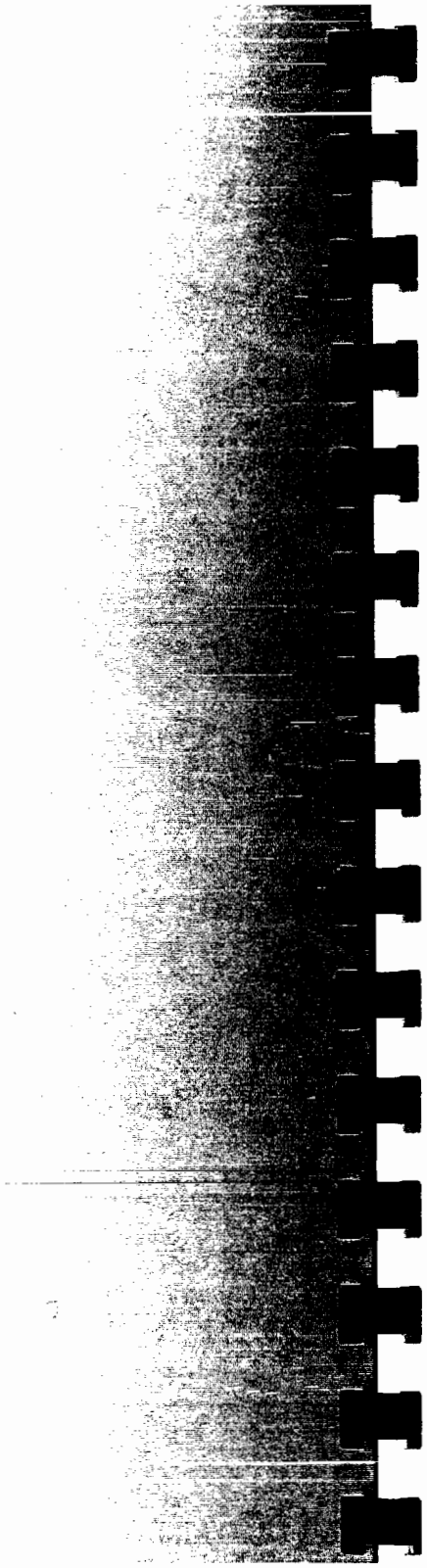
Editor Toolkit

Hawthorne Technology



EDITOR TOOLKIT

for K-OS ONE

DISTRIBUTION PACKAGE

Hawthorne Technology
8836 S. E. Stark
Portland, OR 97216

(503) 254-2005

* * * PRODUCT DISCLAIMER * * *

This software and manual are sold 'as is' and without warranties as to performance or merchantablility. These programs are sold without any express or implied warranties. No warranty of fitness for a particular purpose is offered. The user must assume the entire risk of using the programs and is advised to test the programs thoroughly before relying on them.

Any liability of seller or manufacturer will be limited exclusively to product replacement or refund of the purchase price.

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

EDITOR TOOLKIT

TABLE OF CONTENTS

INTRODUCTION

This toolkit provides you with two working editors that you can use as they are provided, or use as a base to build on. You can add a feature or two at a time and have a functional editor to test the features out on.

The line editor is a generic line editor. It is like the editors that are common on multiuser mini and mainframe computers. The main advantage with a line editor is that it doesn't have to be customized for any particular terminal but can be used by almost anyone just as it comes. It is also far easier to produce a good smooth line editor on a multiuser system than it is to provide a screen editor for multiuser applications.

The command set for the screen editor is similar to the commands of Word Star or Turbo Pascal. This is because we were using both of these editors on an MS/DOS machine while developing the editors for the K-OS ONE system. Because the editor command set is determined by a case statement, it is easy to change it to look like almost any full screen editor. The editors as provided with this package are setup for the Wyse-30 terminal. They can be changed for other terminals by changing the low level words that do the cursor control and screen control.

The text formatter is also a generic type formatter for general text use. In most cases, it follows the text format routine given in the book Software Tools. The text formatter is easy to customize to your own particular needs.

EDITOR STRUCTURE

For any text editor, either line or screen, there are certain basic sections. There is a text buffer that must be managed and manipulated. There is the screen that has to be manipulated. Finally there is the command decoder that determines what will get done next.

A large portion of each of these editors was written in HTPL. The readability of HTPL makes the code easy to modify. A few of the low level routines are in 68000 assembly code to make these editors run faster than they would if they had been coded totally in HTPL. In general a mix of HTPL and assembly language yields a good compromise between cost of development, and speed of operation. The parts that are used the most are optimised for speed and portions that are used less often are optimised for size.

Both of the editors in this package are line oriented. The line editor uses lines directly while the screen editor uses them indirectly. One advantage of being line oriented is the ability to restore a line to its prior condition if editing errors are made. It also cuts the amount of text movement that is needed when characters are inserted and deleted.

Most of the routines have a stack diagram with them on the line where they are defined. This is done because an RPN language like HTPL uses the stack to pass arguments to a procedure and to return results to the calling routine. The items to the left of the dashes tell what the routine expects on the stack when it is entered. The items to the right of the dashes is what the routine will return to the caller. Many routines do not receive anything on the stack and many routines do not return anything on the stack.

HAWTHORNE TECHNOLOGY

LINE EDITOR USER'S MANUAL

Table Of Contents

OPERATING PROCEDURES

This editor has two major modes of operation, ENTER mode and COMMAND mode. Each line of data or command entered is activated by pressing the RETURN key. The backspace key can be used in edit mode to back up and re-work part of or all of a line. This is only good for the current line, so it must be used prior to pressing the return key. Changes on other lines must be made from command mode.

All data is entered from enter mode. The commands EN enter, IN insert, and AP append get you into enter mode. From enter mode, all data is added to the file until a line where the only charactor on the line is a '.' (period) is encountered.

On leaving enter mode, you will be in command mode. Using edit commands while in enter mode will result in the commands being entered as part of the file.

Command mode is designated by a prompt: '*', on the left margin. In command mode the edit commands are used to view or modify the file.

You can enter the editor with a file specified, or if you wish to name the file later, you can enter the editor with no file name.

```
EDITL
```

This will load the line editor and prompt for a command. At this point you can open an existing file or enter data for a new file.

```
EDITL filename.ext
```

This will load the line editor and open the requested file.

COMMAND MODE OPERATION

IMPORTANT NOTES:

1. In Command Mode, all two letter commands must be terminated with a Carriage Return (CR).

2. Commands are not case sensitive. They may be entered either upper or lower case letters.

3. Commands that require a parameter must have one space between the command and the parameter. If no parameter is entered, a default value that is equal to the previous condition or a value of 1 is used.

4. Commands that work with strings of charactors use delimiters to define the limits of the string. A delimiter can be any non-alpha/numeric charactor that is not contained in the text of the string itself.

5. Many of the commands function relative to the line that is considered current. The 'Current Line Pointer' is always pointing to the current line. If the CLP is at line 1, and you PRINT (display) 25 lines, the CLP is not affected. It still points to line 1. Only commands that are specified for moving the CLP will affect its position.

DESCRIPTION OF EACH COMMAND

DATA ENTRY COMMANDS

Append Command = AP

AP (return) - The append command gets you into ENTER mode. The data you enter is added at the end of the file. You remain in ENTER mode until a line with nothing but a . (period) is entered.

Enter Data Command = EN

EN (return) - The enter command gets you into ENTER mode. Each line you type while in ENTER mode is put after the last existing line. A line is completed when you press the return key. You remain in ENTER mode until a line with nothing but a . (period) is entered.

Insert Command = IN

IN (return) - The insert command gets you into ENTER mode. The lines you enter are inserted directly following the 'current' line in the file. The insertion is terminated by entering a line wiht nothing but a . (period) on it.

MOVING CURRENT LINE POINTER (CLP)

Top Command = TO

TO (return) - Positions the CLP at top of the file and displays the first line.

Bottom Command = BO

BO (return) - Positions the CLP at last line in the file, and displays it.

Here Command = HE

HE (return) - Display the current line.

