PAGED : A Page Oriented Text Editor

Egardt, Bo; Elmqvist, Hilding

1979

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-718/)/1-043/(1979)

PAGED - A PAGE ORIENTED TEXT EDITOR

BO EGARDT
HILDING ELMQVIST

DEPARTMENT OF AUTOMATIC CONTROL
LUND INSTITUTE OF TECHNOLOGY
DECEMBER 1979

PAGED

A Page Oriented Text Editor

Bo Egardt

Hilding Elmqvist

Department of Automatic Control
Lund Institute of Technology

December 1979

CONTENTS

INTRODUCTION

PAGED is an editor (PAGe EDitor), designed to be run from an intelligent terminal with alphanumeric display of the type Beehive B100 (B150). This terminal can be used in a local mode of operation (BLOCK MODE) to perform editing of the text on the screen. Some special features - such as cursor control and insertion or deletion of characters - facilitates the editing. This local editing of the screen contents (below denoted the page) provides the basis of the editing facilities in PAGED. It has some obvious benefits compared to other (line oriented) editors:

- Cursor control is easy to learn and natural to use; no special commands are needed to arrive at the desired location in the text.
- The effects of the editing can be monitored continuously on the screen.
- Changes are always made in a known environment; this is particularly useful when editing documents.

Since the terminal is used in BLOCK MODE, it is normally not communicating with the computer. Therefore, special function keys are used to control the actions of PAGED, such as opening and closing of files, advancing in files etc. Both the editing on the screen and the use of function keys are described in the following.

GENERAL

Screen layout

The screen consists of 24 lines with 80 characters. Only 23 lines are used in a normal fashion in the editing. The last line is called the command line and is used to give additional information for the functions and for error messages. The command line has the following initial appearance.

```
.........!.........!.........!.........!.........!.........
```

Every line is ended with a special character, '!', to indicate right margin. Desired location of this margin is specified when opening the files; the page width is maximum 79 characters.

The end of the lines are normally filled with spaces when written on the screen. However, this slows down the editor when reading or writing a page. For that reason there is a special option /NULL in the file specification command which tells PAGED to end the lines with NULLs. Special care must be taken when using this option and the cursor control keys because the nulls are not transmitted to the file even if they appear as spaces on the screen. The function VERIFY can be used to check the layout of the page.

To indicate an empty line, the following convention is used: the line is beginning with a period, followed by spaces or nulls.

Input file format

The specified page width on the screen cannot be exceeded. Those lines in the input file that are too long for the page width are therefore normally split up. Exceptions, see below.

Two characters are converted when read from the input file. Form feed is converted into a new line containing '.$' (cf. output file format) and TAB is converted into the correct number of spaces.

Exceptions: The above conversions of the input file are not used when parts of the input file are directly transferred to the output file, i.e. when either of the functions TOP, TURN, CLOSE or EXIT is used.

Output_file_format

A line on the screen which consists of the string '.$' is converted into form feed when transferred to the output file (cf. input file format). If the option /TAB is specified in the file specification line then sequences of spaces are if possible converted to TABs before output. The size of the output file can sometimes be reduced in that way. Besides this, the output file looks as the input file or the screen with blanks at the end of each line deleted.

OPERATION

Start-up

PAGED is started in the usual fashion, i.e. the following command is entered in response to the period written by the monitor:

R PAGED (return)

PAGED then tells you to set the switch on rear in BLOCK MODE (marked B), writes a prompting character '*' and waits for a file specification. See OPEN.

Using function keys

The function keys control the actions of PAGED. Some rules are common for all functions:

- The function key should have a fairly quick push; if you let your finger "rest" on the key, the actions are delayed by a few seconds.
- PAGED announces that the function is completed by positioning the cursor at the upper left corner (at home). Exceptions: LOCATE and TAB.
- Some functions can be controlled with a command line on the bottom. The characters . | + and space are ignored at the end of the command line. Exception: SETTAB.
- Most functions are illegal if the files have not been opened. Exceptions: OPEN and EXIT.

The different functions are described in detail below. They are ordered and denoted in the same way as on the label, which should be placed around the function keys.

OPEN

Opens the files: The key is pressed when the file specification command has been written on the display after the prompting character '*'. The most complete form of the command is

⟨output file⟩ = ⟨input file⟩ [⟨options⟩]

where [ ] denotes optional argument. The form

⟨output file⟩ = [⟨options⟩]

is used when there is no input file (creating a new file) and the short form

⟨output file⟩ [⟨options⟩]

is used when the input and output files have the

same name.

The file specifications are the standard ones in the RT-11 operating system. The file extension is blank by default.

The following options are available:

/<page width>
Sets the page width for the terminal. Default value of <page width> is 79.

/NULL or /null
Indicates that the lines should be ended with "null"s instead of spaces. This speeds up reading and writing of the screen but causes problems in some cases when using cursor control keys.

/TAB or /tab
Indicates that multiple spaces in the text should be converted to tabs when possible before the text is outputted on the file.

The options can be given in any order.

Examples: DX1:FIL2.PAS=DX1:FIL1.PAS
DX1:FIL.DOC[30]/60
([30] means that 30 blocks are allocated)
new.tst= /70 /tab /null

Note: If the size of the output file is not specified (as in the first example above) or if the largest available area is requested (size=-1), PAGED tests if the allocated area exceeds the size of the input file by 25 blocks (corresponds to 12800 characters). If not so, an error message is given and retyping of the command can be done.

When the files have been opened, the screen will be filled with empty lines, the three first lines from the input file and the command line. It is then possible to start editing on the screen, or to use some other function key.

TOP

Positions the page at the beginning of the file: The screen contents and the rest of the input file are transferred to the output file and the files are closed. New files are then opened with the former output file as new input file. The first three lines of the input file appear on the screen.

Note: No test is made on allocated area when reopening the files. The largest available area is allocated.

BOTTOM

Positions the page at the end of the file: The last 10 lines in the input file appears on the screen.

Note: BOTTOM is legal only if an input file exists.

