01;
SVC7_FETCH_ATTR = #00;

"SVC 7 MODIFIER CODES - ACCESS PRIVILEGES"

CONST SVC7_AP_SRO = #00; SVC7_AP_ERO = #20;
SVC7_AP_SMO = #40; SVC7_AP_EWO = #60;
SVC7_AP_SRW = #80; SVC7_AP_SREW = #A0;
SVC7_AP_ERSW = #C0; SVC7_AP_ERW = #E0;

"SVC 7 MODIFIER CODES - BUFFERING/FIELD TYPE"

CONST SVC7_BUF_DEFAULT = #00; SVC7_BUF_PHYS = #08;
SVC7_BUF_LOG = #10; SVC7_BUF_SVC15 = #18;
SVC7_FTYPE_CONTIG = #00; SVC7_FTYPE_CHAIN = #01;
SVC7_FTYPE_INDEX = #02; SVC7_FTYPE_ITAM = #07;

"ENDMACRO SVC_TYPES"

"MACRO STANDARD_EXTERN"
PROCEDURE BREAKPNT(LINENO:INTEGER);EXTERN;
PROCEDURE SVC1(VAR BLOCK:SVC1_BLOCK);EXTERN;
PROCEDURE SVC3(RETURN_CODE:INTEGER);EXTERN;
PROCEDURE SVC7(VAR BLOCK:SVC7_BLOCK);EXTERN;
"ENDMACRO STANDARD_EXTERN"
"#MACRO CLASS_TYPES"

(* SVC7 CALLS FOR ALLOCATE, RENAME, AND ASSIGN. *)

; TYPE SVC7_CALLS
   = CLASS

   ; VAR SVC7_B : SVC7_BLOCK

   ; PROCEDURE SVC7 ( SVC7B : SVC7_BLOCK )
      ; EXTERN

   ; PROCEDURE ENTRY ALLOCATE_FILE
      ( P_VOLN : CHAR4
         ; P_FN : CHAR8
         ; P_EXTN : CHAR3
         ; VAR ALREADY_THERE : BOOLEAN
      )

   ; BEGIN
      WITH SVC7_B
      DO BEGIN
         SVC7_CMD := SVC7_ALLOC
         ; SVC7_MOD := SVC7_AP_SREW + SVC7_FTYPE_INDEX
         ; SVC7_STAT := 0
         ; SVC7_KEYS := 0
         ; SVC7_RELEN := 512
         ; WITH SVC7_FD
         DO BEGIN
            VOLN := P_VOLN
            ; FN := P_FN
            ; EXTN := P_EXTN
            ; ACCT := 'P'
         END
         END
         ; SVC7 ( SVC7_B )
         ; IF SVC7_B . SVC7_STAT = 4
         THEN ALREADY_THERE := TRUE
         END

   ; PROCEDURE ENTRY RENAME_FILE
      ( P_LU : BYTE ; P_FN : CHAR8 ; P_EXTN : CHAR3 )

   ; BEGIN
      WITH SVC7_B
      DO BEGIN
         SVC7_CMD := SVC7_RENAME
         ; SVC7_LU := P_LU
         ; WITH SVC7_FD
         DO BEGIN

53      FN := P_FN
54      ;  EXTN := P_EXTN
55      ;  ACCT := 'P'
56      END
57      END
58      ;  SVC7 ( SVC7_B )
59      END
60
61      ;  PROCEDURE ENTRY ASSIGN_DEVICE
62      (  P_LU : BYTE ; P_VOLN : CHAR4 )
63
64      ;  BEGIN
65      WITH SVC7_B
66      DO BEGIN
67      ;  SVC7_CMD := SVC7_ASSIGN
68      ;  SVC7_LU := P_LU
69      ;  WITH SVC7_FD
70      DO BEGIN
71      ;  VOLN := P_VOLN
72      ;  FN := '
73      ;  EXTN := ' '
74      END
75      END
76      ;  SVC7 ( SVC7_B )
77      END
78
79      ;  PROCEDURE ENTRY DELETE_FILE
80      (  P_VOLN : CHAR4 ; P_FN : CHAR8 ; P_EXTN : CHAR3 )
81
82      ;  BEGIN
83      WITH SVC7_B
84      DO BEGIN
85      ;WITH SVC7_FD DO BEGIN
86      ;  SVC7_CMD := SVC7_DELETE
87      ;  WITH SVC7_FD DO BEGIN
88      ;  VOLN := P_VOLN
89      ;  FN := P_FN
90      ;  EXTN := P_EXTN
91      END
92      ;  SVC7_KEYS := 0000
93      END
94      ;  SVC7 ( SVC7_B )
95      END
96
97      ;  BEGIN END
98
99  (**********************************************************************************
100      * TERMINAL INPUT AND OUTPUT                                              *
101  **********************************************************************************)
102
103      ; TYPE TERMINAL_IO
104      = CLASS
105  
106      ; VAR SVC1_IN , SVC1_OUT : SVC1_BLOCK
107      ; C : CHAR
; OUT_LINE : LINE
; SVC1_OUTPUT_LENGTH : INTEGER

; PROCEDURE SVC1 ( SVC1_OUT : SVC1_BLOCK )
; EXTERN

; PROCEDURE SET_SVC1_INPUT

; BEGIN
WITH SVC1_IN DO BEGIN
  SVC1_FUNC := SVC1_READ + SVC1_WAIT + SVC1_IMAGE
; SVC1_LU := 0
; SVC1_STAT := 0
; SVC1_DEV_STAT := 0
; SVC1_BUFSTART := ADDRESS ( C )
; SVC1_BUFEND := ADDRESS ( C ) + 1 - 1
END
END

; PROCEDURE SET_SVC1_OUTPUT

; BEGIN
WITH SVC1_OUT DO BEGIN
  SVC1_FUNC := SVC1_WRITE + SVC1_WAIT + SVC1_IMAGE
; SVC1_LU := 0
; SVC1_STAT := 0
; SVC1_DEV_STAT := 0
; SVC1_BUFSTART := ADDRESS ( OUT_LINE )
END
; SVC1_OUTPUT_LENGTH := 0
END

; PROCEDURE SHIP_OUT

; BEGIN
  SVC1_OUT := SVC1_BUFEND + SVC1_BUFSTART + SVC1_OUTPUT_LENGTH - 1
; SVC1 ( SVC1_OUT )
; SVC1_OUTPUT_LENGTH := 0
END

; PROCEDURE ENTRY SHIP_OUT
; BEGIN IF SVC1_OUTPUT_LENGTH <> 0 THEN SHIP_OUT END

; PROCEDURE ENTRY INPUT ( VAR CURRENT_CHAR : CHAR )

; BEGIN
  SVC1 ( SVC1_IN )
; CURRENT_CHAR := CHR ( ORD ( C ) MOD 128 )
END
; PROCEDURE OUTPUT__ ( CC : CHAR )

; BEGIN
SVC1_OUTPUT_LENGTH := Succ ( SVC1_OUTPUT_LENGTH )
; OUT_LINE [ SVC1_OUTPUT_LENGTH ] := CC
; IF SVC1_OUTPUT_LENGTH = 79
  THEN SHIP_OUT_
END

; PROCEDURE ENTRY OUTPUT ( CC : CHAR )

; BEGIN OUTPUT__ ( CC ) END

; PROCEDURE ENTRY OUTPUT_LINE ( ALINE : LINE )

; VAR I : INTEGER

; BEGIN
  I := 1
  WHILE ALINE [ I ] <> '$'
    DO BEGIN
      OUTPUT__ ( ALINE [ I ] )
      ; I := I + 1
    END
  ; SHIP_OUT_
END

; BEGIN (* INIT STATEMENT OF TERMINAL IO *)
  SET_SVC1_OUTPUT
  ; SET_SVC1_INPUT
END

(**************************************************************************
* BUFFER STACK CLASS, IN MEMORY BUFFERS FOR TEXT DATA               *
**************************************************************************)

; TYPE BUFFER_STACK
  = CLASS

; VAR BUFFER_ARRAY : ARRAY [ 1 .. BUFFER_SIZE ] OF CHAR
; HEAD , TAIL , LENGTH , I : INTEGER

; PROCEDURE ENTRY PUSH ( C : CHAR ; VAR FULL : BOOLEAN )

; BEGIN
  BUFFER_ARRAY [ HEAD ] := C
  LENGTH := LENGTH + 1
  FULL := ( LENGTH = BUFFER_SIZE )
  HEAD := ( HEAD MOD BUFFER_SIZE ) + 1
END

; PROCEDURE ENTRY POP ( VAR C : CHAR ; VAR EMPTY : BOOLEAN )

; BEGIN
HEAD := (HEAD + BUFFER_SIZE - 2) MOD BUFFER_SIZE + 1
LENGTH := LENGTH - 1
EMPTY := (LENGTH = 0)
C := BUFFER_ARRAY[HEAD]
END

; PROCEDURE PUSH_ON_BOTTOM (C : CHAR)
BEGIN
BUFFER_ARRAY[TAIL] := C
LENGTH := LENGTH + 1
TAIL := (TAIL + BUFFER_SIZE - 2) MOD BUFFER_SIZE + 1
END

; PROCEDURE POP_OFF_BOTTOM
VAR C : CHAR;
VAR EMPTY : BOOLEAN;
BEGIN
TAIL := (TAIL MOD BUFFER_SIZE) + 1
LENGTH := LENGTH - 1
C := BUFFER_ARRAY[TAIL]
EMPTY := (LENGTH = 0)
END

; PROCEDURE ENTRY PUSH_PAGE (P : PAGE;
REVERSE : BOOLEAN)
BEGIN
IF REVERSE THEN FOR I := 1 TO 512 DO PUSH_ON_BOTTOM (P[I])
ELSE FOR I := 512 DOWNTO 1 DO PUSH_ON_BOTTOM (P[I])
END

; PROCEDURE ENTRY POP_PAGE
VAR P : PAGE;
VAR EMPTY : BOOLEAN;
BEGIN
IF REVERSE THEN FOR I := 512 DOWNTO 1
DO POP_OFF_BOTTOM (P[I], EMPTY)
ELSE BEGIN
I := 0
REPEAT I := I + 1
UNTIL (I = 512) OR EMPTY
END
END
BEGIN TAIL := BUFFER_SIZE; HEAD := 1; LENGTH := 0 END

*****************************************************************************
* FILE STACK CLASS, ON DISK FILE SPACE
*****************************************************************************

; TYPE FILE_STACK
= CLASS ( FILE_NUMBER : FILE )

; VAR PAGE_NUMBER : INTEGER

; PROCEDURE ENTRY PUSH ( VAR PAGE_IN : PAGE )

; BEGIN

    PUT ( FILE_NUMBER , PAGE_NUMBER , PAGE_IN )

    ; PAGE_NUMBER := PAGE_NUMBER + 1

END

; PROCEDURE ENTRY POP

( VAR PAGE_OUT : PAGE ; VAR FILE_EMPTY : BOOLEAN )

; BEGIN

    PAGE_NUMBER := PAGE_NUMBER - 1

    ; GET ( FILE_NUMBER , PAGE_NUMBER , PAGE_OUT )

    ; FILE_EMPTY := ( PAGE_NUMBER = 1 )

END

; BEGIN PAGE_NUMBER := 1 END

******************************************************************************

* LEFT AND RIGHT BUFFER STACK MANAGEMENT

******************************************************************************

; TYPE BUFFER_MANAGEMENT

    = CLASS ( IO : TERMINAL_IO )

(*
THE BUFFER_MANAGEMENT CLASS IS GIVEN ACCESS TO THE TERMINAL_IO CLASS TO THAT IT CAN PUT DOTS AND COLINS ON THE CONSOL SO THE USER KNOW THAT SOMETHING IS HAPPENING.
*)

; VAR LEFT , RIGHT : BUFFER_STACK

; LF , RF : FILE_STACK

; I , J : INTEGER

; P : PAGE

; RIGHT_EMPTY , RIGHT_FULL , LEFT_EMPTY , LEFT_FULL : BOOLEAN

; PROCEDURE ADD_LEFT ( C : CHAR )