Down Command = DO

DO n - Moves the pointer down "n" lines and displays the line. (Default is one line.)

Up Command = UP

UP n - Moves the pointer up "n" lines and displays the line. (Default is one line.)

Find Line Command = LN

LN n - Moves pointer to line number "n".

PRINTING (DISPLAY)

Print — Command = PR

PR - Prints the entire contents of file.

PR n - Displays "n" lines, starting with the next line after the current line.

FIND DATA

Find Data — Command = FI

FI /string/ - Find the group of characters specified by "string" starting from CLP +1. The string specified in the command line must be set off using delimiters. The line on which the "string" is found is displayed. The CLP is now at the found line.

Find Next — Command = FN

FN (return) - Find the next occurrence of the "string" that was specified in the prior Find Data (FI) command. This "find" starts at CLP + 1 and displays the line on which the string was found.

Find All Occurrences — Command = FA

FA /string/ - Find all occurrences of the "string" in the entire file and display each line on which the string was found. This can be useful in creating a cross reference of particular items in your file.

EDITING COMMANDS

Copy — Command = CO

CO begin,end - Copies the lines "begin" through "end" on to the spot just before the current line, where begin is the number of the first line to be copied and end is the last line to be copied.

Delete Lines — Command = DE

DE n - Deletes "n" lines (default 1) starting with the current line.

Move — Command = MO

MO begin,end - Moves the lines "begin" through "end" to just before the current line, where begin is the number of the first line to be moved and end is the number of the last line to be moved.

New Text — Command = NE

NE (return) - Deletes all old text and restarts the editor. Clears the buffer for new text to be entered.

Replace Current Line — Command = RE

RE (return) - Replaces the current line with the next line you enter.

Truncate Last Lines — Command = TR

TR (return) - Deletes all lines after the current line.

CHANGING STRINGS

Change String(s)

Command = CH

CH /old/new/n - The characters in the new string replace those in the old string. If no value for "n" is given, the change takes place only on the current line. If a value is given for "n", the change will be made for every occurrence of the old string, within "n" lines. To delete a string of characters, instead of specifying a "new" string, use two consecutive delimiters i.e. "//". Any valid delimiter, (a non alpha-numeric character), can be used in place of "/". All changed lines will be displayed.

Caution: The charactor used as a delimiter can not be used in old or new text string.

NOTE: If the substitution applies to all occurrences in an entire file, start at the top of the file and use a number for "n" that is larger than the last line number in the file.

Change By Position

Command = CC

CC c/new/n - The "c" indicates the column number in the line at which point "new" data will start overlapping the original data. Change starts on the current line and changes "n" lines. All of the lines that are changed will be displayed.

ADMINISTRATIVE COMMANDS

Open File

Command = OP

OP - Opens a disk file, reads it into memory, and then closes it. This command is used when you want to edit a file that already exists on your disk.

Save File

Command = SA

SA filename - Saves the file on disk. A copy of your file is saved on disk. If the file is already on the disk, the copy you have been working on will replace the old copy on the disk.

Save Same

Command = SS

SS - Saves the file back to the disk using the same file name that you opened earlier. The copy you have been working on will replace the old copy on the disk.

Exit Editor

Command = XX

XX - Exits the line editor and returns command to the operating system.

A TABLE OF EDITOR COMMANDS

DATA ENTRY
- AP - Append
- EN - Enter Data
- IN - Insert
- . - Exit Enter Mode

MOVE CURRENT LINE POINTER
- BO - Bottom
- DO - Down
- HE - Here
- LN - Goto Line
- TO - Top
- UP - UP

PRINTING (DISPLAY)
- PR - Print

FIND STRING
- FA - Find All Occurrences
- FI - Find Data
- FN - Find Next

EDITING COMMANDS
- CO - Copy
- DE - Delete Lines
- MO - Move
- NE - New Text
- RE - Replace Current Line
- TR - Truncate File Here

CHANGING STRINGS
- CC - Change by Position
- CH - Change String

ADMINISTRATIVE COMMANDS
- OP - Open File
- SA - Save File
- SS - Save Same
- XX - Exit Editor

ALPHABETICALLY:
- AP - Append
- BO - Bottom
- CC - Change by Position
- CH - Change String(s)
- CO - Copy
- DE - Delete Lines
- DO - Down
- EN - Enter Data
- FA - Find All Occur.
- FI - Find Data
- FN - Find Next
- HE - Here
- IN - Insert
- LN - Goto Line
- MO - Move
- NE - New Text
- OP - Open File
- PR - Print
- RE - Replace Line
- SA - Save File on Disk
- SS - Save Same
- TO - Top
- TR - Truncate File Here
- UP - UP
- . - Exit Enter Mode

HAWTHORNE TECHNOLOGY

LINE EDITOR MODIFICATION MANUAL

Table Of Contents

LINE EDITOR OVERVIEW

There are several major sections of code in the line editor. At the highest level is the main control routine that gets the next line and decodes the command. The next level is composed of the routines that execute each command. After that is a level that is composed of routines used by the first level. At the very bottom are the run-time library routines and a few very simple utilities.

Most of the editor is written in HTPL. There are a few routines that are written in assembler and are in the runtime library HTPLRTL.HEX. By putting these few routines in assembler the overall speed of the editor has been increased significantly.

Because this is a line editor there is no need to customize it for any given terminal. This is the main reason that a line editor is supplied with the K-OS ONE operating system. This version of the editor has a few new commands in addition to the commands in the editor that comes with the system. It is complete and ready to use.

The first thing that the editor does is to setup space for the local variables. All local variables are allocated from the heap of memory left over after the program is loaded. Then the initialize routine sets reasonable initial values for most of the variables to indicate an empty buffer, but one that can have text added to it.

The decode routine gets a line from the console. It then gets the first two characters and changes them to upper case. A large case statement is used to decode the command. If aliases are wanted for commands two different routines can call the same action routine. If no routine matches in the case statement then an error is given. After a routine is executed, control returns to the top of the loop and the process starts all over again. There is a reset command inside the main loop to catch any errors that may result from unbalanced stack operations. In most cases this won't have any effect, but if there is an error, it will keep the editor from blowing up.

An editor on a PC was used to write the first editor for K-OS ONE, a screen editor on a PC was also used until we had a working screen editor to use native on K-OS ONE. We wrote all the routines first in HTPL and then recoded some of them into assembler to make the editor faster. The editor was not written all at one time. We devised a general structure and defined the data structures we would be working with. Each command was added and tested as a separate piece of code. The generic structure of the commands is similar to many simple line editors that were written for minicomputers in the last 20 years but is not identical to any of them.

LINE EDITOR ROUTINES

The first group of routines move the current line pointer and change the current line number. these are:

```
here    -- where am I
topper  -- go to first line
bottom  -- go to last line
uperlin -- up one line
downer  -- down one line
linner  -- go to line
```

The first three of these check for an empty buffer and call a lower level routine. The last three of these all get a number parameter, check it or set a default, and then use a common line movement routine to get to the target line. Up and down are special cases of the random line movement. The common routine checks for an empty buffer and for running off the end of the buffer or trying to go before the first line.

The delete routine removes lines from the edit buffer. It has to check for special cases. If the buffer becomes empty it notifies the operator. If all lines after the current line get deleted then it tells the operator that the buffer has been truncated.

The two change routines both come in two parts. The first part is the outer loop that gets a parameter or description of the change. Each then calls a simpler routine that changes just one line. Both of these then call a cleanup routine to delete trailing blanks and return the line to the edit buffer.

The find commands are in effect three front ends to a common find routine. FI epects to be given a string to look for. FN assumes that a prior FI was done and uses the same string. FA expects to be given a string to look for and then calls the find routine until it reaches the end of the text buffer.