TURN

Advances through the file: The position of the page is advanced a desired number of lines. Default is 20 lines. If a different number is desired, a command can be written on the last line as follows.

Command:
- n — Advance n lines.
- $n — Advance n lines and change the default number of lines to n.

Note: If TURN advances beyond the end of the input file, the screen will be filled with empty lines.

LOCATE

Advances through the rest of the file (not including the current page) until a specified string is found: The string should be written on the bottom line, see below. If the string is found, the page is chosen so that the string appears on line no. N with the cursor positioned at the string. Default value for N is 10.

Command:
- string — Look for the string 'string' and use default value of N.
- string$n — Look for the string 'string' and change default value of N to n.

Note: The string must not contain '$'. If the string is not found, the screen will contain the N last lines in the file and additional empty lines. The cursor is positioned at Home position. LOCATE is legal only if an input file exists.

CONVERT

Replaces a specified string with another one throughout the file: The two strings are written on the bottom line, see below. The page is at the end of the file when CONVERT is completed.

Command: /string1/string2/ String2 replaces string1; '/' can be replaced by any other character, not present in string1 or string2.

Note: Lines that are changed are displayed on the screen. After completion, CONVERT fills the screen with empty lines. CONVERT is legal only if an input file exists.

INS/DEL

Inserts lines and deletes line feeds on the screen: The screen is read and rewritten with the following conventions. Every insert character (default '!') is converted into carriage return/line feed and every delete character (default '\') at the end of a line (spaces excluded) converts the succeeding carriage return/line feed into a blank. If the resulting page has too many lines (more than 23) some of the first lines of the page will be transferred to the output file. The insert and delete characters can be redefined by a command as follows.

Command: $AB The character 'A' is used for insertion, 'B' for deletion (from now on).

Note: If the delete character is used so that the concatenated lines exceed the page width, they will be split up in the ordinary way.

VERIFY

Verifies the page: The screen contents is read and rewritten with empty lines collected at the end of the page. Can be used to distinguish empty locations (nulls) from spaces or to update the internal buffer (see RECOVER). Furthermore, the end of the lines are filled with spaces even if the option /NULL is used.

SETTAB	Reads tab positions: The tab positions are marked with + signs on the command line.
Command:	Example:

```
.......+.!.........+.........!.....+...!.........
```

TAB	Writes spaces from the current cursor position up to the next tab position.
RECOVER	Writes the contents of the internal buffer on the screen. The buffer contains a copy of the page from the last time a function key (except TAB) was used. Can be used to recover after having accidentally erased the contents of the screen by pressing EOS (end of screen) or to cancel editing performed on the screen since the last function.
CLOSE	Closes the files: Transfers the screen contents and the rest of the input file to the output file and closes the files. A prompting character ('*') is written on the display and a new file specification command can be given.
EXIT	Closes the files and terminates the program.
Note:	Reset the switch on rear.
CANCEL	Closes the input file only: All editing performed is thus without effect. PAGED is terminated.

Editing on the display

The basic editing is performed on the text shown on the display. Some special keys on the terminal facilitates this editing. A short description of these is given below. Additional information is available in the Operator Manual for the terminal.

Note: The position in the upper left corner of the display is denoted the Home position.

↑	Causes the cursor to move one step upwards. If on the first line, the cursor will move to the last line (useful to reach the command line from home position).
↓	Causes the cursor to move one step downwards. If on the last line, the cursor will move to the first line.
←	Causes the cursor to move one step to the left. If in the first column, the cursor will move to the last column on the line above. If in Home position, the cursor will move to the last column on the last line.
→	Causes the cursor to move one step to the right. If in the last column, the cursor will move to the first column on the line below, or - if on the last line - it will move to the Home position.
DEL CHAR	Delete character. Causes the text to the right of the cursor and on the same line to be shifted one position to the left. The character under the cursor will be deleted.
INST CHAR	Insert character is an alternate action key. Any character entered while in this mode will be inserted under the cursor while all characters starting at the cursor and extending to the end of the line will shift one position to the right. The last character will be lost.

FMT TAB	Format tab. Moves the cursor to the first position on the next line.
CLEAR HOME	Shifted: The entire screen is erased. Unshifted: Moves the cursor to the Home position.
ALPHA LOCK	An alternate action key that locks the terminal in an upper case mode.
EOS	End of screen. Causes the text to be erased from the cursor position to the end of the screen.
EOL	End of line. Causes the text to be erased from the cursor position to the end of the line.

Note: When EOS or EOL is used, the screen will contain nulls. There is no way to distinguish between nulls and spaces on the screen. See however VERIFY.

IMPLEMENTATION

PAGED is programmed in PASCAL. Thanks to suitable data structures and easy character handling PASCAL has proved to be well suited for the purpose. The program consists of a main program and some external procedures to control the terminal. A short description of the program is given below.

Data_structures

The global data declarations are as follows:

```
Const version=5;
      esc=33B; cr=15B; lf=12B; stroke=174B; backsl=134B;
      ff=14B;  ht=11B;
      buffsize=2000;              {size of text buffer buff}
      pardelim='$';         {delimiter for command strings}

Type lbuff=array[1..82] of char;

Var buff:array[1..buffsize] of char;          {text buffer}
    line, comline, string1, string2: lbuff;
    first, last, nchar, ncol, lenin, lenout: integer;
    length1, length2, defturn, deflocate: integer;
    filesopen, infilepresent, exitrequest, OK, newdef,
       formfeed: boolean;
    insert, delete: character;
    infile, outfile: text;
    inname, outname: array[1..14] of char;
    fillwithnulls, inserttab: boolean;
```

A few comments will be given. The internal representation of the page is the array BUFF. This array is used in a cyclic manner e.g. when performing LOCATE. The variables FIRST and LAST are used to point at the page limits in the array. A line is stored in the array with ending blanks deleted and a line feed as terminator.

The type LBUFF is used for variables representing just one line. The variable PARDELIM defines the character that indicates new default values for parameters of certain functions. See TURN, LOCATE and INS/DEL.

External_procedures

The external procedures used in PAGED are available as library routines at the department. They are all used to control the terminal, e.g. to read function key number, to position the cursor etc. The procedures are described in Andersson (1979).