; VAR NOTUSED : BOOLEAN

; BEGIN

    LEFT . PUSH ( C , LEFT_FULL )

    ; IF LEFT_FULL

    THEN FOR I := 1 TO NO_PAGES_IN_BUFFER DIV 2

    DO BEGIN

        LEFT . POP_PAGE ( P , FALSE , NOTUSED )

    ; LF . PUSH ( P )
END
END

; PROCEDURE POP_RIGHT ( VAR CURRENT_CHAR : CHAR )

; VAR EMPTY : BOOLEAN

; BEGIN
; RIGHT . POP ( CURRENT_CHAR , RIGHT_EMPTY )
; IF RIGHT_EMPTY AND ( CURRENT_CHAR <> EM )
THEN BEGIN
I := 0
; REPEAT I := I + 1
; RF . POP ( P , EMPTY )
; RIGHT . PUSH_PAGE ( P , TRUE )
UNTIL ( I = NO_PAGES_IN_BUFFER DIV 2 ) OR EMPTY
END
END

; PROCEDURE ENTRY MOVE_LEFT ( VAR CURRENT_CHAR : CHAR )

; BEGIN
ADD_LEFT ( CURRENT_CHAR )
; POP_RIGHT ( CURRENT_CHAR )
END

; PROCEDURE ADD_RIGHT ( C : CHAR )

; VAR NOTUSED : BOOLEAN

; BEGIN
; RIGHT . PUSH ( C , RIGHT_FULL )
; IF RIGHT_FULL
THEN FOR I := 1 TO NO_PAGES_IN_BUFFER DIV 2
DO BEGIN
RIGHT . POP_PAGE ( P , TRUE , NOTUSED )
; RF . PUSH ( P )
END
END

; PROCEDURE POP_LEFT ( VAR CURRENT_CHAR : CHAR )

; VAR EMPTY : BOOLEAN

; BEGIN
; LEFT . POP ( CURRENT_CHAR , LEFT_EMPTY )
; IF LEFT_EMPTY AND ( CURRENT_CHAR <> EM )
THEN BEGIN
I := 0
; REPEAT I := I + 1
; LF . POP ( P , EMPTY )
; LEFT . PUSH_PAGE ( P , FALSE )
UNTIL ( I = NO_PAGES_IN_BUFFER DIV 2 ) OR EMPTY
END
END

; PROCEDURE ENTRY MOVE_RIGHT ( VAR CURRENT_CHAR : CHAR )

; BEGIN
  ADD_RIGHT ( CURRENT_CHAR )
  POP_LEFT ( CURRENT_CHAR )
END

; PROCEDURE ENTRY ADD ( C : CHAR )

; BEGIN ADD_LEFT ( C ) END

; PROCEDURE ENTRY DELETE ( VAR CURRENT_CHAR : CHAR )

; BEGIN POP_RIGHT ( CURRENT_CHAR ) END

; PROCEDURE ENTRY FIND_NO_OF_PAGES
  ( FILE_NO : INTEGER ; VAR NO_OF_PAGES : INTEGER )

; VAR END_OF_FILE : BOOLEAN
; COUNT_OF_DOTS : INTEGER

; BEGIN
  NO_OF_PAGES := 0
  COUNT_OF_DOTS := 12
  END_OF_FILE := FALSE
  WHILE NOT ( END_OF_FILE )
    DO BEGIN
      NO_OF_PAGES := NO_OF_PAGES + 1
      GET ( FILE_NO , NO_OF_PAGES , P )
      IO . OUTPUT ( ' , ' )
      IO . SHIP_OUT
      COUNT_OF_DOTS := COUNT_OF_DOTS + 1
      IF COUNT_OF_DOTS > 80
        THEN BEGIN
          COUNT_OF_DOTS := 1
          IO . OUTPUT ( NL )
          IO . OUTPUT ( CR )
          IO . SHIP_OUT
          END
      FOR I := 1 TO PAGELENGTH
        DO BEGIN
          IF ( P [ I ] = EM )
            THEN END_OF_FILE := TRUE
        END
    END
END

; PROCEDURE ENTRY INIT_RIGHT_STACK
  ( FILE_NO , LAST_PAGE_NO : INTEGER )

; VAR COUNT_OF_COLSINS : INTEGER
BEGIN
  COUNT_OF_COLINS := 1
  IO. OUTPUT ( NL )
  IO. OUTPUT ( CR )
  IO. SHIP_OUT
  FOR I := LAST_PAGE_NO DOWNTO 1
    DO BEGIN
      GET ( FILE_NO, I, P )
      COUNT_OF_COLINS := COUNT_OF_COLINS + 1
      IO. OUTPUT ( ':' )
      IO. SHIP_OUT
      IF COUNT_OF_COLINS > 80
        THEN BEGIN
          IO. OUTPUT ( NL )
          IO. OUTPUT ( CR )
          IO. SHIP_OUT
          COUNT_OF_COLINS := 1
        END
      RF. PUSH ( P )
    END
  END
BEGIN
PROCEDURE ENTRY FILL_LEFT_STACK
END
BEGIN
  VAR EMPTY : BOOLEAN
  BEGIN
    BEGIN
      REPEAT
        LEFT. POP_PAGE ( P, FALSE, EMPTY )
        LF. PUSH ( P )
      UNTIL EMPTY
    END
BEGIN
  INIT_LEFT, RIGHT, LF ( 3 ), RF ( 2 )
  ADD_RIGHT ( NL )
END

(* ------------------------- *)
* COPY BUFFER STACK MANAGEMENT
* ------------------------- *)
BEGIN
  TYPE COPY_BUFFER_MANAGEMENT
  = CLASS
    VAR COPYB : BUFFER_STACK
    COPYF : FILE_STACK
    I : INTEGER
    EMPTY, BUFFER_EMPTY, BUFFER_FULL, NOTUSED : BOOLEAN
    P : PAGE
    BEGIN
PROCEDURE ENTRY COPY_PUSH ( C : CHAR )
BEGIN

COPYB . PUSH ( C , BUFFER_FULL )
; IF BUFFER_FULL
THEN FOR I := 1 TO NO_PAGES_IN_BUFFER DIV 2
DO BEGIN
COPYB . POP_PAGE ( P , FALSE , NOTUSED )
; COPYF . PUSH ( P )
END
END

; PROCEDURE ENTRY COPY_POP ( VAR C : CHAR )

; BEGIN
COPYB . POP ( C , BUFFER_EMPTY )
; IF BUFFER_EMPTY AND ( C <> EM )
THEN BEGIN
I := 0
; REPEAT I := I + 1
; COPYF . POP ( P , EMPTY )
; COPYB . PUSH_PAGE ( P , FALSE )
UNTIL ( I = NO_PAGES_IN_BUFFER DIV 2 ) OR EMPTY
END
END
END

; BEGIN
INIT COPYB , COPYF ( 4 )
; COPYB . PUSH ( EM , NOTUSED )
; COPYB . PUSH ( EM , NOTUSED )
END

"%ENDMACRO CLASS_TYPES"
Sedit main program, file name SEDIT1.PAS

(*---------------------------------------------------------------------------*
 * SSSS       EEEEEE     DDDDDDD    IIIIIII    TTTTTTTT    *
 * SSSSS      EEEEEE     DDDDDDD    IIIIIII    TTTTTTTT    *
 * SSSS       EEEEEE     DDDDDDD    IIIIIII    TTTTTTTT    *
 * SSS        EEEE     DD        DD        II    T    TT    T    *
 * SSS        EEE     DD        DD        II    TT    *
 * SSSS       EEEEEE     DD        DD        II    TT    *
 * SSS        EEEEEE     DD        DD        II    TT    *
 * SS        SSSS       EEEEEE     DD        DD        II    TT    *
 * SS        SSSS       EEEEEE     DD        DD        II    TT    *
 * SSSSS      EEEEEE     DDDDDDD    IIIIIII    TT    *
 * SSSSS      EEEEEE     DDDDDDD    IIIIIII    TTTT    *
 * *
 * *
 * *
 * *
 * SEDIT VERSION 1
 * MADE BY THEODORE JOHN SOCOLOFSKY
 * AS A PROJECT FOR CIS
 * IN THE SUMMER OF 1981
 #---------------------------------------------------------------------------)
 "INCLUDE SVC_TYPES"
 (**************************************************************************
 * OS/32MT3 SVC INTERFACE ROUTINE TYPES
 *
 * #-----------------------------------------------------------------------)
 "INCLUDE FULL_PREFIX"
 (**************************************************************************
 # PREFIX #
 #-----------------------------------------------------------------------)
 PROGRAM SEDIT
 ( PARAM : LINE ) (**************************************************************************
 ; TYPE FILE = 1 .. 5

 ; CONST ETX = '(:3:)' ; BEL = '(:7:)' ; BS = '(:8:)' ; HT = '(:9:)' ; TAB = '(:9:)' ; VT = '(:11:)' ; CR = '(:13:)' ; SUB = '(:26:)' ; ESC = '(:27:)' ; RS = '(:30:)' ; DEL = '(:127:)' ; PAD = '(:127:)' ; SET_OF_DIGITS = [ '0' .. '9' ] ; SET_OF_UPPERCASE_LETTERS = [ 'A' .. 'Z' ] ; SET_OF LOWERCASE_LETTERS = [ 'a' .. 'z' ] ;
; SET_OF_PRINTING_SPECIAL_CHARACTERS
= [ ' ' .. '/ ' , ':' .. ' ' , '[' .. ' ' ]
; SET_OF_VALID_INPUT_CHARACTERS
= SET_OF_DIGITS
  + SET_OF_UPPERCASE_LETTERS
  + SET_OF_LOWERCASE_LETTERS
  + SET_OF_PRINTING_SPECIAL_CHARACTERS
; SET_OF_INVALID_INPUT_CHARACTERS
= ['(o:) ' .. '(31:) ' , '(127:) ' ]
; NO_PAGES_IN_BUFFER = 8
; BUFFER_SIZE = NO_PAGES_IN_BUFFER * PAGELength

"%INCLUDE CLASS_TYPES"
(* MAIN **********************************************)

; VAR IO : TERMINAL_IO
; B_M : BUFFER_MANAGEMENT
; S7 : SVC7_CALLS
; C_B_M : COPY_BUFFER_MANAGEMENT
; SHOW_LINE_NUMBERS , SECOND_EDIT_PROMPT_ON : BOOLEAN
; CURSOR_ROW , CURSOR_COL : INTEGER
; SAVED_CURSOR_ROW , SAVED_CURSOR_COL : INTEGER
; DIRECTION_CHAR : CHAR

; PROCEDURE CLEAR_SCREEN

; BEGIN
  IO . OUTPUT ( SUB )
  ; IO . OUTPUT ( PAD )
  ; IO . OUTPUT ( PAD )
  ; IO . SHIP_OUT
  ; CURSOR_ROW := 1
  ; CURSOR_COL := 1
END

; PROCEDURE HOME_CURSOR

; BEGIN
  IO . OUTPUT ( RS )
  ; IO . OUTPUT ( PAD )
  ; IO . OUTPUT ( PAD )
  ; IO . SHIP_OUT
  ; CURSOR_ROW := 1
  ; CURSOR_COL := 1
END

; PROCEDURE MOVE_CURSOR ( ROW , COL : INTEGER )

; VAR TROW , TCOL : INTEGER

; BEGIN
  TROW := ROW
  ; TCOL := COL
  ; IO . OUTPUT ( ESC )
107 ; IO . OUTPUT ( ' = ' )
108 ; IF ROW < 1
109     THEN TROW := 1
110 ; IF ROW > 24
111     THEN TROW := 24
112 ; IF COL < 1
113     THEN TCOL := 1
114 ; IF COL > 80
115     THEN TCOL := 80
116 ; IO . OUTPUT ( CHR ( TROW + 31 ) )
117 ; IO . OUTPUT ( CHR ( TCOL + 31 ) )
118 ; CURSOR_ROW := TROW
119 ; CURSOR_COL := TCOL
120 ; IO . SHIP_OUT
121 END