The text is stored as a set of lines. The first group of subroutines manipulates these lines and the pointers. The lines are refered to by line number and the line numbers can be displayed with each line. The line numbers are not stored. When going up or down a line there are two variables that are modified, the current line number curnum, and the pointer to the current line curlin. The first and last lines in the buffer have special pointers for them.

```
in.line -- insert line in buffer
ap.line -- append line to text buffer
re.line -- replace line
de.line -- delete line
up.line -- move up one line
do.line -- move down one line
sizline -- determine size of line in inbuf
```

The primary commands call on these routines to move around the text buffer.

There are two routines that interact with the console. One of these is getlin. Its purpose is to get the next line from the console. A simple multiple character system call cannot be used for this because of the need to expand tabs as they are entered. The line is left in inbuf where it is manipulated by primary commands.

The other major console routine is pline. Its purpose is to display the current line on the console. The major complication in this routine is the need to expand tabs to spaces as the line is displayed. This routine also displays the current line number.

IDEAS FOR CHANGES

The routine that displays the current line always displays the line number with it. A pair of routines that turn line numbering on and off is nice sometimes. I have used this on prior editors of this sort and found that it is best to default to number turned on.

A command to echo all lines displayed to the printer or a file would almost turn this simple line editor into a useful filer or data handing program. The find all command acts like a select function in a data base. A special character can be used to encode special fields in each line.

A help routine could easily be added. Look at the help routine in the screen editor. After entering the routine it is just a large bunch of writeln calls to display text. The help command in the command processor for K-OS ONE is the same way.

If the editor is used a lot then some of the general purpose routines from the K-OS ONE command processor could be added. If part of the editor is deleted it could even be combined with the command processor. On a Tiny Giant HT68k board the boot load routine in prom will load anything called command.bin as the command processor. This would almost make it a dedicated editing machine.

Two commands that would be useful are SP, to split a line into two separate lines and JN, to join two separate into a single line. With the present editor you have to copy a line and delete part of it to split it. There is no easy way to join lines with the current editor.

If the command processor syntax is changed then the editor can be made to look more like TECO or like other line editors. To emulate another editor the command part that the user sees can be changed but the underlying routines that do all the work can be left the same.

On most word processors if you are typing in text the editor will generate an automatic return at the end of a line. To do this you count the spaces that are used up on a line and after the max line length is reached a return is generated and a new line is started. If this is done a command to set the right margin would be very helpful.

NOTE:

When making a new version of the editor it is important to make a backup copy of the source and object of the current editor. This way if a disaster happens you can always go back to the last version that worked. Changes should be made one at a time if possible and kept as isolated as possible from existing commands to minimize the amount of debugging that needs to be done.

VARIABLES USED IN THE LINE EDITOR

These are message strings that are used in more than one place in the editor. They are placed here to save space so they won't have to be duplicated.

```
byte eobmes = "END OF BUFFER"
byte tobmes = "TOP OF BUFFER"
byte mtbmes = "EMPTY BUFFER"
byte delmes = "LINE(S) DELETED"
byte nofiln = "NO FILE NAME SPECIFIED"
byte cantop = "CANNT OPEN FILE"
byte nfndms = "STRING NOT FOUND"

long texbuf - pointer to start of text
long nextline - end of text pointer
long finish -
long block -
long lines - number of lines in buffer
long curlin - pointer to current line
long curnum - current line number
byte filnam - temporary file name,
              uses space from rplstr
byte rplstr [ 130 ]  - string to replace with
byte scrstr  [ 130 ] - string to search for
byte workfil [ 40 ] - file being worked on
long scpn - scan pointer for argument fetch
long fndflg - string found flag
long count - a general counter
long start -
long nsave - save place for curnum
long psave - save place for curlin
long position -
long strpnt -
long cursiz -
long dest -
long mvcnt -
long startpn -
long finishpn -
long chngflg -
long repcount - repeat count
long tabpn - tab table pointer
long lnpnt - line pointer
long bfpnt - buffer pointer
long rplpnt - replace pointer
long max -
long lxpn - print out pointer
long tbcnt - tab expand counter
long thsfile -
```



```
long infile - input channel
long outfile - output channel
lcng chrcnt - character counter
long active - disk line active
long dskbfpn - pointer to disk buffer
long frstime - restart flag
word  paramblk [ 10 ] - parameter block  used
                        for system calls
byte inbuf [ 150 ] - line input buffer
byte delim - string delimiter
byte chx - current char
byte tabtab [ 20 ] - table of tab values
```

COMMAND SUMMARY

This is a summary of the commands in the line editor and the first level routines that perform each of the commands. In the editor program this is found in the form of a large case statement.

Command	Routine	Comment
$0D00		IGNORE
' '		IGNORE
'AP'	append	APPEND
'BO'	bottom	BOTTOM OF FILE
'CC'	changc	CHANGE BY POSITION
'CH'	changs	CHANGE BY CONTEXT
'CO'	copper	COPY
'DE'	deletr	DELETE
'DO'	downer	DOWN
'EN'	append	ENTER
'FA'	findal	FIND ALL
'FI'	finder	FIND LINE
'FN'	findex	FIND NEXT
'HE'	here	HERE
'IN'	linnrt	INSERT
'LN'	linner	GOTO LINE
'MO'	moveln	MOVE
'NE'	abort	NEW TEXT
'OP'	openfil	OPEN FILE FOR EDIT
'PR'	printr	PRINT
'RE'	replar	REPLACE
'SA'	savit	SAVE FILE
'SS'	savsam	SAVE TO SAME FILE
'TO'	topper	TOP OF FILE
'TR'	truncate	TRUNCATE FILE HERE
'UP'	uperlin	UP
'XX'	quit	EXIT EDITOR

PROCEDURES

This is a summary of the procedures used by the line editor:

PERET $2E0D - actually a constant
setup - setup to start editor
cmdlin - process the command line

********** MAIN LEVEL ROUTINES **********

quit	- XX -- EXIT EDITOR
here	- HE -- HERE --- WHERE AM I ???
topper	- TO -- TOP OF FILE
bottom	- BO -- BOTTOM OF FILE
uperlin	- UP -- MOVE POINTER UP
downer	- DO -- MOVE POINTER DOWN
linner	- LN -- GOTO LINE
lfind	- general move to line routine
deletr	- DE -- DELETE
truncate	- TR -- TRUNCATE FILE
replar	- RE -- REPLACE LINES
printr	- PR -- PRINT
moveln	- MO -- MOVE LINE OR BLOCK
copper	- CO -- COPY LINE OR BLOCK
copmov	- common code for move and copy
changs	- CH -- CHANGE LINE ---- BY CONTEXT
changlin	- repeating part of ch
chcomper	- compare possible match - update bfpnt
changc	- CC -- CHANGE LINE ---- BY POSITION
chcllrep	- repeating part of cc
colchng	- used by chcllrep
chfix	- common ending for change routines
chop	- chop trailing blanks
getrepcnt	- get repatition count
linnrt	- IN -- INSERT LINES BEFORE CURRENT
append	- AP -- APPEND LINES

PROCEDURES (continued)

```
savit       - SA -- SAVE FILE
savsam      - SS -- SAVE TO SAME FILENAME
wrfilnam    - write buffer to file in FILNAM
openfil     - OP -- OPEN FILE FOR EDITING
rdopened    - read opened file into buffer

findal      - FA -- FIND ALL LINES WITH STRING
finder      - FI -- FIND STRING
findex      - FN -- FIND NEXT OCCURANCE OF STRING
bufemt      - check for empty buffer and abort
```


SECOND LEVEL SUBROUTINES

These are second level general subroutines used by many of the first level primary routines.