Main program

The main program is listed in the appendix. It has the following over-all structure:

```
Program PAGED;

{Global data declarations}

{External procedure declarations}

{Procedures for text line handling}
Procedure ScreenToLine; {reads a line from the screen}
Procedure LineToScreen; {writes a line on the screen}
Procedure Emptylines;   {writes empty lines on the screen}
Procedure Writecomline; {writes bottom line on the screen}
Procedure FileToLine;   {reads a line from input file}
Procedure LineToFile;   {writes a line on output file}
Procedure FileToFile;   {transfers lines from input file to
                         output file}
Procedure LineToBuff;   {puts a line in the buffer}
Procedure BuffToLine;   {gets a line from the buffer}
Procedure BuffToFile;   {transfers lines from buffer to
                         output file}
Procedure ScreenToBuff; {reads the screen into the buffer}
Procedure BuffToScreen; {writes buffer contents on the
                         screen}

{Initialization and error message procedures}
Procedure Prompt;       {writes prompting character}
Procedure Initcomline;  {initializes command line buffer}
Procedure Initialize;   {initializes screen and buffer}
Procedure Writefirstlines; {writes beginning of file
                            on the screen}
Procedure Errmess;

{Command decoding procedures}
Procedure Integ;        {reads an integer}
Procedure Readcomline;  {reads the bottom command line}
Procedure Cominteg;     {decodes an integer in a command}
Procedure Splitcommand; {decodes the file specification
                         command}

{Miscellaneous routines}
Procedure Match;        {tests occurance of string
                         in a line}
Procedure Copyandclose; {copies input file to output file
                         and closes the files}

{Procedures for function keys}
Procedure Open;
Procedure Top;
Procedure Turn;
```

```
Procedure Locate;
Procedure Convert;
Procedure Insdel;
Procedure Verify;
Procedure Settab;
Procedure Tab;
Procedure Recover;
Procedure Closefiles;
Procedure Cexit;
Procedure Cancel;

{Program body}

begin {PAGED}
exitrequest:=false; filesopen:=false; formfeed:=false;
defturn:=20; deflocate:=10;       {default values}
insert:=chr(stroke); delete:=chr(backsl);
Clear;
writeln('PAGED version ', version:2);
writeln('Set switch on rear in position block');
Prompt;

repeat
if not filesopen then
  case Readfunc of
   1: Open;
  12: Cexit;
  else Errmess(1)
  end {case}
else
  case Readfunc of
   2: Top;
   3: if infilepresent then Bottom else Errmess(1);
   4: Turn;
   5: if infilepresent then Locate else Errmess(1);
   6: if infilepresent then Convert else Errmess(1);
   7: Insdel;
   8: Verify;
   9: Settab;
  10: Tab;
  11: Recover;
  12: Closefiles;
  13: Cexit;
  16: Cancel;
  else Errmess(1)
  end; {case}
until exitrequest;
end. {PAGED}
```

Readfunc is an external integer function that waits for a function key to be depressed and returns its number. The program should be self-explanatory. Just note that the basic text handling is done line by line. To this end, a set of procedures is available to move one line to or from the different representations in a file, in the text buffer or on the screen. New functions can easily be added by writing a procedure and extending the repeat statement in the program body.

Storage requirement

The total storage requirement for PAGED together with run-time support is about 9500 words on an LSI-11. The Pascal program itself compiles to 5000 words.

REFERENCES

Andersson L (1979): Paslib - Pascal library. Department of Automatic Control, Lund Institute of Technology.

Beehive B100 Operator Manual. Beehive International 1977.