122 ; PROCEDURE SAVE_CURSOR_POSITION
123
124 ; BEGIN
125     SAVED_CURSOR_ROW := CURSOR_ROW
126 ;     SAVED_CURSOR_COL := CURSOR_COL
127 END

128 ; PROCEDURE UNSAVE_CURSOR_POSITION
129
130 ; BEGIN
131     CURSOR_ROW := SAVED_CURSOR_ROW
132 ;     CURSOR_COL := SAVED_CURSOR_COL
133 END

134 ; PROCEDURE MOVE_TO_END_OF_CURRENT_LINE
135 ( VAR CURRENT_CHAR : CHAR ; VAR LENGTH : INTEGER )
136
137 ; BEGIN
138     LENGTH := 0
139 ;     WHILE CURRENT_CHAR <> NL
140     DO BEGIN
141         B_M . MOVE_LEFT ( CURRENT_CHAR )
142     ; LENGTH := LENGTH + 1
143 END
144 END

145 ; PROCEDURE MOVE_TO_END_OF_PREVIOUS_LINE
146 ( VAR CURRENT_CHAR : CHAR ; VAR LENGTH : INTEGER )
147
148 ; BEGIN
149     LENGTH := 0
150 ;     WHILE CURRENT_CHAR <> NL
151     DO BEGIN
152         B_M . MOVE_RIGHT ( CURRENT_CHAR )
153     ; LENGTH := LENGTH + 1
154 END
155 END

156
; PROCEDURE OUTPUT_REST_OF_LINE
  ( VAR CURRENT_CHAR : CHAR ; VAR LENGTH : INTEGER )

  ; VAR I , J : INTEGER

  ; BEGIN
    IF NOT SHOW_LINE_NUMBERS
    THEN BEGIN
      MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , I )
      ; IF I > 9
      THEN LENGTH := I
      ELSE LENGTH := 9
      ; J := 0
      WHILE ( J < LENGTH ) AND ( CURRENT_CHAR <> EM )
      DO BEGIN
        J := J + 1
        ; B_M . MOVE_LEFT ( CURRENT_CHAR )
      END
      ; IF CURRENT_CHAR = EM
      THEN B_M . MOVE_RIGHT ( CURRENT_CHAR )
    END

    ; LENGTH := 0
    WHILE CURRENT_CHAR <> NL
    DO BEGIN
      IF CURSOR_COL + LENGTH <= 80
      THEN IO . OUTPUT ( CURRENT_CHAR )
      ; B_M . MOVE_LEFT ( CURRENT_CHAR )
      ; LENGTH := LENGTH + 1
    END

    ; IF CURSOR_COL + LENGTH <= 80
    THEN CURSOR_COL := CURSOR_COL + LENGTH
    ELSE BEGIN
      CURSOR_COL := 80
      ; IF CURSOR_COL + LENGTH > 80
      THEN BEGIN
        MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )
        ; IO . OUTPUT ( 'I' )
      END
      END

    ; IF LENGTH <> 0
    THEN IO . SHIP_OUT
  END

; PROCEDURE MOVE_TO_BEGINNING_OF_TEXT ( VAR CURRENT_CHAR : CHAR )

  ; VAR I : INTEGER

  ; BEGIN
    REPEAT
      B_M . MOVE_RIGHT ( CURRENT_CHAR )
    UNTIL CURRENT_CHAR = EM
    ; FOR I := 1 TO 10 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
  END
; PROCEDURE MOVE_TO_END_OF_TEXT ( VAR CURRENT_CHAR : CHAR )
BEGIN
REPEAT
B_M . MOVE_LEFT ( CURRENT_CHAR )
UNTIL CURRENT_CHAR = EM
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
END

; PROCEDURE OUTPUT_REST_OF_SCREEN
( VAR CURRENT_CHAR : CHAR
; VAR LENGTH, NUMBER_OF_LINES : INTEGER
)
BEGIN
VAR TLNGTH : INTEGER

LENGTH := 0
; NUMBER_OF_LINES := 0
WHILE ( CURRENT_CHAR <> EM ) AND ( CURSOR_ROW <= 24 )
DO BEGIN
MOVE_CURSOR ( CURSOR_ROW, CURSOR_COL )
; OUTPUT_REST_OF_LINE ( CURRENT_CHAR, TLNGTH )
LENGTH := LENGTH + TLNGTH + 1
NUMBER_OF_LINES := NUMBER_OF_LINES + 1
; B_M . MOVE_LEFT ( CURRENT_CHAR )
CURSOR_COL := 1
CURSOR_ROW := CURSOR_ROW + 1
END
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
LENGTH := LENGTH - 1
END

; PROCEDURE VERIFY ( VAR CURRENT_CHAR : CHAR )
BEGIN
VAR I, LENGTH, TLNGTH, NUMBER_OF_LINES : INTEGER

NUMBER_OF_LINES := 0
LENGTH := 0
REPEAT MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR, TLNGTH )
LENGTH := LENGTH - TLNGTH - 1
; B_M . MOVE_LEFT ( CURRENT_CHAR )
NUMBER_OF_LINES := NUMBER_OF_LINES + 1
UNTIL ( NUMBER_OF_LINES = 12 ) OR ( CURRENT_CHAR = EM )
NUMBER_OF_LINES := 1
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
LENGTH := LENGTH + 2
REPEAT MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR, TLNGTH )
LENGTH := LENGTH + TLNGTH + 1
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
NUMBER_OF_LINES := NUMBER_OF_LINES + 1
UNTIL ( NUMBER_OF_LINES = 24 ) OR ( CURRENT_CHAR = EM )
}
; B_M . MOVE_LEFT ( CURRENT_CHAR )
; B_M . MOVE_LEFT ( CURRENT_CHAR )
; LENGTH := LENGTH - 2
; CLEAR_SCREEN
; MOVE_CURSOR ( 2, 1 )
; OUTPUT_REST_OF_SCREEN
( CURRENT_CHAR, LENGTH, NUMBER_OF_LINES )
; FOR I := 1 TO LENGTH - LENGTH
DO BEGIN
  B_M . MOVE_RIGHT ( CURRENT_CHAR )
  IF CURRENT_CHAR = NL
  THEN NUMBER_OF_LINES := NUMBER_OF_LINES - 1
END
; LENGTH := 0
; IF CURRENT_CHAR = NL
THEN BEGIN B_M . MOVE_RIGHT ( CURRENT_CHAR ); LENGTH := 1 END
; MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR, LENGTH )
; LENGTH := LENGTH + LENGTH
; FOR I := 1 TO LENGTH DO B_M . MOVE_LEFT ( CURRENT_CHAR )
; IF LENGTH > 80
THEN LENGTH := 80
; NUMBER_OF_LINES := NUMBER_OF_LINES + 1
; MOVE_CURSOR ( NUMBER_OF_LINES, LENGTH )
END

; PROCEDURE MOVE_RIGHT_1_LINE
( var CURRENT_CHAR : CHAR ; var J : INTEGER )

; var L : INTEGER
; end_was_found : BOOLEAN

; begin
L := 0
; IF CURRENT_CHAR = NL
THEN BEGIN
  B_M . MOVE_RIGHT ( CURRENT_CHAR )
  L := 1
; IF CURRENT_CHAR = EM
  THEN BEGIN B_M . MOVE_LEFT ( CURRENT_CHAR ); L := 0 END
END
; MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR, J )
; L := L + J
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
; IF CURRENT_CHAR = EM
THEN BEGIN
  B_M . MOVE_LEFT ( CURRENT_CHAR )
; end_was_found := TRUE
END
ELSE end_was_found := FALSE
; MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR, J )
; J := 1
; B_M . MOVE_LEFT ( CURRENT_CHAR )
; WHILE ( J < L ) AND ( CURRENT_CHAR <> NL )
DO BEGIN
J := J + 1
B_M . MOVE_LEFT ( CURRENT_CHAR )
END
IF END_WAS_FOUND THEN J := 0 END

; PROCEDURE MOVE_LEFT_1_LINE
( VAR CURRENT_CHAR : CHAR ; VAR I : INTEGER )

; VAR L , J : INTEGER
BEGIN
L := 0
IF CURRENT_CHAR = NL THEN BEGIN
B_M . MOVE_RIGHT ( CURRENT_CHAR )
L := 1
IF CURRENT_CHAR = EM THEN BEGIN B_M . MOVE_LEFT ( CURRENT_CHAR ) ; L := 0 END
END
MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , I )
L := L + I
B_M . MOVE_LEFT ( CURRENT_CHAR )
MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR , I )
B_M . MOVE_LEFT ( CURRENT_CHAR )
I := 1
IF CURRENT_CHAR = EM THEN BEGIN
B_M . MOVE_RIGHT ( CURRENT_CHAR )
B_M . MOVE_RIGHT ( CURRENT_CHAR )
MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , I )
FOR J := 1 TO L DO B_M . MOVE_LEFT ( CURRENT_CHAR )
I := 0
END
ELSE WHILE ( CURRENT_CHAR <> NL ) AND ( I < L )
DO BEGIN
I := I + 1
; B_M . MOVE_LEFT ( CURRENT_CHAR )
END
END

; PROCEDURE EDIT_PROMPT ( DIRECTION_CHAR : CHAR )

; VAR SROW , SCOL : INTEGER
BEGIN
SROW := CURSOR_ROW
SCOL := CURSOR_COL
MOVE_CURSOR ( 1 , 1 )
IO . OUTPUT ( DIRECTION_CHAR )
IO . OUTPUT_LINE
( 'EDIT: I(nsert D(elete P(age J(ump$')
; IO . OUTPUT_LINE
377  ( 'X(change P(IX V(verify
378    ; MOVE_CURSOR ( SROW, SCOL )
379  END
380
381  ; PROCEDURE UP_ARROW ( VAR CURRENT_CHAR : CHAR ; VAR TOF : BOOLEAN )
382
383  ; VAR I : INTEGER
384
385  ; BEGIN
386      CURSOR_ROW := CURSOR_ROW - 1
387      ; MOVE_RIGHT_1_LINE ( CURRENT_CHAR, I )
388      ; IF CURRENT_ROW = 1
389      THEN BEGIN
390          IF I <> 0
391              THEN BEGIN
392                  CURSOR_COL := I
393                  ; VERIFY ( CURRENT_CHAR )
394                  ; EDIT_PROMPT ( DIRECTION_CHAR )
395              END
396      ELSE MOVE_CURSOR ( 2, CURSOR_COL )
397  END
398  ELSE IF I <> 0
399      THEN MOVE_CURSOR ( CURSOR_ROW, I )
400      ELSE MOVE_CURSOR ( CURSOR_ROW, CURSOR_COL )
401
402      ; IF I = 0
403      THEN TOF := TRUE
404      ELSE TOF := FALSE
405  END
406
407  ; PROCEDURE DOWN_ARROW
408    ( VAR CURRENT_CHAR : CHAR ; VAR BOF : BOOLEAN )
409
410  ; VAR I, J : INTEGER
411
412  ; BEGIN
413      MOVE_LEFT_1_LINE ( CURRENT_CHAR, J )
414      ; IF J <> 0
415      THEN BEGIN
416          CURSOR_ROW := CURSOR_ROW + 1
417          ; IF CURRENT_ROW = 25
418      THEN BEGIN
419          FOR I := 1 TO J - 1 DO B_M. MOVE_RIGHT ( CURRENT_CHAR )
420          ; IO OUTPUT ( NL )
421          ; MOVE_CURSOR ( 24, 1 )
422          ; OUTPUT_REST_OF_LINE ( CURRENT_CHAR, I )
423          ; EDIT_PROMPT ( DIRECTION_CHAR )
424          ; FOR I := 1 TO J - ( J - 1 )
425          DO B_M. MOVE_RIGHT ( CURRENT_CHAR )
426          ; CURSOR_ROW := 24
427          END
428          MOVE_CURSOR ( CURSOR_ROW, J )
429          END
430          ; IF J = 0
431          THEN BOF := TRUE
ELSE BOF := FALSE
END

; PROCEDURE LEFT_ARROW
( VAR CURRENT_CHAR : CHAR ; VAR TOF : BOOLEAN )