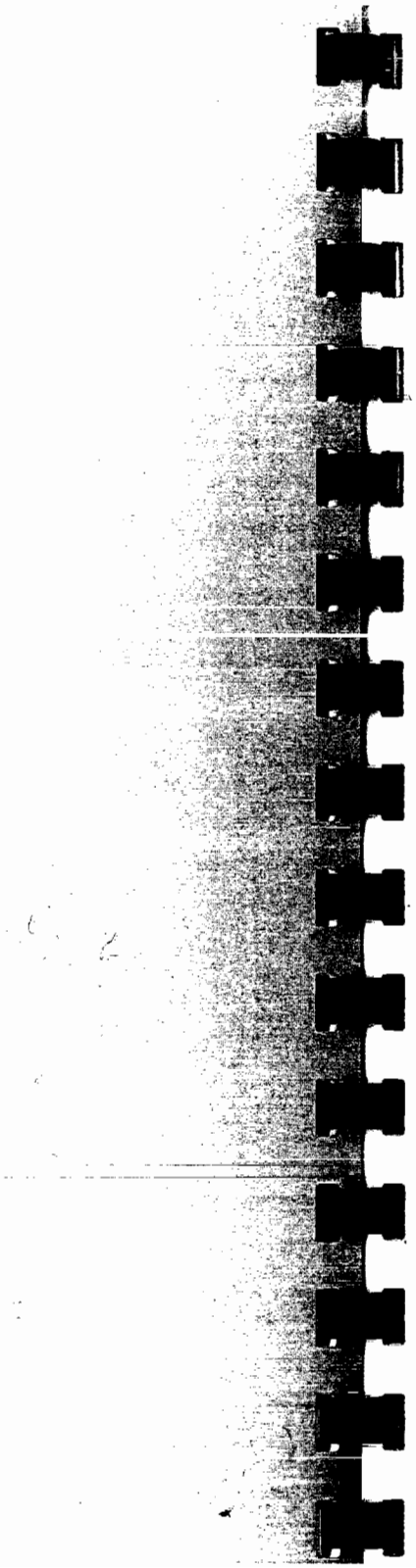
```
in.line     - insert line in texbuf
ap.line     - append line to buffer
re.line     - replace line in buffer
de.line     - delete line in buffer
up.line     - go up one line in buffer
do.line     - go down one line in buffer
sizline     - determine size of inline and nul it

getprm      - get numeric parameter
X10         - multiply by 10
nxtch       - get next char from buffer
skipbl      - skip leading blanks
blanklin    - fill a buffer with spaces
getstr      - get search string
getrpl      - get replace string
gstring     - get a string
getnam      - get file name and put in buffer
gcmdstr     - get command line
openit      - open file if possible
opfilnam    - open file in filnam
rewritef    - rewrite file in filnam
closeit     - close file - unit on stack
fndexx      - find string in buffer
              true found, false end of buffer
```

MAIN I/O ROUTINES

These are the main I/O routines used to get text into and out of the text buffer. For more information on how the calls for K-OS services work consult the K-OS ONE programmers manual.

```
getlin     - get next line from console
keepit     - keep a char from an input
backup     - backspace in field
backchr    - back space console
readln     - get linef from disk file
fgetc      - get character from disk file
savlin     - send current line to disk buffer
fputc      - send one char to disk
putblk     - write current block or part to disk
pline      - print current line with line number
putcx      - put char using lxpn
writeln    - write string to console w/crlf
crlf       - send cr and lf to console
continue   - " PRESS ANY KEY TO CONTINUE >> "
upcase     - convert lower to upper case
scopy      - string copy, does not copy final nul
```



HAWTHORNE TECHNOLOGY

SCREEN EDITOR USER MANUAL

Table of Contents

SCREEN EDITOR

Introduction

A full screen editor is something that many K-OS ONE users have asked for. We are providing the source code for the screen editor so it can be easily customized for any terminal. This package supplies the tools you will need to create YOUR editor. You can change the default settings or the commands to be what you like. You can add commands to do special functions.

COMMAND SET FOR SCREEN EDITOR:

```
-------- FOR HELP SCREEN ^J or Line Feed --------
 ^E - line up             | ^G - del char
 ^X - line down           | ^T - del word
 ^R - page up             | ^L - repeat last find
 ^C - page down           | ^Y - delete line
 ^D - char right          | ^A - word left
 ^S - char left           | ^F - word right
-------------------------------------------------
 ^QB - to start of block  | ^KB - mark start of blk
 ^QK - to end of block    | ^KK - mark end of block
 ^QS - start of line      | ^KR - read file
 ^QD - end of line        | ^KW - write file
 ^QR - top of file        | ^KD - exit editor
 ^QC - bottom of file     | ^KC - copy block
 ^QA - find and replace   | ^KV - move block
 ^QF - find               | ^KY - delete block
 ^QL - restore line          ** block markers are
 ^QY - del to end of line          always column 1
-------------------------------------------------
```


USING THE SCREEN EDITOR

To start a new file using the screen editor, you invoke the editor giving no file names. With the EDITS.BIN program in the default drive, type:

```
EDITS (return)
```

The editor will be loaded and you will start with a clear screen.

To edit an existing file you can specify the file name on the command line:

```
EDITS filename.ext
```

You can also read the file into the edit buffer after entering the editor:

```
EDITS
^KR  (prompts 'FILE NAME':)
filename.ext
```

While using the editor, any time that you get the message: PRESS # TO CONTINUE, pressing any key will allow you to resume editing.

EDIT COMMANDS

The following is a description of each edit command that is supplied with the screen editor in the edit toolkit.

In describing these commands, a ^ is used to indicate a control character, where the control key is held down while the character is pressed.

HELP COMMAND

^J HELP

This command causes a list of the edit commands to be desplayed on the console terminal. Pressing any key will restore the screen and allow you to resume editing.

LINE FEED HELP

This command acts the same as ^J

MOVE CURSOR

These commands are used to position the cursor on the screen.

^E - line up
^X - line down
^D - char right
^S - char left
^A - word left
^F - word right
^QS - start of line
^QD - end of line
^QR - top of file
^QC - bottom of file
^QB - to start of block
^QK - to end of block

^E Line Up

Moves the cursor to the line just above the line it was on. The cursor will remain in the same column regardless of the length of the line.

^X Line Down

Moves the cursor to the line just below the line it was on. The cursor will remain in the same column regardless of the length of the line.

MOVE CURSOR (continued)

^S Character Left

Moves the cursor to the left one character.

^D Character Right

Moves the cursor to the right one character.

^A Word Left

Moves the cursor to the left one word. The cursor is moved to the left most character of the group of characters to the left of it's original position.

^F Word Right

Moves the cursor to the right one word. The cursor is moved right, to the first charactor following a space.

^QS Start of Line

Moves the cursor to the first column of the line it is on.

^QD End of Line.

Moves the cursor to the end of line it is on. This puts the cursor just past the last character of the line.

^QR Top of File

Moves the cursor to the first character at the top of the file. The screen will change to show the first screen of the file.

MOVE CURSOR (continued)

^QC Bottom of File

Moves the cursor to the last charactor at the bottom of the file. The screen will change to show the last screen of the file.

^QB To Start of Block

Moves the cursor to the start of the marked block.

^QK To End of Block

Moves the cursor to the end of the marked block.

MOVE SCREEN - SCROLL

These commands are used to scroll through the text by displaying the next screen requested.

^R Scroll Up

Moves the display so the next screen up towards the beginning of the file is displayed.

^C Scroll Down

Moves the display so the next screen down towards the end of the file is displayed.

DELETE AND RESTORE

These commands are used to delete items from your file.

^G - delete character
^T - delete word
^Y - delete line
^QY - delete to end of line
^QL - restore line

^G Delete Character

Deletes the character that the cursor is on.

BACKSPACE or DEL Delete Character Left

The Backspace or DEL (Delete) keys will cause the charactor to the left of the cursor to be deleted.

^T Delete Word

Deletes the word to the right of the cursor, starting with the charactor the cursor is on. If the cursor is on a space between words, the space that the cursor is on and all of the spaces to the right will be deleted, up to the next word.

^Y Delete Line

Deletes the entire line that the cursor is on.

^QY Delete to End of Line

Deletes from the cursor to the end of the line that the cursor is on.

^QL Restore Line

Restores the line that the cursor is on to its prior state. A line that has had words or characters changed is restored to the way it was before any editing changes were made. (An entire line that has been deleted can not be restored because the cursor is no longer on that line.)

FIND AND REPLACE

These commands are used for searching and search and replace.