APPENDIX

Program listing

```
          Program PAGED;

{ Page oriented text editor for the Beehive B100 (B150) terminal.
  The editor is based on editing on the terminal screen with the
  terminal in block mode.
  The function keys control the actions of PAGED.
  The following functions are implemented:

   1: OPEN     - opens the files
   2: TOP      - advances to the beginning of the file
   3: BOTTOM   - advances to the end of the file
   4: TURN     - advances a number of lines through the file
   5: LOCATE   - advances to a specified string
   6: CONVERT  - converts a specified string to a new one throughout
                 the file
   7: INSDEL   - inserts and deletes lines on the screen
   8: VERIFY   - writes a verification of the screen
   9: SETTAB   - reads new tab positions
  10: TAB      - moves to next tab position
  11: RECOVER  - writes a backup copy of the screen
  12: CLOSE    - closes the files
  13: EXIT     - closes the files and terminates the program
  16: CANCEL   - cancels all editing and terminates the program

  Authors:  Bo Egardt             Department of Automatic Control
            Hilding Elmqvist      Lund Institute of Technology
  Date:     1979-08-23
  Revised:  1979-11-04
            1980-01-27 }

{-----------------------------------------------------------------}


{$A-} {$T-}
Const version = 5;
      esc=33B; cr=15B; lf=12B; stroke=174B; backsl=134B; ff=14B;
      ht=11B;
      buffsize=2000;                    {size of text buffer buff}
      pardelim='$';                     {delimiter for command strings}
      tabmark='+';
Type lbuff=array[1..82] of char;
Var buff:array[1..buffsize] of char;    {text buffer}
    line, comline, string1, string2: lbuff;
    first, last, nchar, ncol, lenin, lenout, length1, length2:integer;
    defturn, deflocate: integer;
    filesopen, infilepresent, exitrequest, OK,
    newdef, formfeed: boolean;
    insert, delete: char;
    infile, outfile: text;
    inname, outname: array[1..14] of char;
    fillwithnulls, inserttab: boolean;

{-----------------------------------------------------------------}
```

```
      { EXTERNAL PROCEDURES }
      { ------------------- }

Function Readfunc:integer; external;
{ Waits for a function key and returns its number. }

Procedure Home; external;
{ Positions the cursor in the upper left corner. }

Procedure Cursor(row,col: integer); external;
{ Positions the cursor at specified location. }

Procedure Getblock(length: integer); external;
{ Reads length lines off the screen so that its contents can be
  decoded by the read statements of Pascal. }

Procedure Getline; external;
{ Reads a line off the screen so that its contents can be decoded by
  the read statements of Pascal. }
Procedure Formon; external;
{ Sets the terminal in format mode. }

Procedure Formoff; external;
{ Takes the terminal out of format mode. }

Procedure Protect; external;
{ Writes a character sequence to indicate protected area. }

Procedure Unprotect; external;
{ Writes a character sequence to indicate end of protected area. }

Procedure Clear; external;
{ Clears the screen. }

{-------------------------------------------------------------------}
```

```
      { PROCEDURES FOR TEXT LINE HANDLING }
      { ------------------------------- }

Procedure ScreenToLine(var line:lbuff; var nchar:integer);
{ Reads one line from the screen, received by Getblock/Getline/
  Getscreen. Nchar char:s returned in line. }

  begin
  nchar:=0;
  while not eoln do
    begin
    nchar:=nchar+1;
    read(line[nchar]);
    end;
  readln;
  while (line[nchar]=' ') and (nchar>1) do nchar:=nchar-1;
                                { delete trailing spaces }
  if (nchar=1) and (line[1]='.') then nchar:=0;
                                {indicate if empty line}
end;

{-------------------------------------------------------------------}

Procedure LineToScreen(var line:lbuff; nchar:integer);
{ Writes one line on the screen with protected right margin.
  Nchar char:s are written from line.
  Spaces are inserted, if not /NULL }

  var i: integer;
  begin
    { The scrolling does not work when getting automatic CR-LF by
      writing in the 80th column, if INST CHAR is depressed.
      Avoid the problem by writing a LF followed by an inversed LF. }
  writeln;
  write(chr(esc),'A');                    {inversed line feed}
  if fillwithnulls then
     begin write(chr(esc)); write('K'); end;     { EOL - empty line }
  for i:=1 to nchar do write(line[i]);
  if (nchar=0) then begin write('.'); nchar:=1 end;

  if not fillwithnulls then
    for i:=nchar+1 to ncol do write(' ')
  else
    begin                                 { add 3 spaces at end of line }
    i := nchar+1;
    while (i <= ncol) and (i <= nchar+3) do
      begin write(' '); i := i+1; end;
    cursor(23,ncol+1);
    end;
  Protect;
  write(chr(stroke));                     {right margin}
  for i:=ncol+2 to 80 do write (' ');
  Unprotect;
end;
```

```
{-------------------------------------------------------------------}

Procedure Emptylines(n:integer);
{ Writes n empty lines on the screen. }

  var i,j:integer;
  begin
  for i:=1 to n do
    begin
    writeln;                          { Avoid problem when INST CHAR }
    write(chr(esc),'A');              { is depressed. See LineToScreen. }
    if fillwithnulls then
      begin write(chr(esc)); write('K'); end;  { EOL - empty line }
   write('.');

    if not fillwithnulls then
      for j:=2 to ncol do write(' ')
    else
      cursor(23,ncol+1);

    Protect;
    write(chr(stroke));                    {right margin}
    for j:=ncol+2 to 80 do write(' ');
    Unprotect
    end
end;

{-------------------------------------------------------------------}

Procedure Writecomline;
{ Writes the bottom line on the screen and positions the cursor at
  home. }

  var i: integer;
  begin
  Cursor(24,1);

  write(chr(esc)); write('K');    { EOL - empty line }

  for i := 1 to ncol do
    write(comline[i]);

  Home
end;

{-------------------------------------------------------------------}
```

```
Procedure FileToLine(var line:lbuff; var nchar:integer);
{ Reads one line of input file. Nchar (max ncol) char:s are returned
  in line. Tab:s are converted into spaces; form feed is converted
  into '.$' on a separate line. }

  var i,j,k:integer; ch:char;
  begin