; VAR J , I : INTEGER

BEGIN

TOF := FALSE
IF CURSOR_COL = 80 THEN BEGIN
IF CURRENT_CHAR = NL
THEN BEGIN B_M . MOVE_RIGHT ( CURRENT_CHAR ) ; I := 1 END
ELSE I := 0
MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , J )
J := J + I
FOR I := 1 TO J DO B_M . MOVE_LEFT ( CURRENT_CHAR )
IF ( J > 88 ) OR ( ( J > 80 ) AND SHOW_LINE_NUMBERS )
THEN CURSOR_COL := CURSOR_COL + 1
END
CURSOR_COL := CURSOR_COL - 1
IF ( CURSOR_COL < 1 )
OR ( SHOW_LINE_NUMBERS AND ( CURSOR_COL < 9 ) )
THEN BEGIN
IF CURRENT_CHAR = NL
THEN BEGIN B_M . MOVE_RIGHT ( CURRENT_CHAR ) ; I := 1 END
ELSE I := 0
MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , J )
J := J + I
B_M . MOVE_RIGHT ( CURRENT_CHAR )
IF CURRENT_CHAR <> EM
THEN BEGIN
MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , J )
B_M . MOVE_LEFT ( CURRENT_CHAR )
MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR , J )
CURSOR_ROW := CURSOR_ROW - 1
END
ELSE BEGIN
FOR I := 1 TO J + 1 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
IF SHOW_LINE_NUMBERS
THEN J := 8
ELSE J := 0
TOF := TRUE
END
IF SHOW_LINE_NUMBERS
THEN CURSOR_COL := J + 1
ELSE CURSOR_COL := J - 7
END
ELSE B_M . MOVE_RIGHT ( CURRENT_CHAR )
IF CURSOR_COL > 80
THEN CURSOR_COL := 80
IF CURSOR_ROW = 1
THEN BEGIN
PROCEDURE RIGHT_ARROW
  ( VAR CURRENT_CHAR : CHAR ; VAR BOF : BOOLEAN )
  ; VAR I , J : INTEGER

BEGIN
  BOF := FALSE
  CURSOR_COL := CURSOR_COL + 1
  IF CURRENT_CHAR = NL THEN BEGIN
    IF SHOW_LINE_NUMBERS THEN BEGIN
      B_M . MOVE_LEFT ( CURRENT_CHAR )
      IF CURRENT_CHAR <> EM THEN BEGIN
        CURSOR_ROW := CURSOR_ROW + 1
        CURSOR_COL := 9
        IF CURSOR_ROW = 25 THEN BEGIN
          IO . OUTPUT ( NL )
          MOVE_CURSOR ( 24 , 1 )
          OUTPUT_REST_OF_LINE ( CURRENT_CHAR , I )
          FOR J := 1 TO I - 8 DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
          CURSOR_COL := 9
          EDIT_PROMPT ( DIRECTION_CHAR )
          END
        ELSE FOR J := 1 TO 8 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
      END
    END
    ELSE BEGIN
      CURSOR_COL := CURSOR_COL - 1
      B_M . MOVE_RIGHT ( CURRENT_CHAR )
      BOF := TRUE
    END
  END
END
ELSE BEGIN
  B_M . MOVE_LEFT ( CURRENT_CHAR )
  IF CURRENT_CHAR <> EM THEN BEGIN
    CURSOR_ROW := CURSOR_ROW + 1
    CURSOR_COL := 1
    IF CURSOR_ROW = 25 THEN BEGIN
      IO . OUTPUT ( NL )
      MOVE_CURSOR ( 24 , 1 )
      OUTPUT_REST_OF_LINE ( CURRENT_CHAR , I )
      FOR J := 1 TO I DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
      CURSOR_COL := 1
    END
  END
END
539 ; EDIT_PROMPT ( DIRECTION_CHAR )
540 END
541 END
542 ELSE BEGIN
543 CURSOR_COL := CURSOR_COL - 1
544 ; B_M . MOVE_RIGHT ( CURRENT_CHAR )
545 END
546 END
547 END
548 ELSE B_M . MOVE_LEFT ( CURRENT_CHAR )
549 ; IF CURSOR_COL = 81
550 THEN CURSOR_COL := 80
551 ; MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )
552 END
553
554 ; PROCEDURE RETURN ( VAR CURRENT_CHAR : CHAR )
555
556 ; VAR I , J , L : INTEGER
557 ; NEW_LINE : BOOLEAN
558
559 ; BEGIN
560 NEW_LINE := FALSE
561 ; MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR , L )
562 ; IF SHOW_LINE_NUMBERS
563 THEN BEGIN
564 B_M . MOVE_LEFT ( CURRENT_CHAR )
565 ; IF CURRENT_CHAR <> EM
566 THEN BEGIN
567 CURSOR_ROW := CURSOR_ROW + 1
568 ; CURSOR_COL := 9
569 ; IF CURSOR_ROW = 25
570 THEN BEGIN
571 IO . OUTPUT ( NL )
572 ; MOVE_CURSOR ( 24 , 1 )
573 ; OUTPUT_REST_OF_LINE ( CURRENT_CHAR , I )
574 ; FOR J := 1 TO I - 8 DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
575 ; CURSOR_COL := 9
576 ; NEW_LINE := TRUE
577 END
578 ELSE FOR J := 1 TO 8 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
579 END
580 ELSE BEGIN
581 CURSOR_COL := CURSOR_COL + L
582 ; B_M . MOVE_RIGHT ( CURRENT_CHAR )
583 END
584 END
585 ELSE BEGIN
586 B_M . MOVE_LEFT ( CURRENT_CHAR )
587 ; IF CURRENT_CHAR <> EM
588 THEN BEGIN
589 CURSOR_ROW := CURSOR_ROW + 1
590 ; CURSOR_COL := 1
591 ; IF CURSOR_ROW = 25
592 THEN BEGIN
IO . OUTPUT ( NL )
; MOVE_CURSOR ( 24 , 1 )
; OUTPUT_REST_OF_LINE ( CURRENT_CHAR , I )
; FOR J := 1 TO I DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
; CURSOR_COL := 1
; NEW_LINE := TRUE
END

END
ELSE BEGIN
CURSOR_COL := CURSOR_COL + L
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
END

END

; WHILE ( CURRENT_CHAR = ' ' ) AND ( CURSOR_COL < 80 )
DO BEGIN
CURSOR_COL := CURSOR_COL + 1
; B_M . MOVE_LEFT ( CURRENT_CHAR )
END

; IF NEW_LINE
THEN EDIT_PROMPT ( DIRECTION_CHAR )
ELSE MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )

END

; PROCEDURE MOVE_PAGE
( VAR CURRENT_CHAR : CHAR ; DIRECTION_CHAR : CHAR )

; VAR I , J : INTEGER

; BEGIN
IF DIRECTION_CHAR = '<'
THEN BEGIN
MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , J )
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
; IF CURRENT_CHAR <> EM
THEN BEGIN
FOR I := 1 TO J + 1 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
END
END

; IF DIRECTION_CHAR = '>'
THEN BEGIN
MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR , J )
; B_M . MOVE_LEFT ( CURRENT_CHAR )
; IF CURRENT_CHAR <> EM
THEN BEGIN
FOR I := 1 TO J + 1 DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
END

; VERIFY ( CURRENT_CHAR )
; EDIT_PROMPT ( DIRECTION_CHAR )
END
ELSE FOR I := 1 TO J + 1 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
END
647 END
648 ELSE FOR I := 1 TO J + 1
649 DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
650 END
651 END

; PROCEDURE ADD_LINE_NUMBERS ( VAR CURRENT_CHAR : CHAR )
653 ; VAR LINE_NUMBER , NUMBER , J , I : INTEGER
654 ; INTEGER_STRING : ARRAY [ 1 .. 4 ] OF CHAR
658 ; BEGIN
659 "CHECK FOR BUFFER CONTAINING EM NL EM"
660 IF IT DOES ADD A NL CHARACTER, THEN MOVE BACK TO LEFT EM."
661 ; B_M . MOVE_LEFT ( CURRENT_CHAR )
662 ; B_M . MOVE_LEFT ( CURRENT_CHAR )
663 ; IF CURRENT_CHAR = EM
664 THEN BEGIN
665 B_M . ADD ( NL )
666 B_M . MOVE_RIGHT ( CURRENT_CHAR )
667 END
668 ; B_M . MOVE_RIGHT ( CURRENT_CHAR )
669 ; B_M . MOVE_RIGHT ( CURRENT_CHAR )
670 ; IO . OUTPUT ( NL )
671 ; IO . OUTPUT ( CR )
672 ; IO . SHIP_OUT
673 ; LINE_NUMBER := 0
674 ; REPEAT WHILE CURRENT_CHAR <> NL
675 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
676 ; B_M . MOVE_LEFT ( CURRENT_CHAR )
677 ; IF CURRENT_CHAR <> EM
678 THEN BEGIN
679 LINE_NUMBER := ( LINE_NUMBER + 1 ) MOD 10000
680 ; NUMBER := LINE_NUMBER
681 ; IF NUMBER MOD 10 = 0
682 THEN BEGIN
683 IO . OUTPUT ( '<' )
684 ; IO . SHIP_OUT
685 ; IF NUMBER MOD 800 = 0
686 THEN BEGIN
687 IO . OUTPUT ( NL )
688 ; IO . OUTPUT ( CR )
689 ; IO . SHIP_OUT
690 END
691 END
692 ; FOR I := 4 DOWNTO 1
693 DO BEGIN
694 INTEGER_STRING [ I ] := CHR ( NUMBER MOD 10 + 48 )
695 ; NUMBER := NUMBER DIV 10
696 END
697 ; I := 1
698 ; WHILE ( INTEGER_STRING [ I ] = '0' ) AND ( I < 4 )
699 DO BEGIN
700 B_M . ADD (' ' )
; I := I + 1
END

; FOR J := I TO 4 DO B_M . ADD ( INTEGER_STRING [ J ] )
; FOR J := 1 TO 4 DO B_M . ADD ( ' ' )
END

UNTIL CURRENT_CHAR = EM

; B_M . MOVE_RIGHT ( CURRENT_CHAR ) "MOVE BACK TO BEGINNING."

; LINE_NUMBER := 1
; IO . OUTPUT ( NL )
; IO . OUTPUT ( CR )
; IO . SHIP_OUT
REPEAT B_M . MOVE_RIGHT ( CURRENT_CHAR )

; IF CURRENT_CHAR = NL
THEN BEGIN
LINE_NUMBER := LINE_NUMBER + 1
THEN BEGIN
IO . OUTPUT ( '>' )
IO . SHIP_OUT
IF LINE_NUMBER MOD 800 = 0 THEN BEGIN
IO . OUTPUT ( NL )
IO . OUTPUT ( CR )
IO . SHIP_OUT
END
END
UNTIL CURRENT_CHAR = EM

; FOR I := 1 TO 10 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
END

PROCEDURE INSERT_PROMPT

; VAR SRCROW, SCL : INTEGER
BEGIN
SRCROW := CURSOR_ROW
SCL := CURSOR_COL
MOVE_CURSOR ( 1, 1 )
IO . OUTPUT_LINE
( 'INSERT: [left arrow] a char, [del] a li$' )
IO . OUTPUT_LINE
( 'ne, [esc] to abort, [ctrl<c>] to exit $' )
MOVE_CURSOR ( SRCROW, SCL )
END

PROCEDURE OUTPUT_FRONT_OF_LINE ( VAR CURRENT_CHAR : CHAR )
BEGIN
IF CURRENT_CHAR = NL THEN BEGIN B_M . MOVE_RIGHT ( CURRENT_CHAR ) ; I := 1 END