^QF - Find
^QA - Find and Replace
^L - Find / Find & Replace Again

^QF Find

Finds the specified text string and moves the cursor to that location. If requested string is not found, the cursor will remain at its original location.

^QA Find and Replace

Finds the specified text string and replaces it with the requested new text.

^L Find / Find & Replace Again

Find / replace text again. Repeats the previous find or find and replace command.

FILE AND BLOCK OPERATIONS

The following commands are used for dealing with files or blocks of text.

^KB - mark start of block
^KK - mark end of block
^KR - read file
^KW - write file
^KD - exit editor
^KC - copy block
^KV - move block
^KY - delete block

^KR Read a File

Read a file into memory. After you give the ^KR command, it prompts for the name of the file to read.

^KW

Write a file to disk. This saves the text are editing to a file name that you specify. After you give the ^KW command, it prompts for the file name.

** Block markers are always at column one. This means that all of the lines within a block are full lines, with no partial lines. No matter where you are on the line when you give a block mark command, the mark will be put on column one.

^KB Mark Block - Beginning

Mark beginning of block. This should be put on the first character you wish to have within the block.

^KK Mark Block - End

Mark end of block. This should be put on the line following the last line you wish to have within the block.

^KD Exit Editor

Exit editor to operating system. The command will prompt: ** ARE YOU SURE ** ?. This is to give you a chance to save the file before exiting the editor. A 'Y' or 'Yes' response will cause you to exit the editor.

^KC Block Copy

Make a copy of the marked block at the current cursor location. Block markers are all cleared after a copy of the block is made.

^KV Block Move

Move the marked block to the line just before the current cursor location. All block markers are cleared after the block is moved.

^KY Delete the marked block.

^KS Save the file back to the original file with the same name.

HAWTHORNE TECHNOLOGY

SCREEN EDITOR MODIFICATION MANUAL

Table of Contents

STRUCTURE

There are several major sections of code in the screen editor. The first level is a routine that gets a character from the console. It uses a case statement to see if this is a control character that needs to have a routine executed. If it is not a control character then it is checked to see if it is a printing character. If it is a printing character, it is inserted into the current line of text. Because of the large number of commands and the limited number of control characters a two level decodeing system is used. If a control Q or a control K is entered then a new menu is entered and the next character is decoded. The next level is composed of the routines that execute each command. The next level is composed of routines used by the first level. At the very bottom are the runtime library routines and a few very simple utilities.

The first thing that the editor does is to setup space for the local variables. All local variables are allocated from the heap of memory left over after the program is loaded. Then the initialize routine sets reasonable initial values for most of the variables to indicate an empty buffer but one that can have text added to it.

The routines that move the cursor on the screen are hardware dependent. The routines as supplied are setup for a Wyse-30 teminal. We have tried to minimize the number of screen commands used to make the editor more portable. To use the editor with a new terminal, these routines need to be changed. The routines that call these routines do not do any screen access. In some cases the control characters used many have to be changed to match a new terminal. The current control codes are patterned after Wordstar (tm) or Turbo Pascal (tm). In general the more frequently used routines are single control codes while the less frequently used routines or the more complex ones require a two code sequence.

The main loop gets a character and then uses a case statement to decode the command. If aliases are wanted for commands two different routine can call the same action routine. If no routine matches in the case statement then the character is inserted into the current line of text. After a routine is executed, control returns to the top of the loop and the process starts all over again. There is a reset command in the loop to reset the stack pointers in case there is any problem. This will prevent the editor from blowing up if an error is made.

The help screen is a simple routine that exits the current line, clears the screen, and then displays many lines of text on the screen. At the end of display it uses the continue routine to wait for a response from the operator before it continues. It then uses the new screen routine to create a new copy of the work area on the screen and reposition the cursor to where it was before the help was requested. If any routines are changed or added to the editor then the help screen should be changed also.

If more explanation is needed for some of the routines then a separate help screen could be added. In general though, I have found that a piece of paper with all the commands on it is a better guide than help menus and doesn't take up as much space in the computer.

The text is stored in lines in the edit buffer. A major advantage of this is that a line can be restored if it gets messed up. Also, since only one line is worked on at a time, not as many characters have to be moved when inserting or deleting. When the cursor is moved to a new line a routine is executed to enter the line. When the cursor is moved off a line a flag is checked to see if the line has been modified. If it has been modified then it is put back into the edit buffer.

The routine to enter a line is enterlin. This copies the line from the text buffer to the line edit buffer. When this is done the line is padded to a full 80 characters with extra blanks. This is so the cursor can be moved past the end of the

existing text as is possible with the Turbo Pascal editor. A table of the spaces used for tabs is calculated. The are two important pointers for position in a line. The first one, colpos, keeps track of the column on the screen. The other, bufpos, keeps track of the position in the buffer. When any command makes a change to a line, a flag is set. By having an old copy of the line that is being edited it is possible to restore the line if it is desired to undo the edit changes on that line.

The routine to leave a line is leavln. If no changes have been made the routine does nothing. If the line has been changed then the trailing blanks are deleted and the line is replaced back in the text buffer. If the size of the line has been changed then the hole it came from must be increased or decreased to fit. To accomplish this the entire buffer that follows the current line is moved. In theory this may not seem like a fast way to do it but in practice it works fast enough even with large files.

When a change is made in the line, or the cursor is moved, the line is redisplayed using a routine called dispinbuf. This makes it seem slow on slow displays but at 4800 baud or higher it looks ok. We did not use cursor positioning commands because of the spaces associated with tabs. With a low baud rate it is possible to overrun the cursor when using a repeat key on many terminals.

The major concern with a screen editor is to keep the screen that the user sees syncronized with the edit buffer that the program sees. If any change is made to any cursor movement command then this syncronization must be checked.

For most commands there are two parts. One part takes care of the effect that the command has on the screen. The other part takes care of the effect the command has on the text in the buffer. Both of these have to check for boundary conditions like beginning of line, end of line, top of screen, and bottom of screen.

The general purpose routine re.line replaces a line in the text buffer. This will either expand or contract the buffer as needed. If a line is deleted it is not replaced but the prior line is entered and the nul that deliniates a line in the buffer is changed to a cr. If a line is inserted, it is saved with an extra cr that is changed to a nul delimiter after the line is put back into the buffer.

The move block command in the screen editor is very different from the move command in the line editor. In the line editor a copy is made of the text being moved and then the old text is deleted. In the screen editor the buffer that has the old text and the position for the new text is treated as a single large buffer. The routine does a rotate in place and each character is only moved one time. This is why movement of text with most word processors doesn't result in a memory full message. The copy and delete block routines work basicly the same way in the screen editor as they do in the screen editor. The main difference is in how the first and last lines to be copied or deleted are determined.

The file read and write commands are similar. When reading from a file it is opened and lines are read one at a time and placed into the edit buffer. Since many editors use a cr and lf to mark the end of line, care must be taken not to create phantom lines in the buffer. When a file is written it is created. This will create a new file if one did not exist or empty a file if it did exist. Lines are blocked into bigger blocks to be written out but this is not strictly needed. It does spead up reading and writing if sector size blocks can be read or written.

Before making any changes to the editor you should make a copy of the source and a copy of a working binary version. This way if an error occurs you can back up to a know working copy. If possible you should only make one small change at a time and try to isolate it from the other commands. This will decrease the amount of debugging that must be done.

VARIABLES

This is a summary of all the variables used in the screen editor and what they are used for. In some cases a variable may be used for more than one purpose in which case only the most common use of the variable is given. Some of the variables contain permanent information and some are used in a transient manner.