  if formfeed then
    begin                                  {indicate form feed}
    line[1]:='.';
    line[2]:='$';
    nchar:=2;
    formfeed:=false;
    end {formfeed}
  else
    begin
    i:=1;
    while (not eoln(infile)) and (i<=ncol) and (not formfeed) do
      begin
      read(infile,ch);
      if ch=chr(ht) then
        begin                              {tab received-convert to spaces}
        j:=i+7-(i-1) mod 8;
        for k:=i to j do line[k]:=' ';
        i:=j+1
        end
      else if (ch=chr(ff)) then formfeed:=true else {remember ff}
        begin
        line[i]:=ch;
        i:=i+1
        end
      end;
    nchar:=i-1;

    if (nchar = 0) then
      begin
      if formfeed then
        begin
        line[1] := '.';  line[2] := '$';
        nchar := 2;
        formfeed := false;
        end
      else
        begin
        line[1] := ' ';
        nchar := 1;
        end;
      end;

    if nchar>=ncol then nchar:=ncol;
    if eoln(infile) then readln(infile) {move to the next line}
    end;
end;

{-------------------------------------------------------------------}
```

```
Procedure LineToFile(var line:lbuff; var nchar:integer);
{ Writes one line on output file. Nchar char:s are written from line.
  Trailing blanks are deleted. A line containing '.$' is converted
  into form feed. }

  var i, pos, spaces: integer;  ch: char;
  begin
  if nchar>0 then
    begin
    while (line[nchar]=' ') and (nchar>1) do nchar:=nchar-1;

    if (nchar=2) and (line[1]='.') and (line[2]='$') then
      write(outfile,chr(ff))
    else if not inserttab then
      begin
      for i:=1 to nchar do write(outfile,line[i]);
      writeln(outfile);                    {write the line}
      end
    else
      begin       { Convert spaces to tabs. }
      pos := 0; spaces := 0;
      for i := 1 to nchar do
        begin
        pos := (pos+1) mod 8;
        ch := line[i];
        if ch <> ' ' then
          begin
          while spaces > 0 do
            begin
            write(outfile,' '); spaces := spaces-1;
            end;
          write(outfile,ch);
          end
        else
          begin
          spaces := spaces+1;
          if pos = 0 then
            begin
            if spaces > 1 then write(outfile,chr(ht))
            else write(outfile,' ');
            spaces := 0;
            end;
          end;
        end;
      writeln(outfile);
      end;
    end
end;

{-------------------------------------------------------------}
```

```
Procedure FileToFile;
{ Transfers one line from infile to outfile. }

  var ch:char;
  begin
  while not eoln(infile) do              {end of line?}
    begin                                {no; transfer one character}
    read(infile,ch);
    write(outfile,ch)
    end;
  writeln(outfile);                {yes;write carriage return/line feed}
  readln(infile)                         {move to the next line}
end;

{-------------------------------------------------------------------}

Procedure LineToBuff(var line:lbuff; nchar:integer);
{ Puts a line with nchar char:s in the text buffer. }

  var i:integer;
  begin
  i:=0;
  while i<nchar do
    begin
    if (last=buffsize) then last:=1 else last:=last+1;
                                         {update buffer pointer}
    i:=i+1;
    buff[last]:=line[i];
    end; {while}
  if (last=buffsize) then last:=1 else last:=last+1;
  buff[last]:=chr(1f)                    {indicate end of line}
end;

{-------------------------------------------------------------------}

Procedure BuffToLine(var line:lbuff; var chnr,nchar:integer);
{ Gets one line from text buffer starting in position chnr.
  Number of char:s returned in nchar.
  Chnr returned pointing at line feed position. }

  var i:integer;
  begin
  i:=1;
  while buff[chnr]<>chr(1f) do
    begin
    line[i]:=buff[chnr];
    if (chnr=buffsize) then chnr:=1 else chnr:=chnr+1;
                                         {update buffer pointer}
    i:=i+1
    end; {while}
  nchar:=i-1;
  if (nchar=1) and (line[1]='.') then nchar:=0
end;

{-------------------------------------------------------------------}
```

```
Procedure BuffToFile(noflines: integer);
{ Outputs the first noflines lines in the text buffer to the output
  file. }

  var chnr,length,n: integer;
  begin
  n:=0; chnr:=first;
  while (n<noflines) do
    begin
    BuffToLine(line,chnr,length);
    LineToFile(line,length);
    n:=n+1;
    if (chnr=buffsize) then chnr:=1 else chnr:=chnr+1;
    end; {while}
  first:=chnr
end;

{-------------------------------------------------------------------}

Procedure ScreenToBuff;
{ Reads the screen into the text buffer. }

  var i:integer;
  begin
  Getblock(23);
  i:=0;
  first:=1; last:=0;                    {initialize buffer pointers}
  repeat
    ScreenToLine(line,nchar);           {read a line}
    i:=i+1;
    LineToBuff(line,nchar);             {and put it into the buffer}
  until (i=23);
end;

{-------------------------------------------------------------------}

Procedure BuffToScreen;
{ Writes the contents of the text buffer on the screen. }

  var chnr,length: integer; eob: boolean;
  begin
  chnr:=first;
  Formoff;
  Cursor(24,1);
  repeat
    BuffToLine(line,chnr,length);
    LineToScreen(line,length);
    eob:=(chnr=last);                       {end of buffer?}
    if (chnr=buffsize) then chnr:=1 else chnr:=chnr+1;
  until eob;
  Formon;
  Writecomline
end;

{-------------------------------------------------------------------}
```

```
      { INITIALIZATION AND ERROR MESSAGE PROCEDURES }
      { ------------------------------------------ }

Procedure Prompt;
{ Writes the prompting character for the file specification command. }

  begin
  Cursor(3,1);
  Protect;
  Write('*');
  Unprotect;
  Formon;
  Cursor(3,2);
end;

{-------------------------------------------------------------------------}

Procedure Initcomline;
{ Initializes the command line buffer. }

  var i: integer;
  begin
  for i := 1 to 80 do comline[i] := '.';
  for i := 1 to 7 do comline[10*i] := chr(stroke);
  end;

{-------------------------------------------------------------------------}

Procedure Initialize;
{ Initializes the screen and the text buffer with empty lines. }

  var i:integer;
  begin
  Formoff;
  Clear;

  cursor(24,1);
  Emptylines(23);
  Formon;
  first:=1; last:=0;                    {initialize buffer pointers}