VAR I, L : INTEGER
BEGIN
IF CURRENT_CHAR = NL THEN BEGIN B_M . MOVE_RIGHT ( CURRENT_CHAR ) ; I := 1 END
DELSE I := 0
756 ; MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , L )
757 ; L := L + I
758 ; B_M . MOVE_LEFT ( CURRENT_CHAR )
759 ; MOVE_CURSOR ( CURSOR_ROW , 1 )
760 ; FOR I := 1 TO L - 1
761 DO BEGIN
762 IO . OUTPUT ( CURRENT_CHAR )
763 ; B_M . MOVE_LEFT ( CURRENT_CHAR )
764 END
765 ; CURSOR_COL := L
766 ; IO . SHIP_OUT
767 END
768
769 ; PROCEDURE SPREAD_END_OF_LINE ( VAR CURRENT_CHAR : CHAR )
770
771 ; VAR I , J , L , NUMBER_OF_SPACES : INTEGER
772
773 ; BEGIN
774 ; MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR , J )
775 ; FOR I := 1 TO J DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
776 ; NUMBER_OF_SPACES := 80 - J - CURSOR_COL + 1
777 ; IF NUMBER_OF_SPACES > 1
778 THEN BEGIN
779 ; SAVE_CURSOR_POSITION
780 ; IF CURRENT_CHAR <> NL
781 THEN BEGIN
782 FOR I := 1 TO NUMBER_OF_SPACES DO IO . OUTPUT ( ' ' )
783 ; FOR I := 1 TO J
784 DO BEGIN
785 ; IO . OUTPUT ( CURRENT_CHAR )
786 ; B_M . MOVE_LEFT ( CURRENT_CHAR )
787 END
788 ; FOR I := 1 TO J DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
789 END
790 ; UNSAVE_CURSOR_POSITION
791 ; MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )
792 END
793
794 ; PROCEDURE SPREAD_CURRENT_LINE
795 ( VAR CURRENT_CHAR : CHAR ; VAR NUMBER_OF_SPACES : INTEGER )
796
797 ; VAR I , L , J : INTEGER
798
799 ; BEGIN
800 ; MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR , J )
801 ; FOR I := 1 TO J DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
802 ; NUMBER_OF_SPACES := 80 - J - CURSOR_COL + 1
803 ; IF NUMBER_OF_SPACES > 1
804 THEN BEGIN
805 ; SAVE_CURSOR_POSITION
806 ; OUTPUT_FRONT_OF_LINE ( CURRENT_CHAR )
807 ; IF CURRENT_CHAR <> NL
THEN BEGIN
FOR I := 1 TO NUMBER_OF_SPACES DO IO. OUTPUT ( ' ' )
; FOR I := 1 TO J
DO BEGIN
IO. OUTPUT ( CURRENT_CHAR )
; B_M. MOVE_LEFT ( CURRENT_CHAR )
END
; FOR I := 1 TO J DO B_M. MOVE_RIGHT ( CURRENT_CHAR )
END
; UNSAVE_CURSOR_POSITION
; MOVE_CURSOR ( CURSOR_ROW, CURSOR_COL )
END

; PROCEDURE SPLIT_CURRENT_LINE ( VAR CURRENT_CHAR : CHAR )

; VAR I, J, L, K : INTEGER

; BEGIN
OUTPUT_FRONT_OF_LINE ( CURRENT_CHAR )
; SAVE_CURSOR_POSITION
; FOR I := CURSOR_COL TO 80 DO IO. OUTPUT ( ' ' )
; WHILE CURSOR_ROW < 23
DO BEGIN
CURSOR_ROW := CURSOR_ROW + 1
; MOVE_CURSOR ( CURSOR_ROW, 1 )
; FOR I := 1 TO 80 DO IO. OUTPUT ( ' ' )
;
END
; MOVE_CURSOR ( CURSOR_ROW + 1, 1 )
; MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR, J )
; FOR I := 1 TO J DO B_M. MOVE_RIGHT ( CURRENT_CHAR )
IF J > 81 THEN K := 81 ELSE K := J
; FOR I := 1 TO 80 - K DO IO. OUTPUT ( ' ' )
; FOR I := 1 TO K
DO BEGIN
IO. OUTPUT ( CURRENT_CHAR )
; B_M. MOVE_LEFT ( CURRENT_CHAR )
END
; FOR I := 1 TO K DO B_M. MOVE_RIGHT ( CURRENT_CHAR )
; UNSAVE_CURSOR_POSITION
; MOVE_CURSOR ( CURSOR_ROW, CURSOR_COL )
END

; PROCEDURE INSERT_A_NL_AND_LINE_NO ( VAR CURRENT_CHAR : CHAR )

; VAR I, J : INTEGER
; INTEGER_STRING : ARRAY [ 1 .. 4 ] OF CHAR
; DECIMAL_STRING : ARRAY [ 1 .. 2 ] OF CHAR
; INT, DEC : INTEGER

; BEGIN
IF CURRENT_CHAR = NL
THEN BEGIN B_M . MOVE_RIGHT ( CURRENT_CHAR ) ; I := 1 END
ELSE I := 0

MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , J )
J := I + J
B_M . MOVE_LEFT ( CURRENT_CHAR )
INT := 0
FOR I := 1 TO 4
DO BEGIN
  INTEGER_STRING [ I ] := CURRENT_CHAR
  IF CURRENT_CHAR <> ' '
  THEN INT := INT * 10 + ORD ( CURRENT_CHAR ) - 48
  B_M . MOVE_LEFT ( CURRENT_CHAR )
END

B_M . MOVE_LEFT ( CURRENT_CHAR )
DEC := 0
FOR I := 1 TO 2
DO BEGIN
  IF CURRENT_CHAR <> ' '
  THEN DEC := DEC * 10 + ORD ( CURRENT_CHAR ) - 48
  B_M . MOVE_LEFT ( CURRENT_CHAR )
END

FOR I := 1 TO J - 8 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR , J )
B_M . MOVE_LEFT ( CURRENT_CHAR )
IF CURRENT_CHAR = EM
THEN BEGIN
  FOR I := 1 TO J + 1 DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
  B_M . ADD ( NL )
  C_B_M . COPY_PUSH ( NL )
  INT := INT + 1
  FOR I := 4 DOWNTO 1
  DO BEGIN
    INTEGER_STRING [ I ] := CHR ( INT MOD 10 + 48 )
    INT := INT DIV 10
  END
  I := 1
  WHILE ( INTEGER_STRING [ I ] = '0' ) AND ( I < 4 )
  DO BEGIN
    B_M . ADD ( ' ' )
    C_B_M . COPY_PUSH ( ' ' )
    I := I + 1
  END
END
B_M . ADD ( ' ' )
C_B_M . COPY_PUSH ( INTEGER_STRING [ J ] )
END

FOR J := 1 TO 4
DO BEGIN
B_M . ADD ( ' ' )
C_B_M . COPY_PUSH ( ' ' )
END
END
917     ELSE BEGIN
918         FOR I := 1 TO J + 1 DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
919         B_M . ADD ( NL )
920         C_B_M . COPY_PUSH ( NL )
921         FOR I := 1 TO 4
922             DO BEGIN
923                 B_M . ADD ( INTEGER_STRING [ I ] )
924                 C_B_M . COPY_PUSH ( INTEGER_STRING [ I ] )
925             END
926             DEC := DEC + 1
927             B_M . ADD ( '.' )
928             C_B_M . COPY_PUSH ( '.' )
929             FOR I := 2 DOWNTO 1
930                 DO BEGIN
931                     DECIMAL_STRING [ I ] := CHR ( DEC MOD 10 + 48 )
932                 DEC := DEC DIV 10
933             END
934             FOR I := 1 TO 2
935                 DO BEGIN
936                     B_M . ADD ( DECIMAL_STRING [ I ] )
937                     C_B_M . COPY_PUSH ( DECIMAL_STRING [ I ] )
938             END
939             B_M . ADD ( ' ' )
940             C_B_M . COPY_PUSH ( ' ' )
941         END
942     END
943
944     ; PROCEDURE output_current_line ( VAR CURRENT_CHAR : CHAR )
945
946     ; VAR L, J, I : INTEGER
947
948     ; BEGIN
949     IF CURRENT_CHAR = NL
950         THEN BEGIN L := 1 ; B_M . MOVE_RIGHT ( CURRENT_CHAR ) END
951     ELSE L := 0
952         ; MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , J )
953     ; L := L + J - 1
954     ; B_M . MOVE_LEFT ( CURRENT_CHAR )
955         ; MOVE_CURSOR ( CURSOR_ROW , 1 )
956     ; OUTPUT_REST_OF_LINE ( CURRENT_CHAR , J )
957     ; FOR I := J TO 80 DO IO . OUTPUT ( ' ' )
958     ; IF SHOW_LINE_NUMBERS
959         THEN BEGIN
960             CURSOR_COL := L + 1
961         ; FOR I := J DOWNTO L + 1 DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
962     END
963     ELSE BEGIN
964         CURSOR_COL := L - 7
965         ; FOR I := J + 8 DOWNTO L DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
966     END
967     ; MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )
968     END
969
970     ; PROCEDURE insert_mode ( VAR CURRENT_CHAR : CHAR )
; VAR C, CC : CHAR
; I, J, L, SAVED_COPY_COUNT, COPY_COUNT, NUMBER_OF_SPACES
; INTEGER
; SPREAD, SPLIT : BOOLEAN

; BEGIN

INSERT_PROMPT
; IO . INPUT ( C )
; IF NOT ( C IN [ ETX, ESC ] )
THEN BEGIN

REPEAT C_B_M . COPY_POP ( CC ) UNTIL CC = EM
; C_B_M . COPY_PUSH ( EM )
; COPY_COUNT := 0
; SPLIT := FALSE
; SPREAD := TRUE
; SPREAD_CURRENT_LINE ( CURRENT_CHAR, NUMBER_OF_SPACES )
; IF NUMBER_OF_SPACES < 2
THEN BEGIN

SPLIT_CURRENT_LINE ( CURRENT_CHAR )
; SPLIT := TRUE
; SPREAD := FALSE
END

REPEAT IF C = CR
THEN BEGIN

IF SPREAD
THEN BEGIN

FOR I := CURSOR_COL TO 80 DO IO . OUTPUT ( ' ' )
; IF ( CURSOR_ROW = 24 ) OR ( CURSOR_ROW = 23 )
THEN BEGIN

IO . OUTPUT ( NL )
; IF SHOW_LINE_NUMBERS
THEN CURSOR_COL := 9
ELSE CURSOR_COL := 1
; CURSOR_ROW := 24
; INSERT_PROMPT
; COPY_COUNT := COPY_COUNT + 9
; INSERT_A_NL_AND_LINE_NO ( CURRENT_CHAR )
; SPREAD_CURRENT_LINE
( CURRENT_CHAR, NUMBER_OF_SPACES )
END

ELSE BEGIN

CURSOR_ROW := CURSOR_ROW + 1
; COPY_COUNT := COPY_COUNT + 9
; INSERT_A_NL_AND_LINE_NO ( CURRENT_CHAR )
; SPREAD := FALSE
; SPLIT := TRUE
; SPLIT_CURRENT_LINE ( CURRENT_CHAR )
END