These are character strings that are used in more than one place in the editor.

```
byte nofiln = "NO FILE NAME SPECIFIED" ;
byte cantop = "CANNT OPEN FILE" ;
byte nfndms = "STRING NOT FOUND" ;
byte hedtxt = "LINE:        COLUMN:
            COPYWRITE HAWTHORNE TECHNOLOGY 1987";
```

These are all local variables and need to have space allocated to them by the initialize routine.

```
long scrntop - top line displayed
long kblin - line where ^KB is
long kbcol - column where ^KB is
long kklin - line where ^KK is
long kkcol - column where ^KK is
long curlin - pointer to current line
long curnum - current line number
long lines - number of line in buffer
long fndnum - line number to look for
long fndlin -
long newnum -
long bufpos - position if buffer
long linpos - screen line
long colpos - screen column
long curpn -
long nsflg -
long l11 - search string size
long l12 - replace string size
long reptyp - type of repeat for ^L
byte inbuf [ 128 ] - input and edit buffer
byte newbuf [ 128 ] - overflow
byte optbuf [ 40 ] - option buffer
long texbuf - point to start of text buffer
long nextline - point to end of text buffer
long block - pointer to I/O block
byte rplstr [ 130 ] - replacement string
byte scrstr [ 130 ] - search string
byte workfil [ 40 ] - working file name
```


VARIABLES (continued)

```
long fndflg - found flag
long px - move pointer
long ofx - move offset
long count
long start
long nsave
long psave
long cursiz
long mvcnt
long size
long kk
long cyclpn
long startpn
long finishpn
long movsiz
long chngflg
long repcount
long tabpn
byte chx
byte filnam [ 40 ]
byte pbuf [ 130 ]
byte tabtab [ 40 ]
byte even
long thsfile
long infile
long outfile
long tbcnt
long lxpn
long chrcnt
long active
long dskbfpn
long bufend
long xitflg
word paramblk [ 10 ]
```

PRIMARY ROUTINES

These are the primary routines for doing the setup of the buffer and the main menu decoding.

setup - get buffer ready for editing
cmdlin - process the command line
program - main program
screened - main screen edit procedure

code	routine	action
[$12]	pageup	^R page up
[5]	lineup	^E line up
[$18]	linedn	^X line dn
[3]	pagedn	^C page dn
[4]	charrht	^D right ch
[$13]	charlft	^S left ch
[1]	wordlft	^A word left
[6]	wordrht	^F word rht
[8]	dellft	^H del left ch, BS
[$11]	quickmenu	^Q menu
[$0A]	helper	^J help screen
[$0B]	specialK	^K menu
[$0D]	newline	new line
[$0C]	repeatcmd	^L repeat
[7]	deletech	^G del chr
[$7F]	dellft	delete left
[$14]	delword	^T del word
[$19]	delline	^Y del line
[9]	charok	tab char

helper - special help screen ^J

quickmenu - menu for ^Q items

All of these codes must be prefixed with a control Q. If this is made into a remote system editor than the next most common control for this purpose is control O.

code	routine	action
['A']	findrpl	^QA search and replace
['B']	tostartb	^QB move to start of block
['C']	endofile	^QC move to end of file
['D']	toeol	^QD move to end of line
['F']	findx	^QF find string
['K']	toendk	^QK move to end of block
['L']	restorlin	^QL restore line
['R']	topofile	^QR move to top of file
['S']	tostrtl	^QS move to start of line
['Y']	delendln	^QY delete to end of line

specialK - menu for ^K items

code	routine	action
['B']	markstrt	^KB mark start of block
['C']	copyblock	^KC copy marked block
['D']	quiter	^KD stop edit
['K']	markend	^KR mark end of block
['R']	readfil	^KR read file into buffer
['V']	movblock	^KV move marked block
['W']	writefil	^KW write buffer to file
['Y']	delblock	^KY delete marked block
['S']	savefil	^KS save file

These are the primary routines called by the decoding cases when a character is input. These routines call the secondary routines.

```
quiter     - ^KD quit editing
pageup     - ^R page up
lineup     - ^E line up
linedn     - ^X line dn
pagedn     - ^C page dn
charrht    - ^D right ch
charlft    - ^S left ch
wordlft    - ^A word left
leftchk    - check if begin of line
leftwhite  - move left until white
leftnotwh  - move right until not white
iswhite    - true if tab or space
wordrht    - ^F word right
rhtchk     - check if end of line
rhtwhite   - move right until white
rhtnotwh   - move right until not white
newline    - new line
repeatcmd  - ^L repeat last ^QF or ^QA
deletech   - ^G delete this char
dellft     - delete to left
delword    - ^T del word
delline    - ^Y del line
charok     - simple character
```

```
------ quickmenu routines for ^Q items -------

findrpl    - ^QA search and replace
tostartb   - ^QB move to start of block
endofile   - ^QC move to end of file
toeol      - ^QD move to end of line
fndend     - used by toeol
fndcolpl   - find colpos given bufpos
findx      - ^QF find string
restorlin  - ^QL restore line
toendk     - ^QK move to end of block
topofile   - ^QR move to top of file
tostrtl    - ^QS move to start of line
delendln   - ^QY delete to end of line
```



```
------ specialK routines  for ^K items -------

markstrt   - ^KB mark start of block
copyblock  - ^KC copy marked block
markend    - ^KK mark end of block
readfil    - ^KR read file into buffer
rdopened   - read opened file into buffer
movblock   - ^KV move marked block
writefil   - ^KW write buffer to file
savefil    - ^KS save buffer to file in FILNAM
wrfilnam   - write buffer to file in FILNAM
delblock   - ^KY delete marked block
```

```
---------- generic routines ------------------

enterlin   - start changes on this line
leavln     - leave this line, clean it up
chop       - chop trailing blanks
dspinbuf   - display inbuf and fix cursor
newscreen  - do a new screen display
newhead    - redo heading on screen
newcoltxt  - print new col and line number
cursorpl   - move cursor to colpos, linpos
```

```
------ cursor movement --- put in RTL later ------
------ these are for Wyse-30 ------

clrscr     - clear screen
eraseln    - erase to end of line
delcrtlin  - delete line
inscrtlin  - insert line
colcrtlin  - goto column and line
scrolup    - scroll up, new line at bottom
scroldn    - scroll down, new line at top
cursof     - turn off cursor
curson     - turn cursor on block, blink
```

```
-----------------------------------------------

finder     - find string
changer    - change string found
```

```
************ SUBROUTINES **********************

re.line      - replace line in buffer
up.line      - go up one line in buffer
do.line      - go down one line in buffer
tomarklin    - validate line with marker
gotolin      - goto a line - linnum
fnd.line     - find line curnum
lookup       - find line,assume curnum good
sizline      - determine size of line in inbuf
inclines     - increment lines and fix pointers
declines     - decrement lines and fix pointers
resetmarks   - turn off both block markers
blocknok     - returns if block not ok or inside
insideblock  - returns if cursor in block

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

getstr     - get search string
getrpl     - get replacement string
getoption  - get options
getnam     - get file name
gcmdstr    - get command line from K-OS
gstring    - display prompt and get string
openit     - open file
opfilnam   - open file in filnam
rewritef   - rewrite file in filnam
closeit    - close file - unit on stack
prnterr    - print error message

****************** IN AND OUT ROUTINES **********

readln    - read next line from disk
fgetc     - get character from disk file
savlin    - send current line to disk buffer
fputc     - send one char to disk
putblk    - write current block or part to disk
pline     - print current line w/o crlf
putcx     - put char in output line
writeln   - write string to console w/crlf
crlf      - send cr and lf to console
continue  - " PRESS * TO CONTINUE >>"
upcase    - convert lower case to upper case
```

HAWTHORNE TECHNOLOGY

TEXT FORMATTER USER MANUAL

Table of Contents

STRUCTURE

This text formatter is patterned after the formatters that were used on the early UNIX systems, (roff and nrof). Before word processing became popular, text was entered and modified using an editor. It was then formatted using a formatter as it was printed out or transfered to another file. Today many word processing programs combine these functions to produce what is known as a WYSIWYG editor. The name comes from the initials of: What You See Is What You Get. A WYSIWYG editor is more complex than one devoted strictly to editing, such as the kind you use in program development.

A major reason for using a text formatter in addition to an editor is that the text formatter can be specialized for different print jobs. In the field of desktop publishing there is a program called TEX and a language called Postscript that are used to prepare documents for printing. A text formatter can be set up to handle many of the tasks that are found in desk top publishing.