  for i:=1 to 23 do LineToBuff(line,0);
end;

{-------------------------------------------------------------------------}
```

```
Procedure Writefirstlines;
{ Writes the first 3 lines of a file on the screen. }

  var n: integer;
  begin
  if infilepresent then
    begin
    Formoff;
    cursor(24,1);
    n := 0;
    while (not eof(infile)) and (n<3) do
      begin
      FileToLine(line,nchar);
      LineToScreen(line,nchar);
      LineToBuff(line,nchar);
      n := n+1;
      end;
    Formon;
    end;
  end;

{-------------------------------------------------------------------------}

Procedure Errmess(n:integer);
{ Writes an error message on the bottom line. }

  begin
  Cursor(24,56);
  case n of
    1: write('Illegal function key');
    2: write('Command string error');
    else write('Error') end;
  home;
end;

{-------------------------------------------------------------------------}
```

```
      { COMMAND DECODING PROCEDURES }
      { ------------------------- }

Function Integ(var chnr,int:integer): boolean;
{ Decodes an integer from line starting at chnr. Returns value in int.
  The value of integ is true if an integer was found. }

  var zero,nine,dig,ip:integer; negative:boolean;
  begin
  zero:=ord('0'); nine:=ord('9');
  negative:=false;
  if (line[chnr]='-') then begin negative:=true; chnr:=chnr+1; end;
  int:=0;  ip:=chnr;
  dig:=ord(line[chnr]);
  while (chnr<=nchar) and (dig>=zero) and (dig<=nine) do
    begin
    int:=10*int+(dig-zero);
    chnr:=chnr+1;
    dig:=ord(line[chnr]);
    end;
  if negative then int:=-int;
  integ:=chnr>ip;
end;

{-------------------------------------------------------------------}

Procedure Readcomline(length: integer);
{ Reads the bottom command line into array line.
  Nb. The terminal is left in format off mode. }

  begin
  Formoff;
  Cursor(24,length);
  Getline;
  ScreenToLine(line,nchar);
  if (nchar=0) then nchar:=1;
  while ((line[nchar]='.') or (line[nchar]=chr(stroke))
    or (line[nchar]=tabmark) )
    and (nchar>1) do nchar:=nchar-1;
end;

{-------------------------------------------------------------------}
```

```
Procedure Cominteg(var chnr,int:integer);
{ Decodes an integer, possibly defining a new default value, from line
  starting at chnr. Returns value in int. Newdef is set true if new
  default value is requested. }

  begin
  newdef:=false;
  OK:=true;
  if (line[chnr]='.') or (line[chnr]=' ') then int:=0 else
    begin
    if (line[chnr]=pardelim) then
      begin                                  {new default value}
      chnr:=chnr+1;
      newdef:=true;
      end;
    if not Integ(chnr,int) then
      begin
      OK:=false;
      Errmess(2);
      end;
    end;
end;

{-------------------------------------------------------------------}

Procedure Splitcommand;
{ Decodes the file specification command. }

label 10,20;
var chnr,i:integer;  opt: array[1..5] of char;

function letter(ch:char):boolean;
  var c:integer;
  begin
  c:=ord(ch) and 137B;
  letter:=(c >= ord('A')) and (c <= ord('Z'));
end;

procedure skipspaces(var ip:integer);
  begin
  while line[ip]=' ' do ip:=ip+1;
end;

  begin {Splitcommand}
  fillwithnulls := false;                { Default values }
  inserttab := false;
  infilepresent:=true;
  OK:=true;
  ncol:=79;
  for i:=1 to 14 do
    begin
    inname[i]:=' '; outname[i]:=' ';
    end;
  chnr:=2;                               {skip the prompting char. *}
  i:=1;
```

```
skipspaces(chnr);
if not letter(line[chnr]) then goto 10; {legal file name?}

while (chnr(=nchar) and not (line[chnr]='=') and
    not (line[chnr]='/') and not (line[chnr]='[') do
  begin                                  {read output file name}
  if i(=14 then outname[i]:=line[chnr];
  i:=i+1;
  chnr:=chnr+1;
  end;

lenout:=-1;
if line[chnr]='[' then
  begin                                  {read output file size}
  chnr:=chnr+1;
  skipspaces(chnr);
  if not Integ(chnr,lenout) then goto 10;
  if not (line[chnr]=']') then goto 10;
  chnr:=chnr+1;
  skipspaces(chnr)
  end;
if line[chnr]='=' then
  begin                                  {read input file name}
  chnr:=chnr+1;
  skipspaces(chnr);
  if not letter(line[chnr]) then         {input file present?}
    infilepresent:=false                 {no, indicate}
  else
    begin                                {yes, read the name}
    i:=1;
    while (chnr(=nchar) and not (line[chnr]='/') do
      begin
      if i(=14 then inname[i]:=line[chnr];
      i:=i+1;
      chnr:=chnr+1;
      end;
    end;
  end
else
  inname:=outname;          {input file name same as output file name}

while line[chnr]='/' do                  { Read options }
  begin
  chnr:=chnr+1;
  skipspaces(chnr);
  if letter(line[chnr]) then
    begin
    i := 0;   opt := '     ';
    while (letter(line[chnr])) and (i ( 5) do
      begin
      i := i+1;
      opt[i] := line[chnr];
      chnr := chnr+1;
      end;
```

```
      if (opt = 'NULL ') or (opt = 'null ') then fillwithnulls:=true
      else if (opt = 'TAB  ') or (opt = 'tab  ') then inserttab:=true
      else goto 10;
      end
    else
      begin
      if not Integ(chnr,ncol) then goto 10;
      if (ncol<1) or (ncol>79) then goto 10;
      end;
    skipspaces(chnr);
    end;
  if chnr<(nchar+1) then goto 10;
  goto 20;

10:
  cursor(4,1);
  write('Invalid command');
  Forwon;
  cursor(3,chnr);
  OK:=false;

20:
end;

{-------------------------------------------------------------------}
```

```
      { MISCELLANEOUS ROUTINES }
      { ---------------------- }

Function Match(var string:lbuff; var length,pos:integer): boolean;
{ Checks the occurance of the string string[1..length] in
  line[1..nchar] starting at pos. Returned true if string is found,
  with pos indicating position in line. }

  var n,i,j:integer; found:boolean;
  begin
  found:=false;
  n:=nchar-length;
  i:=pos-1;
  while (i<=(n+1) ) and (not found) do
    begin
    i := i+1;
    j:=0;
    repeat
      j:=j+1;
      found:=(line[i-1+j]=string[j]);
    until (not found) or (j=length);
    end;
  pos:=i;
  Match:=found;
end;

{-------------------------------------------------------------------}

Procedure Copyandclose;
{ Copies the screen and the rest of the infile to the outfile and
  closes the files. }

  begin
  ScreenToBuff;
  BuffToFile(23);
  if infilepresent then
    begin