ELSE IF SPLIT
THEN BEGIN

IF CURSOR_ROW < 23
THEN BEGIN
CURSOR_ROW := CURSOR_ROW + 1
COPY_COUNT := COPY_COUNT + 9
INSERT_A_NL_AND_LINE_NO ( CURRENT_CHAR )
OUTPUT_FRONT_OF_LINE ( CURRENT_CHAR )
END
ELSE_BEGIN
COPY_COUNT := COPY_COUNT + 9
INSERT_A_NL_AND_LINE_NO ( CURRENT_CHAR )
MOVE_CURSOR ( 24, 1 )
IF CURRENT_CHAR <> NL
THEN FOR I := 1 TO 80 DO IO . OUTPUT ( ' ' )
FOR I := 1 TO 19 DO IO . OUTPUT ( NL )
MOVE_CURSOR ( 24, 9 )
SPREAD_END_OF_LINE ( CURRENT_CHAR )
INSERT_PROMPT
CURSOR_ROW := 5
CURSOR_COL := 1
OUTPUT_FRONT_OF_LINE ( CURRENT_CHAR )
END
END
ELSE_IF C IN SET_OF_VALID_INPUT_CHARACTERS
THEN_BEGIN
B_M . ADD ( C )
C_B_M . COPY_PUSH ( C )
COPY_COUNT := COPY_COUNT + 1
CURSOR_COL := CURSOR_COL + 1
IO . OUTPUT ( C )
IO . SHIP_OUT
IF SPREAD
THEN_BEGIN
NUMBER_OF_SPACES := NUMBER_OF_SPACES - 1
IF NUMBER_OF_SPACES < 2
THEN BEGIN
IF CURSOR_ROW = 24
THEN BEGIN
IO . OUTPUT ( NL )
CURSOR_ROW := CURSOR_ROW - 1
INSERT_PROMPT
END
SPLIT_CURRENT_LINE ( CURRENT_CHAR )
SPLIT := TRUE
SPREAD := FALSE
END
END
END
IF ( COPY_COUNT > 0 )
AND ( ( ( CURSOR_COL = 1 )
OR ( SHOW_LINE_NUMBERS AND ( CURSOR_COL = 9 )
) )
AND ( C = BS )
)
OR ( C = DEL )
THEN BEGIN
  L := CURSOR_COL
  ; B_M . MOVE_RIGHT ( CURRENT_CHAR )
  ; WHILE ( COPY_COUNT > 0 ) AND ( CURRENT_CHAR <> NL )
  DO BEGIN
    B_M . DELETE ( CURRENT_CHAR )
    ; B_M . MOVE_RIGHT ( CURRENT_CHAR )
    ; COPY_COUNT := COPY_COUNT - 1
    ; C_B_M . COPY_POP ( CC )
    ; CURSOR_COL := CURSOR_COL - 1
  END
  ; IF ( COPY_COUNT = 0 ) AND ( CURRENT_CHAR <> NL )
  THEN BEGIN
    B_M . MOVE_LEFT ( CURRENT_CHAR )
    ; IF SPREAD
    THEN BEGIN
      SPREAD_CURRENT_LINE
      ( CURRENT_CHAR , NUMBER_OF_SPACES )
      ; IF CURRENT_CHAR = NL
      THEN FOR I := CURSOR_COL TO 80
      DO IO . OUTPUT ( ' ' )
      END
    ELSE BEGIN
      MOVE_CURSOR ( CURSOR_ROW , 9 )
      ; FOR I := 9 TO 80 DO IO . OUTPUT ( ' ' )
      ; OUTPUT_FRONT_OF_LINE ( CURRENT_CHAR )
    END
  END
END
ELSE BEGIN
  MOVE_CURSOR ( CURSOR_ROW , 1 )
  ; FOR I := 1 TO L DO IO . OUTPUT ( ' ' )
  ; B_M . DELETE ( CURRENT_CHAR )
  ; COPY_COUNT := COPY_COUNT - 1
  ; C_B_M . COPY_POP ( CC )
  ; IF CURRENT_CHAR = NL
  THEN BEGIN
    I := 1
    ; B_M . MOVE_RIGHT ( CURRENT_CHAR )
  END
  ELSE I := 0
  ; MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , J )
  ; J := J + 1
  ; FOR I := 1 TO J DO B_M . MOVE_LEFT ( CURRENT_CHAR )
  ; CURSOR_COL := J
  ; CURSOR_ROW := CURSOR_ROW - 1
  ; IF CURSOR_ROW = 1
  THEN BEGIN
    CURSOR_ROW := 2
    ; OUTPUT_FRONT_OF_LINE ( CURRENT_CHAR )
  END
  ELSE MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )
END
END
ELSE IF ( C = BS ) AND ( COPY_COUNT > 0 )
THEN BEGIN
COPY_COUNT := COPY_COUNT - 1 ; C_B_M . COPY_POP ( CC )
; B_M . MOVE_RIGHT ( CURRENT_CHAR )
; B_M . DELETE ( CURRENT_CHAR )
; IF CURSOR_COL < 81
THEN BEGIN
IO . OUTPUT ( BS )
; IO . OUTPUT ( ' ' )
; IO . OUTPUT ( BS )
END
ELSE IO . OUTPUT ( ' ' )
; CURSOR_COL := CURSOR_COL - 1
; IO . SHIP_OUT
END
; IO . INPUT ( C )
UNTIL C IN [ ESC , ETX ]
; IF C = ESC THEN "DELETE COPY_COUNT OF CHAR BACKWORDS." BEGIN
IF COPY_COUNT > 0 THEN BEGIN
FOR I := COPY_COUNT DOWNTO 1
DO BEGIN
B_M . MOVE_RIGHT ( CURRENT_CHAR )
; IF CURRENT_CHAR = NL
THEN SPLIT := TRUE
; B_M . DELETE ( CURRENT_CHAR )
END
END
END
; IF SPLIT THEN VERIFY ( CURRENT_CHAR )
ELSE OUTPUT_CURRENT_LINE ( CURRENT_CHAR )
END
END
END
; PROCEDURE DIRECTION ( VAR COMMAND_CHAR , DIRECTION_CHAR : CHAR )
BEGIN
IF COMMAND_CHAR IN [ '<' , ',' ]
THEN BEGIN
DIRECTION_CHAR := '<'
; SAVE_CURSOR_POSITION
; MOVE_CURSOR ( 1 , 1 )
; IO . OUTPUT ( '<' )
; UNSAVE_CURSOR_POSITION
; MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )
END
; IF COMMAND_CHAR IN [ '>' , '.' ]
THEN BEGIN
DIRECTION_CHAR := '>'
; SAVE_CURSOR_POSITION
; MOVE_CURSOR ( 1 , 1 )
1187 ; IO . OUTPUT ( '<' )
1188 ; UNSAVE_CURSOR_POSITION
1189 ; MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )
1190 END
1191 END
1192
1193 ; PROCEDURE DELETE_PROMPT ( DIRECTION_CHAR : CHAR )
1194
1195 ; VAR SROW , SCOL : INTEGER
1196
1197 ; BEGIN
1198     SROW := CURSOR_ROW
1199     SCOL := CURSOR_COL
1200     MOVE_CURSOR ( 1 , 1 )
1201     IO . OUTPUT ( DIRECTION_CHAR )
1202     IO . OUTPUT_LINE
1203     ( 'DELETE: [moving commands], [esc] to abort$' )
1204     IO . OUTPUT_LINE
1205     ( 't, [ctrl<i>o] to exit $' )
1206     MOVE_CURSOR ( SROW , SCOL )
1207 END
1208
1209 ; PROCEDURE BLANK_RIGHT
1210 ( VAR CURRENT_CHAR : CHAR
1211 ; VAR J : INTEGER
1212 ; VAR BOF : BOOLEAN
1213 )
1214
1215 ; VAR I : INTEGER
1216
1217 ; BEGIN
1218 IF CURRENT_CHAR = NL
1219 THEN BEGIN
1220 RIGHT_ARROW ( CURRENT_CHAR , BOF )
1221 ; IF NOT BOF
1222 THEN BEGIN
1223 J := J + 9
1224 ; C_B_M . COPY PUSH ( NL )
1225 ; IF SHOW_LINE_NUMBERS
1226 THEN BEGIN
1227 MOVE_CURSOR ( CURSOR_ROW , 1 )
1228 ; IO . OUTPUT_LINE ( '$' )
1229 ; CURSOR_COL := 9
1230 END
1231 ; FOR I := 8 DOWNTO 1 DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
1232 ; FOR I := 1 TO 8
1233 DO BEGIN
1234 C_B_M . COPY_PUSH ( CURRENT_CHAR )
1235 ; CURRENT_CHAR := '
1236 ; B_M . MOVE_LEFT ( CURRENT_CHAR )
1237 END
1238 END
1239 END
1240 ELSE BEGIN
J := J + 1
; C_B_M . COPY_PUSH ( CURRENT_CHAR )
; CURRENT_CHAR := '
; IO . OUTPUT ( ' ' )
; RIGHT_ARROW ( CURRENT_CHAR , BCF )
END
END

; PROCEDURE BLANK_LEFT
( VAR CURRENT_CHAR : CHAR
; VAR J : INTEGER
; VAR TOF : BOOLEAN
)

; VAR I : INTEGER
BEGIN
IF ( SHOW_LINE_NUMBERS AND ( CURSOR_COL = 9 ) )
OR ( CURSOR_COL = 1 )
THEN BEGIN
LEFT_ARROW ( CURRENT_CHAR , TOF )
IF NOT TOF
THEN BEGIN
J := J - 9
IF SHOW_LINE_NUMBERS
THEN BEGIN
SAVE_CURSOR_POSITION
; CURSOR_ROW := CURSOR_ROW + 1
; MOVE_CURSOR ( CURSOR_ROW , 1 )
; IO . OUTPUT_LINE ( ' $' )
; UNSAVE_CURSOR_POSITION
; MOVE_CURSOR ( CURSOR_ROW , CURSOR_COL )
END
END
END
END
DO BEGIN
B_M . MOVE_LEFT ( CURRENT_CHAR )
END
do I := 0 TO 3 DO B_M . MOVE_LEFT ( CURRENT_CHAR )
END
END
END
ELSE BEGIN
J := J - 1
; LEFT_ARROW ( CURRENT_CHAR , TOF )
; C_B_M . COPY_PUSH ( CURRENT_CHAR )
; IO . OUTPUT ( ' ' )
; IO . OUTPUT ( BS )
; IO . SHIP_OUT
; CURRENT_CHAR := '
END
END
PROCEDURE RECOVER_LEFT

VAR CURRENT_CHAR : CHAR;
VAR J : INTEGER;
VAR I : INTEGER;
NOTUSED : BOOLEAN;
BEGIN
IF (SHOW_LINE_NUMBERS AND (CURSOR_COL = 9))
OR (CURSOR_COL = 1)
THEN BEGIN
J := J - 9
FOR I := 8 DOWNTO 1 DO BEGIN
B_M . MOVE_RIGHT (CURRENT_CHAR)
C_B_M . COPY_POP (CURRENT_CHAR)
END
ELSE BEGIN
J := J - 1
END
END

PROCEDURE RECOVER_RIGHT

VAR CURRENT_CHAR : CHAR;
VAR J : INTEGER;
VAR I : INTEGER;
NOTUSED : BOOLEAN;
BEGIN
IF CURRENT_CHAR = NL THEN BEGIN
J := J + 9
C_B_M . COPY_POP (CURRENT_CHAR)
RIGHT_ARROW (CURRENT_CHAR, NOTUSED)
FOR I := 8 DOWNTO 1 DO B_M . MOVE_RIGHT (CURRENT_CHAR)
1349 ; IF SHOW_LINE_NUMBERS
1350 THEN MOVE_CURSOR ( CURSOR_ROW , 1 )
1351 ; FOR I := 1 TO 8
1352 DO BEGIN
1353 C_B_M . COPY_POP ( CURRENT_CHAR )
1354 IF SHOW_LINE_NUMBERS
1355 THEN IO . OUTPUT ( CURRENT_CHAR )
1356 ; B_M . MOVE_LEFT ( CURRENT_CHAR )
1357 END
1358 ; CURSOR_COL := 9
1359 ; IO . SHIP_OUT
1360 END
1361 ELSE BEGIN
1362 J := J + 1
1363 C_B_M . COPY_POP ( CURRENT_CHAR )
1364 ; IO . OUTPUT ( CURRENT_CHAR )
1365 ; RIGHT_ARROW ( CURRENT_CHAR , NOTUSED )
1366 END
1367 END
1368
1369 ; PROCEDURE DELETE_MODE_DOWN
1370 ( VAR CURRENT_CHAR : CHAR
1371 ; VAR COPY_COUNT , P_TARGET_COL : INTEGER
1372 )
1373
1374 VAR BOF : BOOLEAN
1375 ; TARGET_COL : INTEGER
1376
1377 BEGIN
1378 ; TARGET_COL := P_TARGET_COL
1379 IF COPY_COUNT > - 1
1380 THEN BEGIN
1381 BOF := FALSE
1382 WHILE ( CURRENT_CHAR <> NL ) AND NOT BOF
1383 DO BLANK_RIGHT ( CURRENT_CHAR , COPY_COUNT , BOF )
1384 ; IF NOT BOF
1385 THEN BLANK_RIGHT ( CURRENT_CHAR , COPY_COUNT , BOF )
1386 ; WHILE NOT BOF
1387 AND ( CURRENT_CHAR <> NL )
1388 AND ( CURSOR_COL < TARGET_COL )
1389 DO BLANK_RIGHT ( CURRENT_CHAR , COPY_COUNT , BOF )
1390 END
1391 ELSE BEGIN
1392 WHILE ( CURRENT_CHAR <> NL ) AND ( COPY_COUNT <> 0 )
1393 DO RECOVER_RIGHT ( CURRENT_CHAR , COPY_COUNT )
1394 ; IF COPY_COUNT = 0
1395 THEN BEGIN
1396 WHILE ( CURRENT_CHAR <> NL ) AND NOT BOF
1397 DO BLANK_RIGHT ( CURRENT_CHAR , COPY_COUNT , BOF )
1398 ; IF NOT BOF
1399 THEN BLANK_RIGHT ( CURRENT_CHAR , COPY_COUNT , BOF )
1400 ; WHILE NOT BOF
1401 AND ( CURRENT_CHAR <> NL )
1402 AND ( CURSOR_COL < TARGET_COL )
DO BLANK_RIGHT (CURRENT_CHAR, COPY_COUNT, BOF)
END
ELSE BEGIN
  RECOVER_RIGHT (CURRENT_CHAR, COPY_COUNT)
  WHILE (COPY_COUNT < 0)
  AND (CURRENT_CHAR <> NL)
  AND (CURSOR_COL < TARGET_COL)
  DO RECOVER_RIGHT (CURRENT_CHAR, COPY_COUNT)
  ; WHILE (CURRENT_CHAR <> NL)
  AND (CURSOR_COL < TARGET_COL)
  DO BLANK_RIGHT (CURRENT_CHAR, COPY_COUNT, BOF)
END
END