The main parts of a text formatter are the command processor and the text handler. As each line is read it is checked to see if the first character is a period. This is something that seldom happens in regular text. When a period is found in the first column, the next two letters are decoded to determine what command has been requested. No text is recognized on command lines. If the line is not a command the format program formats it according to the current settings and prints it out.

TEXT FORMATTER COMMANDS

Format commands are given using a period, a two-letter name, and in some cases, optional information. The command must be the only thing on the line and begin in column 1.

If specific formatting commands are not given, the formatter will use default values when formatting the text.

As the formatter takes in the input lines, it builds output lines. The output lines will be nearly equal in length no matter how un-equal the input lines.

The formatter has two modes: No-Fill, or Fill. The No-Fill mode, one input line equals one output line. There is no rigth margin justification. Margins, page spacing, line spacing etc. all function the same. In No-Fill mode, each line is treated seperately. In the Fill mode the output lines are made as near to equal in length as word sizes allows. Spaces are then added to justify the right margin.

These commands can be assembled to create packages similar to Mail Merge or even build up to a simple version of Postscript.

Text formatting commands are inserted in your text file as it is created. The text file is then processed with the formatter program.

To involke the text formatter program you give it the command ROFF and file name(s) on the same line. If no destination file name is given, the results of formatting will be desplayed on the console.

EXAMPLE:

```
ROFF sourcefile.ext destination.ext
```

FORMATTER COMMAND TABLE:

CMD		BREAK?	DEFAULT	COMMAND
.BP	n	Y	n=+1	begin page number n
.BR		Y		cause break
.CE	n	Y	n=1	center next n lines
.FI		Y		start filling
.FO		N	empty	footer title
.HE		N	empty	header title
.IN	n	N	n=0	indent n spaces
.LS	n	N	n=1	line spacing
.NF		Y		stop filling lines
.PL	n	N	n=66	set page length to n
.RM	n	N	n=60	set right margin to n
.SP	n	Y	n=1	space n lines
.TI	n	Y	n=0	temporary indent of n
.UL	n	N	n=1	underline words from next n lines

Unknown commands are always ignored by the text formatter.

.BP n Begin Page

This command is used to start a new page. 'n' is the page number that will be used on the next output page. If .BP is the last thing on the page, it will not cause a blank page. If no 'n' value is specified, the default value will be n+1 or the next number after the prior page number. The .BP causes a break.

.BR Break

This command is used to force a partial line. Text following the .BR will be put on the next line. This would be used in places like starting a new paragraph.

.CE n Center Text

This command is used to center text. The default value if 'n' is not specified is one line. The line(s) to be centered follows the command.

```
.CE 2
This line will be CENTERED.
This will also be CENTERED.
This line WILL NOT BE CENTERED.
```

The center command causes a break.

.FI Fill

This command turns on the fill feature. When it is on, output text is justified (right margins made even). The default of this formatter is with Fill on, so this command only needs to be used when you have requested no-fill mode and now wish to return to fill mode. This command causes a break.

.FO Footer Title

.HE Header Title

These commands cause a title to be printed on each page. The Footer Title goes on the bottom of the page. The Header Title goes on the top of the page. These commands default as blank, so nothing is printed unless titles are specified. The title is specified following the command on the same line.

.HE Page Title

If you wish to incorporate the current line number in your title, this can be done by using the '#' character in the title where you wish the page number to be placed.

.HE Title on Page Number #

In this case the title will reflect the current line number on each page without the operator having to reset the header title on each page.

.IN n Indent n Spaces

This causes all of the lines put out after this command to be indented by 'n' spaces. The default value of n is zero. To stop the lines from being indented, you set the indent value back to zero.

.LS n Line Spacing

The line spacing defaults to one for single spacing. For double spaced output, you set the value of 'n' to 2.

.LS 2

All of the lines output after this command will be double spaced until a new value is give.

.NF No Fill

This causes the fill mode to be turned off. Each line of input becomes one line of output and will not add spaces to justify the right margin. This command causes a break.

.PL n Page Length

This command is used to set the number of lines per page on your printer. The default value is 66. This is the proper value for eleven inch paper on a standard six line per inch printer.

.RM n Right Margin

The right margin defaults to column 60. The .RM command is used when some other value is desired.

.SP n Space n Lines

This causes 'n' blank lines to be produced by the formatter. The default value of 'n' is one. This command causes a break. The .SP does not cross a page boundry. If the end of a page is incountered, the formatter will go on to the next line of the input.

.SP

This command would cause a break in the text and one blank line to be inserted.

.TI n Temporary Indent of n

This command causes a break and indents the line following the command by 'n' characters. It is refered to as a temporary indent because it only affects one line.

.UL n Underline Words On Next n Lines

This command is used to underline complete lines of text. It is useful for underlining headings or large blocks of text.

```
.CE 3
.UL 2
SUPER PRODUCTION
WEEKLY PROGRESS REPORT
By Robert Brown
```

These three lines will be centered on the page and the first two of them will be underlined.

```
.RM 50
.CE 2
TEXT FORMATTER
SAMPLE TEXT

This is sample text to be used to see how
the text formatter works.
The text formatter should even out
the length of these lines so they will all be
approximately the
same length. If the fill mode is on, the right
margin will end up even. If the fill
mode is off, the
lines will be nearly the same length.
.BR
.SP
The right margin for this document will be column
fifty.
The two lines of
heading on this text will be centered.
A second paragraph will be started after the .BR
and .SP commands which cause a line break and
then a blank line.
.HE TEXT FORMATTER (Continued)
All of the
pages after this will have the header:
TEXT FORMATTER (Continued)
at the top of them.
.IN 5
The lines that start after this command will
be indented by 5 characters each.
This should continue until
the indent
is set to some other value. The default value
on indent is zero so you don't have
any indent unless you ask for one.
.IN 0
The next lines should again start on the left
margin with no indentation.
this will give a reference point for
the next feature to be tested which is the
temporary indent.
.BR
.SP
.TI 3
This will do an indent of 3 characters at the
beginning of this new paragraph.
```

```
The rest of the paragraph will be flush
on the left margin. This is handy and you don't
have to remember to
put it back when you don't want
it to indent any more.
.LS 2
Now we should get some double spaced text as a
result of setting the line spacing
to two.
When we get done seeing how double spacing works
we can try some text in the fill-off mode.
Well, I think we can all see double
(spaced that is) so first I'll set the
spacing back to single space
.LS 1
and now I'll turn off the fill mode.
.NF
These lines should now come out with
nearly the same length, but without any blank
spaces added to justify the right
margin.
You can see, that
this makes the output more readable
than the input without any extra
spaces.
.BR
.SP
By now we should have reached the second page, so
lets try changing the page length.
.PL
By setting the page length to 45, I can get
an output that will fit sideways in the
copier without any reductions.
You may never need to do this, but I never know
when I will want to do something different.
```

TEXT FORMATTER
SAMPLE TEXT

This is sample text to be used to see how the text formatter works. The text formatter should even out the length of these lines so they will all be approximately the same length. If the fill mode is on, the right margin will end up even. If the fill mode is off, the lines will be nearly the same length.

The right margin for this document will be column fifty. The two lines of heading on this text will be centered. A second paragraph will be started after the .BR and .SP commands which cause a line break and then a blank line. All of the pages after this will have the header: TEXT FORMATTER (Continued) at the top of them. The lines that start after this command will be indented by 5 characters each. This should continue until the indent is set to some other value. The default value on indent is zero so you don't have any indent unless you ask for one. The next lines should again start on the left margin with no indentation. this will give a reference point for the next feature to be tested which is the temporary indent.

This will do an indent of 3 characters at the beginning of this new paragraph. The rest of the paragraph will be flush on the left margin. This is handy and you don't have to remember to put it back when you don't want it to indent any more.