    while (not eof(infile)) do FileToFile;
    close(infile);
    end;
  close(outfile);
end;

{-------------------------------------------------------------------}
```

```
        { PROCEDURES FOR FUNCTION KEYS }
        { ---------------------------- }

Procedure Open;
{ Opens the files. }

  var n:integer;
  begin
  Formoff;
  Cursor(3,80);
  Getline;
  ScreenToLine(line,nchar);
  writeln;
  write('                              ');       {erase error message}
  line[nchar+1]:=chr(1f);
  Splitcommand;
  lenin:=-25;

  if OK and infilepresent then
    begin
    reset(infile,inname,'DOC',lenin);    {open input file}
    if (lenin=-1) then
      begin                              {file not found}
      Cursor(4,1);
      write('File not found');
      Formon;
      Cursor(3,2);
      OK:=false;
      end;
    end; {if OK}

  if OK then
    begin
    n:=lenout;
    rewrite(outfile,outname,'    ',lenout); {open output file}
    if ((n=-1) and (lenout(lenin+25)) or (lenout(0) then
      begin                              {not enough available space}
      if infilepresent then close(infile);
      Cursor(4,1);
      write('Not enough space');
      Formon;
      Cursor(3,2);
      OK:=false;
      end;
    end; {if OK}

  if OK then
    begin
    filesopen:=true;
    Initialize;
    Writefirstlines;
    Writecomline;
    end;
end;
```

```
{-------------------------------------------------------------------}

Procedure Top;
{ Closes the files and opens new ones with the former output file as
  new input file. }

  var i:integer;
  begin
  Copyandclose;
  inname:=outname;                          {make output file new input file}
  reset(infile,inname,'DOC',lenin);
  infilepresent:=true;
  lenout:=-1;
  rewrite(outfile,outname,'DOC',lenout);  { There should be a check. }
  filesopen:=true;
  Initialize;
  Writefirstlines;
  Writecomline;
end;

{-------------------------------------------------------------------}

Procedure Bottom;
{ Advances to the end of the file. }

  const nlines=10;
  var i:integer;
  begin
  ScreenToBuff;                             {read the screen}
  Formoff;
  Cursor(24,1);
  Emptylines(5);                            {indicate busy}
  BuffToFile(23-nlines);
  while not eof(infile) do
    begin
    BuffToFile(1);
    FileToLine(line,nchar);
    LineToBuff(line,nchar);
    end;
  for i:=nlines+1 to 23 do LineToBuff(line,0);
  BuffToScreen;
end;

{-------------------------------------------------------------------}

Procedure Turn;
{ Advances through the files. }

  var chnr,n,m:integer;
  begin
  Readcomline(55);                          {read command line}
  chnr:=1;
  Cominteg(chnr,n);                         {and decode it}
  if OK and (n<0) then begin OK:=false; Errmess(2) end;
```

```
  Formon;

  if OK then
    begin
    if newdef then defturn:=n;
    if n=0 then n:=defturn;
    ScreenToBuff;                          {read the screen}
    Formoff;
    Cursor(24,1);
    if (n<23) then m:=n else m:=23;
    BuffToFile(m);
    if (n>23) and infilepresent then
      begin
      Emptylines(5);                       {indicate busy}
      while (not eof(infile)) and (n>23) do
        begin
        FileToFile;
        n:=n-1;
        end;
      end; {if (n>23)}
    if infilepresent then
      begin
      while (not eof(infile)) and (m>0) do
        begin
        FileToLine(line,nchar);
        LineToScreen(line,nchar);
        LineToBuff(line,nchar);
        m:=m-1;
        end; {while}
      end; {if infilepresent}
    Emptylines(m);
    Formon;
    Writecomline;
    for n:=1 to m do LineToBuff(line,0);
    end; {if OK}
end;

{-------------------------------------------------------------------}

Procedure Locate;
{ Advances through the files until a specified string is found. }

  var chnr,n,i,pos:integer; found:boolean;
  begin
  Readcomline(55);                         {read command line}
  chnr:=2;
  while (chnr<=nchar) and (line[chnr]<>pardelim) do chnr:=chnr+1;
  length1:=chnr-1;
  OK:=true;
  n:=deflocate;
  if (line[chnr]=pardelim) then Cominteg(chnr,n);
                                           {decode integer parameter}
  if OK and ((n<1) or (n>23)) then
    begin
    OK:=false;
```

```
    Errmess(2);
    end;
  Formon;

  if OK then
    begin
    deflocate:=n;
    for i:=1 to length1 do string1[i]:=line[i]; {read string}
    ScreenToBuff;                              {read the screen}
    found:=false;
    Formoff;
    Cursor(24,1);
    Emptylines(5);                             {indicate busy}
    BuffToFile(23-n);
    while (not eof(infile)) and (not found) do
      begin
      BuffToFile(1);
      FileToLine(line,nchar);
      LineToBuff(line,nchar);
      pos := 1;
      found:=Match(string1,length1,pos);   {look for the string}
      end;
    while (not eof(infile)) and (n<23) do
      begin
      FileToLine(line,nchar);
      LineToBuff(line,nchar);
      n:=n+1;
      end;
    for i:=n+1 to 23 do LineToBuff(line,0);
    BuffToScreen;
    if found then Cursor(deflocate,pos);   {if found, indicate where}
    end; {if OK}
end;

{-------------------------------------------------------------------}

Procedure Convert;
{ Replaces a string with a new one throughout the file. }

  var chnr, pos1, pos2, i, nch: integer;   delim: char;

  procedure writechar(ch: char);

    begin
    write(outfile,ch);
    if nch <= ncol then write(ch);
    end;

  begin
  length1:=0; length2:=0;
  Readcomline(55);                           {read command line}
  delim:=line[1];                            {get delimiter}
  chnr:=2;
  while (chnr<=nchar) and (line[chnr]<>delim) do
    begin                                    {read the first string}
```

```
    length1:=length1+1;
    string1[length1]:=line[chnr];
    chnr:=chnr+1;
    end;
  chnr:=chnr+1;
  while (chnr<=nchar) and (line[chnr]<>delim) do
    begin                                    {read the second string}
    length2:=length2+1;
    string2[length2]:=line[chnr];
    chnr:=chnr+1;
    end;
  OK:=true;
  if (length1=0) or (chnr<>nchar) or (line[chnr]<>delim) then
    begin                                    {error}
    OK:=false;
    Errmess(2);
    end;
  Formon;

  if OK then
    begin
    ScreenToBuff;                            {read the screen}
    Formoff; clear;
    BuffToFile(23);
    while not eof(infile) do
      begin