; PROCEDURE DELETE_MODE_RETURN
( VAR CURRENT_CHAR : CHAR ; VAR COPY_COUNT : INTEGER )

; VAR COL : INTEGER

BEGIN
  IF SHOW_LINE_NUMBERS
   THEN BEGIN
     COL := 9
   ; DELETE_MODE_DOWN (CURRENT_CHAR, COPY_COUNT, COL)
   END
   ELSE BEGIN
     COL := 1
     ; DELETE_MODE_DOWN (CURRENT_CHAR, COPY_COUNT, COL)
   END
END

; PROCEDURE DELETE_MODE_UP
( VAR CURRENT_CHAR : CHAR ; VAR COPY_COUNT : INTEGER )

; VAR TOF : BOOLEAN
; TARGET_COL : INTEGER

BEGIN
  TARGET_COL := CURSOR_COL
  IF COPY_COUNT < 1
   THEN BEGIN
     TOF := FALSE
     WHILE (CURRENT_CHAR <> NL) AND NOT TOF
     DO BLANK_LEFT (CURRENT_CHAR, COPY_COUNT, TOF)
   END
   IF NOT TOF
   THEN BLANK_LEFT (CURRENT_CHAR, COPY_COUNT, TOF)
   ; WHILE NOT TOF AND (CURSOR_COL > TARGET_COL)
   DO BLANK_LEFT (CURRENT_CHAR, COPY_COUNT, TOF)
END
ELSE BEGIN
  WHILE (CURRENT_CHAR <> NL) AND (COPY_COUNT <> 0)
  DO RECOVER_LEFT (CURRENT_CHAR, COPY_COUNT)
END
IF COPY_COUNT = 0
THEN BEGIN
  WHILE ( CURRENT_CHAR <> NL ) AND NOT TOF
    DO BLANK_LEFT ( CURRENT_CHAR , COPY_COUNT , TOF )
  ; IF NOT TOF
  THEN BLANK_LEFT ( CURRENT_CHAR , COPY_COUNT , TOF )
  WHILE CURSOR_COL > TARGET_COL
  DO BLANK_LEFT ( CURRENT_CHAR , COPY_COUNT , TOF )
END
ELSE BEGIN
  RECOVER_LEFT ( CURRENT_CHAR , COPY_COUNT )
  ; WHILE ( COPY_COUNT > 0 )
    AND ( CURRENT_CHAR <> NL )
    AND ( CURSOR_COL > TARGET_COL )
  DO RECOVER_LEFT ( CURRENT_CHAR , COPY_COUNT )
  ; WHILE CURSOR_COL > TARGET_COL
  DO BLANK_LEFT ( CURRENT_CHAR , COPY_COUNT , TOF )
END

; PROCEDURE DELETE_MODE ( VAR CURRENT_CHAR : CHAR )

; VAR C , CC : CHAR
; NL_FOUND : BOOLEAN
; COPY_COUNT , J , I , L , SAVED_COPY_COUNT : INTEGER
; NOTUSED : BOOLEAN

BEGIN
  DELETE_PROMPT ( DIRECTION_CHAR )
  ; IO . INPUT ( C )
  ; IF NOT ( C IN [ ESC , ETX ] )
  THEN BEGIN
    REPEAT C_B.M . COPY_POP ( CC ) UNTIL CC = EM
    ; C_B.M . COPY_PUSH ( CC )
    ; COPY_COUNT := 0
    ; REPEAT IF ( C = BS )
    OR ( ( C = ' ' ) AND ( DIRECTION_CHAR = '<' ) )
    THEN IF COPY_COUNT < 1
      THEN BLANK_LEFT ( CURRENT_CHAR , COPY_COUNT , NOTUSED )
      ELSE RECOVER_LEFT ( CURRENT_CHAR , COPY_COUNT )
    ; IF ( C = FF )
    OR ( ( C = ' ' ) AND ( DIRECTION_CHAR = '>' ) )
    THEN IF COPY_COUNT > - 1
      THEN BLANK_RIGHT ( CURRENT_CHAR , COPY_COUNT , NOTUSED )
      ELSE RECOVER_RIGHT ( CURRENT_CHAR , COPY_COUNT )
    ; IF C = NL
    THEN DELETE_MODE_DOWN
      ( CURRENT_CHAR , COPY_COUNT , CURSOR_COL )
    ; IF C = VT
    THEN DELETE_MODE_UP ( CURRENT_CHAR , COPY_COUNT )
    ; IF C = CR
    THEN DELETE_MODE_RETURN ( CURRENT_CHAR , COPY_COUNT )
    ; IF C IN [ 'V' , 'v' ]
  THEN BEGIN
1511        VERIFY ( CURRENT_CHAR )
1512        ; DELETE_PROMPT ( DIRECTION_CHAR )
1513        END
1514        ; IF C IN [ '>', '<', '.', ',', ']' ]
1515        THEN DIRECTION ( C, DIRECTION_CHAR )
1516        ; IO . INPUT ( C )
1517        UNTIL C IN [ ESC, ETX ]
1518        ; IF C = ESC
1519        THEN BEGIN
1520        SAVED_COPY_COUNT := COPY_COUNT
1521        ; WHILE COPY_COUNT < 0
1522        DO RECOVER_RIGHT ( CURRENT_CHAR, COPY_COUNT )
1523        ; WHILE COPY_COUNT > 0
1524        DO RECOVER_LEFT ( CURRENT_CHAR, COPY_COUNT )
1525        ; IF SAVED_COPY_COUNT < 0
1526        THEN BEGIN
1527        FOR I := - 1 DOWNTO SAVED_COPY_COUNT
1528        DO BEGIN
1529            B_M . MOVE_RIGHT ( CURRENT_CHAR )
1530            ; C_B_M . COPY_PUSH ( CURRENT_CHAR )
1531        END
1532        ; FOR I := - 1 DOWNTO SAVED_COPY_COUNT
1533        DO B_M . MOVE_LEFT ( CURRENT_CHAR )
1534        END
1535        ; IF SAVED_COPY_COUNT > 0
1536        THEN BEGIN
1537        FOR I := 1 TO SAVED_COPY_COUNT
1538        DO BEGIN
1539            C_B_M . COPY_PUSH ( CURRENT_CHAR )
1540            ; B_M . MOVE_LEFT ( CURRENT_CHAR )
1541        END
1542        ; FOR I := 1 TO SAVED_COPY_COUNT
1543        DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
1544        END
1545        END
1546        ; IF C = ETX
1547        THEN BEGIN
1548        NL_FOUND := FALSE
1549        ; IF COPY_COUNT < 0
1550        THEN FOR I := - 1 DOWNTO COPY_COUNT
1551        DO BEGIN
1552            IF CURRENT_CHAR = NL
1553            THEN NL_FOUND := TRUE
1554            ; B_M . DELETE ( CURRENT_CHAR )
1555        END
1556        ; IF COPY_COUNT > 0
1557        THEN FOR I := 1 TO COPY_COUNT
1558        DO BEGIN
1559            B_M . MOVE_RIGHT ( CURRENT_CHAR )
1560            ; IF CURRENT_CHAR = NL
1561            THEN NL_FOUND := TRUE
1562            ; B_M . DELETE ( CURRENT_CHAR )
1563        END
1564        ; IF NL_FOUND
THEN VERIFY ( CURRENT_CHAR )
ELSE OUTPUT_CURRENT_LINE ( CURRENT_CHAR )
END

; EDIT_PROMPT ( DIRECTION_CHAR )
PEND

; PROCEDURE EXCHANGE_PROMPT

; VAR SROW , SCOL : INTEGER
BEGIN
SROW := CURSOR_ROW
SCOL := CURSOR_COL
MOVE_CURSOR ( 1 , 1 )
IO . OUTPUT_LINE
( 'EXCHANGE: to return characters [left arr$]' )
IO . OUTPUT_LINE
( 'ow], to exit [ctrl<c>], to abort [esc] $' )
MOVE_CURSOR ( SROW , SCOL )
END

; PROCEDURE EXCHANGE_MODE ( VAR CURRENT_CHAR : CHAR )

; VAR C : CHAR
; COPY_COUNT , I : INTEGER
; NOTUSED : BOOLEAN
BEGIN
COPY_COUNT := 0
EXCHANGE_PROMPT
IO . INPUT ( C )
WHILE NOT ( C IN [ ESC , ETX ] ) DO BEGIN
IF ( C IN SET_OF_VALID_INPUT_CHARACTERS )
AND ( CURRENT_CHAR <> NL )
THEN BEGIN
C_B_M . COPY_PUSH ( CURRENT_CHAR )
COPY_COUNT := COPY_COUNT + 1
CURRENT_CHAR := C
IO . OUTPUT ( CURRENT_CHAR )
RIGHT_ARROW ( CURRENT_CHAR , NOTUSED )
END
IF ( C = FF ) AND ( CURRENT_CHAR <> NL )
THEN BEGIN
C_B_M . COPY_PUSH ( CURRENT_CHAR )
COPY_COUNT := COPY_COUNT + 1
RIGHT_ARROW ( CURRENT_CHAR , NOTUSED )
END
IF C = BS
THEN BEGIN
IF ( COPY_COUNT = 0 )
THEN LEFT_ARROW ( CURRENT_CHAR , NOTUSED )
IF COPY_COUNT > 0
THEN BEGIN
    LEFT_ARROW ( CURRENT_CHAR, NOTUSED )
    \ C_B_M . COPY_POP ( CURRENT_CHAR )
    \ COPY_COUNT := COPY_COUNT - 1
    \ IO . OUTPUT ( CURRENT_CHAR )
    \ IO . OUTPUT ( BS )
    \ IO . SHIP_OUT
    END
END

;

IF C = ETX
THEN FOR I := 1 TO COPY_COUNT DO C_B_M . COPY_POP ( C )
;
IF C = ESC
THEN BEGIN
FOR I := 1 TO COPY_COUNT
DO BEGIN
    LEFT_ARROW ( CURRENT_CHAR, NOTUSED )
    \ C_B_M . COPY_POP ( CURRENT_CHAR )
    \ IO . OUTPUT ( CURRENT_CHAR )
END
;
IO . SHIP_OUT
END

END

PROCEDURE QUIT_MODE
( VAR DONE : BOOLEAN ; VAR CURRENT_CHAR : CHAR )