Now we should get some double spaced text as a result of setting the line spacing to two. When we get done seeing how double spacing works we can try some text in the fill-off mode. Well, I think we can all see double (spaced that is) so first I'll set the spacing back to single space and now

I'll turn off the fill mode.
These lines should now come out with
nearly the same length, but without any blank
spaces added to justify the right
margin.
You can see, that
this makes the output more readable
than the input without any extra
spaces.

By now we should have reached the second page, so lets try changing the page length.
By setting the page length to 45, I can get
an output that will fit sideways in the
copier without any reductions.
You may never need to do this, but I never know
when I will want to do something different.

HAWTHORNE TECHNOLOGY

TEXT FORMATTER MODIFICATION MANUAL

Table of Contents

STRUCTURE

The text formatter is in the general style associated with roff and nroff that were first written for UNIX in the early 1970's. A dominant feature is the use of a period in the first column of a line to mark a command. The specific version that this program most closely follows is the version given in the book: Software Tools by Kernighan & Plauger.

The basic job of the text formatter is to combine short lines to make longer lines or break long lines to make shorter lines. Page breaks must also be handled.

The main loop reads lines from the input file. As each line is read it checks the first character to see if it is a period. If it is, the line is treated as a command or as a comment if the command can't be decoded. If the line is not a command line it is scanned and the words are sent one at a time to the output buffer. When the output buffer is full it is expanded by adding spaces and sent to the output.

The input to the text formatter always comes from a disk file. The output can be directed to the printer or to a disk file. It could also be directed to a modem or other device. If no output is specified then then formatted text is sent to the console.

Some of the commands like BR (break) cause an immediate action to occur. Others set a string variable like TI (title). Some are switches like fill and are either on or off. A few like page size have a numeric value. The ones with a numeric value are set by a routine that looks at the min value, max value, default value, and new value. This insures that the variables are set to some reasonable value even if the command is not good.

The file I/O is done by a group of generic routines. These are similar to the routines in the command processor that is part of the operating system and are also similar to routine with the same functions in the editors.

CHANGE SUGGESTIONS

Because of the modular nature of the text formatter is is easy to add new commands to do special functions. These additions could be published or put in user libraries. While the formatter as a whole is copyright protected, any changes and instructions on how to add them would be new material.

As with any flexible program it is sometimes hard to decide what to put in and what to leave out. If you want a small, fast, convenient program that doesn't take up much space so you can have it around then you have to leave out many features that may not be used very often. If you do heavy work with a text formatter like publishing a newsletter or magazine then you will want to leave all of the commands in and probably add some.

A command to turn the justify on and off would be useful. These could be JU - to start justify and NJ - to stop justify. This would make it possible to have parts of the source file copied with the lines being equized in length but with no fill characters added to justify the right margin.

In its present form there is no way to set the size of the heading and footing margins on the page. These could be done with HS - headspace and FS - foot space. This would make it a little easier for certain print jobs.

For advertizing copy it is sometimes desired to have text right justified with a ragged left margin. Another use for right justify is for headings in manuals. A new command RJ - for right justify could be used for this. To impliment it all that would be needed is to have a routine that is similar to fluf that simply moves the text to the right end of the print buffer.

To make this formatter into a full mail/merge type of program all that is needed is the ability to read from an alternate file and have some method of looping to repeat the body of the text. For

form letters it would be nice to have a canned heading in a file that could be called up. A document could even be assembled by inserting stock paragraphs from a text data base.

Most people who have a dot matrix printer know that it is possible to select different type styles and sizes under program control. A group of simple commands can be added to select these different type styles. Because this would be different for each brand / model of printer, a short library of control codes could be published. For the printers that have the ability to load a whole new type font, a command could read the new font from a file and send it to the printer as part of the text formatting process. Could you imagine a program listing, possibly in COBOL printed out in Old English script.

The K-OS ONE operating system maintains a date and time of day clock. A useful command would be to read the system date and automatically put todays date and possibly time on the heading of a letter or report.

Sometimes you don't want to split a paragraph between two pages or you don't want the title of a section on one page and the start of the text on the next page. In that case it would be useful to have a command NE 22. This is a conditional new page command. If you need (NE) a certain number (22) of lines for the block of text you want to keep together, and they aren't available on the remainder of this page, then a new page is started. If the lines needed are available on this page then the command has no effect.

VARIABLES

These are all of the variables used in the text formatter. Most of them are used for a single purpose but some of them serve several transient uses. Most of the variables are declared as long even if they are a small value. This is done because the amount of storage taken by the variable is more than offset by the shorter code that is generated for long operations rather than for word or byte operations. Most of these are local variables and need to have space allocated to them and need to be initailized.

```
byte inbuf [ 384 ] - command line holder
byte tbuf [ 384 ] - underline buffer
byte outbuf [ 128 ] - output buffer
byte pline [ 128 ] - print line
long fillon - fill on flag
long lsval - line space
long inval - indent count
long rmval - right margin
long tival - temporary indent amount
long ceval - center line count
long ulval - underline char count
long curpag - current page number
long newpag - new page flag
long lineno - current line number
long plval - page length
long m1val - before heading line count
long m2val - after heading line count
long m3val - before foot line count
long m4val - after foot line count
long bottom - bottom of page flag
byte header [ 128 ] - heading text
byte footer [ 128 ] - foot text
byte filename [ 64 ] - current file name
byte lasteol [ 2 ] - last eol char
long outp - ouput pointer
long outw - ouput words
long outwds - words in outbuf
long spval - blank line count
word  paramblk [ 20 ] - parameter block used for
                        system calls
long val - argement value
long argtyp - argument type
long wrdwid - word size
long wrdlen - word length
```

```
long outpn - outbuf pointer
long wrdpn - pointer to word
long prnpn - printer pointer
long block - disk input block pointer
long dskbfpn - disk buffer input pointer
long eof - end of file flag
long infile - input channel
long outfile - output channel
long chrcnt - char count in disk block
long scpn - argument scan pointer
long inpn - input pointer
long allholes - number of spaces in line
long wideholes - number of wide holes
long thinholes - number of thin holes
long nholes - number of holes
long nextra - extra spaces needed
long direction - direction to fill
long srcpn - source pointer
long dstpn - destination pointer
long ipn - underline pointer
long tpn - underline pointer
```



```
(-- action routines --)

proc - command ( format command )

code  action
--------------------------------
BP ... begin page
BR ... break
CE ... center lines
FI ... start filling text
FO ... footer
HE ... heading
IN ... indent
LS ... line spaceing
NF ... no fill, stop filling
PL ... page length
RM ... right margin
SP ... insert blank lines
TI ... temporary indent
UL ... under line

text      - process text lines
getval    - get a function value
getfil    - get a file name
getstr    - get a text string
ctoi      - convert characters to integer
iswhite   - check if char is blank
iseol     - check if char is end of line
skipbl    -
skipwrd   -
set       - fix new value
getwrd    - get next word in text line
get       - get a whole line
center    - center line in outbuf
underl    - underline words in inbuf
nextch    - get next char
putwrd    - add word to line if it fits
put       - put out line
            with proper spacing and indenting
space     - put linefeeds in page
phead     - print top margins and header
pfoot     - print bottom margins and footer
puttl     - put title in print buffer
            w/optional page number
putdec    - put page num in buffer
skip      - add linefeeds to buffer
leadbl    - delete leading blanks, set tival
```

```
break      - send line
fluf       - add spaces to fill line
copybl     - copy backwards for fluf,
             true if blank copied
putbl      - put blank in backwards for fluf

(------ generic i/o routines --------)

openfil    - open file for in and out
createfil  - create file for output
readfil    - read file
writefil   - write to channel
closefil   - close channel
gcmdstr    - get command line
filerror   - "** ERROR CANT OPEN FILE **"
getlin     - get a line from source file
getcx      - raw char input
swrite     - write string to output file
writeln    - write string and crlf to con:
crlf       - send cr and lf to console
continue   - " PRESS * TO CONTINUE >>"
upcase     - convert lower to upper case
hexlong    - write long in hex to con:
hexword    - write word in hex to con:
hexbyte    - write byte in hex to con:
hexnyb     - write 4 bits in hex to con:
```