      FileToLine(line,nchar);
      pos2 := 1;
      if Match(string1,length1,pos2) then    {look for the string }
        begin                                {found - replace it}
        nch := 0;
        for i:=1 to pos2-1 do writechar(line[i]);
        for i:=1 to length2 do writechar(string2[i]);
        pos1 := pos2+length1;
        pos2 := pos1;
        while Match(string1, length1, pos2) do
          begin
          for i := pos1 to pos2-1 do writechar(line[i]);
          for i := 1 to length2 do writechar(string2[i]);
          pos1 := pos2+length1;
          pos2 := pos1;
          end;

        for i:=pos1 to nchar do writechar(line[i]);
        writeln(outfile); writeln;
        end
      else LineToFile(line,nchar);          {not found}
      end; {while}
    Initialize;
    Writecomline;
    end; {if OK}
end;

{-------------------------------------------------------------------}
```

```
Procedure Insdel;
{ Inserts empty lines and deletes line feeds on the screen. }

  var i,j,n,noflines:integer; ch:char;
  begin
  Readcomline(55);                          {read the command line}
  OK:=(nchar=1) and (line[1]='.');          {no command line}
  if not OK then
    begin                                   {command line present}
    OK:=(nchar)=3) and (line[1]=pardelim) and (line[2]<>line[3]);
    if OK then begin insert:=line[2]; delete:=line[3] end;
    end;
  if not OK then Errmess(2);
  Formon;

  if OK then
    begin
    Getblock(23);                           {read the screen}
    i:=0; n:=0; noflines:=0;
    first:=1; last:=0;                      {initialize buffer pointers}

    repeat
    ScreenToLine(line,nchar);               {read one line}
    i:=i+1;                                 {number of read lines}
    for j:=1 to nchar do if (line[j]=insert) then line[j]:=chr(1f);
                                            {insert lf where indicated}
    if nchar>0 then
      begin
      if (line[nchar]=delete) then line[nchar]:=' '
        else begin nchar:=nchar+1; line[nchar]:=chr(1f) end;
      end;                                  {delete lf if indicated}

    for j:=1 to nchar do
      begin
      ch:=line[j];                          {read one character}
      last:=last+1;                         {update buffer pointer}
      buff[last]:=ch;
      n:=n+1;
      if (ch=chr(1f)) then
        begin n:=0; noflines:=noflines+1 end
      else if (n=ncol) and ((j=nchar) or (line[j+1]<>chr(1f))) then
        begin                               {page width reached - insert}
        last:=last+1;                       {line feed}
        buff[last]:=chr(1f);
        n:=0;
        noflines:=noflines+1;
        end;
      end; {for}
    until (i=23);

    for i:=noflines+1 to 23 do LineToBuff(line,0);
    clear;
    cursor(24,1);
    BuffToScreen;
    Cursor(24,1);                           {indicate busy}
```

```
    BuffToFile(noflines-23);
    Home;
    end; {if OK}
end;

{-----------------------------------------------------------------}

Procedure Verify;
{ Reads the screen and rewrites its contents. Fills end of lines
  with spaces. }

  var i,j: integer;  savefillstatus: boolean;
  begin
  savefillstatus := fillwithnulls;
  fillwithnulls := false;
  Getblock(23);
  i:=0; j:=0;
  first:=1; last:=0;
  repeat
  ScreenToLine(line,nchar);
  i:=i+1;
  if (nchar)0) then
    begin
    LineToBuff(line,nchar);
    j:=j+1;
    end;
  until (i=23); {end repeat}
  for i:=j+1 to 23 do LineToBuff(line,0);
  clear;
  cursor(24,1);
  BuffToScreen;
  fillwithnulls := savefillstatus;
end;

{-----------------------------------------------------------------}

Procedure Settab;
{ Reads a command line with tab positions. }

  begin
  readcomline(ncol);
  if nchar ) 1 then
    Errmess(2)        { other characters than dot, stroke or tabmark. }
  else
    comline := line;
  Formon;
  home;
  end;

{-----------------------------------------------------------------}
```

```
Procedure Tab;
{ Performs tab operation. }

  var pos, tabpos, i: integer; ch: char;
  begin
    { Get cursor position. }
  getline;
  pos := 0;
  while not eoln do
    begin
    read(ch);
    pos := pos+1;
    end;
  readln;

  if pos<ncol then
    begin

      { Get position for next tab mark. }
    tabpos := pos+1;
    while (tabpos < ncol) and (comline[tabpos] <> tabmark) do
      tabpos := tabpos+1;

      { Perform tab operation. }
    for i:=pos to tabpos-1 do
      write(' ');
    write(' ');
    write(chr(10B));    { Backspace }
    end;

  end;

{-------------------------------------------------------------------}

Procedure Recover;
{ Writes the contents of the text buffer on the screen. }

  begin
  clear;
  cursor(24,1);
  BuffToScreen;
end;

{-------------------------------------------------------------------}

Procedure Closefiles;
{ Closes the files. }
  begin
  Copyandclose;
  filesopen:=false;
  Formoff;
  Clear;
  Prompt;
end;
```

```
{-------------------------------------------------------------------------}

Procedure Cexit;
{ Closes the files and terminates the program. }
  begin
  if filesopen then Copyandclose;
  Formoff;
  Clear;
  writeln('Reset switch on rear.');
  exitrequest:=true;
end;

{-------------------------------------------------------------------------}

Procedure Cancel;
{ Closes the input file only to annul all editing performed. }

  begin
  if infilepresent then close(infile);
  filesopen:=false;
  Formoff;
  Clear;
  writeln('Reset switch on rear.');
  exitrequest:=true;
end;

{-------------------------------------------------------------------------}
```

```
begin {PAGED}

exitrequest:=false; filesopen:=false; formfeed:=false;
defturn:=20; deflocate:=10;  {default values}
insert:=chr(stroke); delete:=chr(backsl);
Initcomline;
Clear;
writeln('PAGED  Version ', version:2);
writeln('Set switch on rear in position "block".');
Prompt;

repeat
if not filesopen then
  case Readfunc of
   1: Open;
  13: Cexit;
  else Errmess(1)
  end {case}
else
  case Readfunc of
   2: Top;
   3: if infilepresent then Bottom else Errmess(1);
   4: Turn;
   5: if infilepresent then Locate else Errmess(1);
   6: if infilepresent then Convert else Errmess(1);
   7: Insdel;
   8: Verify;
   9: Settab;
  10: Tab;
  11: Recover;
  12: Closefiles;
  13: Cexit;
  16: Cancel;
  else Errmess(1)
  end; {case}
until exitrequest;

end. {PAGED}
```