;
VAR I , COUNT : INTEGER

;
BEGIN
    DONE := TRUE
    ; MOVE_CURSOR ( 24, 1 )
    ; IO . OUTPUT ( NL )
    ; IO . SHIP_OUT
    ; IO . OUTPUT_LINE ( 'QUITTING$' )
    ; WHILE CURRENT_CHAR <> EM DO B_M . MOVE_RIGHT ( CURRENT_CHAR )
    FOR I := 1 TO 10 DO B_M . DELETE ( CURRENT_CHAR )
    COUNT := 90
    ; IO . OUTPUT ( '***' )
    ; IO . SHIP_OUT
    ; REPEAT WHILE CURRENT_CHAR <> NL
        ; B_M . MOVE_LEFT ( CURRENT_CHAR )
    ; B_M . MOVE_LEFT ( CURRENT_CHAR )
    ; IF CURRENT_CHAR <> EM
    ; THEN BEGIN
        FOR I := 1 TO 8 DO B_M . DELETE ( CURRENT_CHAR )
        COUNT := COUNT + 1
    ; IF COUNT MOD 10 = 0
    ; THEN BEGIN
        ; IO . OUTPUT ( '***' )
    ; IO . SHIP_OUT
    ; IF COUNT MOD 800 = 0
    ; THEN BEGIN
IO . OUTPUT ( NL )
; IO . OUTPUT ( CR )
; IO . SHIP_OUT
END
END
UNTIL CURRENT_CHAR = EM
; B_M . MOVE_LEFT ( CURRENT_CHAR )
; B_M . FILL_LEFT_STACK
; IO . OUTPUT ( NL )
; IO . OUTPUT ( CR )
; IO . SHIP_OUT
END

; PROCEDURE SPACE ( VAR CURRENT_CHAR, DIRECTION_CHAR : CHAR )
VAR NOTUSED : BOOLEAN
BEGIN
IF DIRECTION_CHAR = '<'
THEN LEFT_ARROW ( CURRENT_CHAR, NOTUSED )
IF DIRECTION_CHAR = '>'
THEN RIGHT_ARROW ( CURRENT_CHAR, NOTUSED )
END

; PROCEDURE JUMP_PROMPT
VAR SROW, SCOL : INTEGER
BEGIN
SROW := CURSOR_ROW
SCOL := CURSOR_COL
MOVE_CURSOR ( ?, ? )
IO . OUTPUT_LINE
'JUMP: E(nd B(e)ginning L(ine) number, to exit [space,ctrl<o>,or esc]$'
IO . OUTPUT_LINE ( ' $' )
MOVE_CURSOR ( SROW, SCOL )
END

; PROCEDURE JUMP_MODE ( VAR CURRENT_CHAR : CHAR )
VAR C : CHAR
BEGIN
JUMP_PROMPT
REPEAT IO . INPUT ( C )
IF C IN [ 'E', 'e' ]
THEN BEGIN
B_M . MOVE_LEFT ( CURRENT_CHAR )
IF CURRENT_CHAR = EM
THEN B_M . MOVE_RIGHT ( CURRENT_CHAR )
ELSE BEGIN
MOVE_TO_END_OF_TEXT (CURRENT_CHAR)
; VERIFY (CURRENT_CHAR)
END

IF C IN ['B', 'b'] THEN BEGIN
  B_M . MOVE_RIGHT (CURRENT_CHAR)
  IF CURRENT_CHAR = EM THEN BEGIN
    B_M . MOVE_LEFT (CURRENT_CHAR)
  END
END ELSE BEGIN
  MOVE_TO_BEGINNING_OF_TEXT (CURRENT_CHAR)
  ; VERIFY (CURRENT_CHAR)
END
UNTIL (C IN [' ', ESC, ETX, CR, 'E', 'e', 'B', 'b'])

END

; PROCEDURE FIX_PROMPT

; BEGIN
; CLEAR_SCREEN
  ; IO . OUTPUT_LINE
  ( 'FIX: Fix mode will let you rid your file of$' )
  ; IO . OUTPUT_LINE ( ' unwanted characters.(:10:)(:13:)$' )
  ; IO . OUTPUT_LINE
  ( '(:10:)Type a G or g to start the search$' )
  ; IO . OUTPUT_LINE
  ( ' from the current cursor position.(:10:)(:13:)$' )
  ; IO . OUTPUT_LINE
  ( '(:10:)All nonprinting characters except linefeed and(:10:)(:13:)$' )
  ; IO . OUTPUT_LINE
  ( '(:10:)Tab characters will be detect.(:10:)(:13:)$' )
  ; IO . OUTPUT_LINE
  ( '(:10:)All trailing blanks will be deleted.(:10:)(:13:)$' )
  ; IO . OUTPUT_LINE ( '(:10:)Type anything else to continue.$' )
END

; PROCEDURE FIX_MODE (VAR CURRENT_CHAR : CHAR)

; VAR C : CHAR
; J, I : INTEGER

; BEGIN
  FIX_PROMPT
  ; IO . INPUT (C)
  ; IF C IN ['G', 'g']
THEN BEGIN
    B_M . MOVE_RIGHT ( CURRENT_CHAR )
    MOVE_TO_END_OF_PREVIOUS_LINE ( CURRENT_CHAR , J )
    FOR I := 1 TO J DO B_M . MOVE_LEFT ( CURRENT_CHAR )
    WHILE ( CURRENT_CHAR = ' ' ) AND ( J > 8 )
    DO BEGIN
        B_M . DELETE ( CURRENT_CHAR )
        B_M . MOVE_RIGHT ( CURRENT_CHAR )
        J := J - 1
    END

ELSE IF CURRENT_CHAR
    IN SET_OFVALID_INPUT_CHARACTERS + [ TAB ]
    THEN B_M . MOVE_LEFT ( CURRENT_CHAR )
ELSE BEGIN
    B_M . DELETE ( CURRENT_CHAR )
    IO . OUTPUT_LINE
        ( '**:10:):(13:)BAD CHARACTER FOUND!$' )
END

UNTIL CURRENT_CHAR = EM
B_M . MOVE_RIGHT ( CURRENT_CHAR )
IO . OUTPUT_LINE ( '**:10:):(10:):(13:)DONE$' )
IO . OUTPUT_LINE
    ( '**:10:):(13:)Type any key to continue.$' )
IO . INPUT ( C )
END

BEGIN
    IF SECOND_EDIT_PROMPT_ON
    THEN BEGIN
        EDIT_PROMPT ( DIRECTION_CHAR )
        SECOND_EDIT_PROMPT_ON := FALSE
    END
ELSE BEGIN
    SECOND_EDIT_PROMPT_ON := TRUE
    SROW := CURSOR_ROW
    SCOL := CURSOR_COL
    MOVE_CURSOR ( 1 , 1 )
    IO . OUTPUT_LINE
        ( 'SEDIT: < > [movement commands] $' )
    IO . OUTPUT_LINE
        ( ' $' )
    MOVE_CURSOR ( SROW , SCOL )
; PROCEDURE EXECUTE_COMMAND
( VAR CURRENT_CHAR, COMMAND_CHAR, DIRECTION_CHAR : CHAR
; VAR DONE : BOOLEAN
)

; VAR NOTUSED : BOOLEAN
; J, N : INTEGER

BEGIN

CASE COMMAND_CHAR OF
BS : LEFT_ARROW ( CURRENT_CHAR, NOTUSED )
; FF : RIGHT_ARROW ( CURRENT_CHAR, NOTUSED )
VT : UP_ARROW ( CURRENT_CHAR, NOTUSED )
NL : DOWN_ARROW ( CURRENT_CHAR, NOTUSED )
CR : RETURN ( CURRENT_CHAR )
'i', 'I'
BEGIN
INSERT_MODE ( CURRENT_CHAR )
EDIT_PROMPT ( DIRECTION_CHAR )
END

'd', 'D' : DELETE_MODE ( CURRENT_CHAR )
;x', 'X'
BEGIN
EXCHANGE_MODE ( CURRENT_CHAR )
EDIT_PROMPT ( DIRECTION_CHAR )
END

'q', 'Q' : QUIT_MODE ( DONE, CURRENT_CHAR )
'y', 'Y'
BEGIN
VERIFY ( CURRENT_CHAR )
EDIT_PROMPT ( DIRECTION_CHAR )
END

'p', 'P' : MOVE_PAGE ( CURRENT_CHAR, DIRECTION_CHAR )
'<', '>', 'r', 'R'
DIRECTION ( COMMAND_CHAR, DIRECTION_CHAR )
'
SPACE ( CURRENT_CHAR, DIRECTION_CHAR )
'j', 'J'
BEGIN
JUMP_MODE ( CURRENT_CHAR )
EDIT_PROMPT ( DIRECTION_CHAR )
END

'f', 'F'
BEGIN
FIX_MODE ( CURRENT_CHAR )
VERIFY ( CURRENT_CHAR )
EDIT_PROMPT ( DIRECTION_CHAR )
END

'l', '?' : SECOND_EDIT_PROMPT ( DIRECTION_CHAR )
'1' : OUTPUT_REST_OF_LINE ( CURRENT_CHAR, J )
'2' : MOVE_TO_END_OF_CURRENT_LINE ( CURRENT_CHAR, J )
; '3' : SPREAD_CURRENT_LINE ( CURRENT_CHAR , J )
; '4' : SPLIT_CURRENT_LINE ( CURRENT_CHAR )
; '5' : MOVE_TO_END_OF_TEXT ( CURRENT_CHAR )
; '6' : OUTPUT_REST_OF_SCREEN ( CURRENT_CHAR , J , N )
; '7' : MOVE_RIGHT_1_LINE ( CURRENT_CHAR , J )
; '8' : MOVE_LEFT_1_LINE ( CURRENT_CHAR , J )
; '9' : OUTPUT_CURRENT_LINE ( CURRENT_CHAR )
ELSE : BEGIN END
END

; PROCEDURE INITIALIZE_WITH_FILE
( FILE_LOGICAL_UNIT_NUMBER : INTEGER ; VAR CURRENT_CHAR : CHAR )
VAR NO_OF_PAGES : INTEGER
BEGIN

IO . OUTPUT_LINE ( 'INITIALIZING$' )
; B_M . FIND_NO_OF_PAGES
( FILE_LOGICAL_UNIT_NUMBER , NO_OF_PAGES )
; B_M . INIT_RIGHT_STACK
( FILE_LOGICAL_UNIT_NUMBER , NO_OF_PAGES )
CURRENT_CHAR := EM
ADD_LINE_NUMBERS ( CURRENT_CHAR )
END

; PROCEDURE GO

VAR CURRENT_CHAR , COMMAND_CHAR : CHAR
DONE : BOOLEAN
BEGIN
CLEAR_SCREEN
IO . OUTPUT_LINE ( 'EDIT VERSION 1(:10:)(:10:)(:13:)$' )
INITIALIZE_WITH_FILE ( 1 , CURRENT_CHAR )
VERIFY ( CURRENT_CHAR )
DIRECTION_CHAR := '>'
EDIT_PROMPT ( DIRECTION_CHAR )
DONE := FALSE
REPEAT
"ACCEPT_COMMAND"

IO . INPUT ( COMMAND_CHAR )
EXECUTE_COMMAND
( CURRENT_CHAR , COMMAND_CHAR , DIRECTION_CHAR , DONE )
UNTIL DONE
END

BEGIN
INIT IO , B_M ( IO ), S7 , C_B_M
SHOW_LINE_NUMBERS := TRUE
SECOND_EDIT_PROMPT_ON := FALSE
GO
END
A FULL SCREEN EDITOR IMPLEMENTED IN PASCAL

by

THEODORE JOHN SOCOLOFSKY

B.S., KANSAS STATE UNIVERSITY, 1980

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1981
ABSTRACT

A Full Screen Editor has been built. This editor provides a CRT terminal user an efficient means of editing information displayed on the screen. This editor, in contrast to most editors, provides the user a direct visual image of the contents of a text file, and a means to move, correct, change and manipulate displayed text files. Accordingly, full screen editors are much more efficient and effective than other usual types of editors. The editor was written in PASCAL language which was chosen because of its portability to other machines and operating systems.