DESIGN AND IMPLEMENTATION OF AN INTEGRATED SCREEN-ORIENTED TEXT-ETC.

JUN 80  L A TALMAGE
DESIGN AND IMPLEMENTATION OF AN INTEGRATED SCREEN-ORIENTED TEXT EDITING AND FORMATTING SYSTEM

by

Lisa Anne Talmage

June 1980

Thesis Advisor: F. Burkhead

Approved for public release; distribution unlimited
Design and Implementation of an Integrated Screen-Oriented Text Editing and Formatting System.

Lisa Anne Talmage

Naval Postgraduate School
Monterey, California 93940

Approved for public release; distribution unlimited

Text editors and text formatters/processors are described. The state-of-the-art in text processing is examined. Design and implementation considerations in developing an interactive, integrated, screen-oriented text editing and formatting system are discussed. A Berkeley PASCAL implementation of such a system for a PDP-11 minicomputer under the UNIX timesharing system is presented. Intended system users are non-computer.
block number 20.

Scientists, e.g. secretaries, business executives, engineers, students, etc. Programming experience is not required to utilize the system.
DESIGN AND IMPLEMENTATION
OF AN INTEGRATED SCREEN-ORIENTED
TEXT EDITING AND FORMATTING SYSTEM

by

Lisa Anne Talmage
Captain, United States Marine Corps
B.S., William Woods College, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1980

Author

Lisa A. Talmage

Approved by:

Thesis Advisor

Second Reader

Chairman, Department of Computer Science

Dean of Information and Policy Sciences
ABSTRACT

Text editors and text formatters/processors are described. The state-of-the-art in text processing is examined. Design and implementation considerations in developing an interactive, integrated, screen-oriented text editing and formatting system are discussed. A Berkeley PASCAL implementation of such a system for a PDP-11 minicomputer under the UNIX timesharing system is presented. Intended system users are non-computer scientists, e.g. secretaries, business executives, engineers, students, etc. Programming experience is not required to utilize the system.
# Table of Contents

## I. Introduction
- A. Thesis Objective
- B. Text Editor Definition
  1. Line-Oriented Editors
  2. Context-Oriented Editors
- C. Text Formatter Definition

## II. Text Processing Survey
- A. Commercial Text Processing
- B. Commercial Trends
- C. Existing NPS Text Editors and Formatters
- D. Conclusions

## III. Design Methodology
- A. Product Definition
- B. Design Goals
- C. The Design Process
  1. Terminal Characteristics
  2. Scope Features
    - a. File Handling
    - b. Screen-Orientation
    - c. System-User Communications
    - d. Error Handling and Recovery
    - e. Editing
    - f. Formatting
I. INTRODUCTION

A. THESIS OBJECTIVE

Most existing text editors and formatters currently available are deficient for various reasons. Specifically, most text editors are line-oriented, many formatters are non-interactive, and few editors and formatters are integrated into a single system. For these reasons, the design of a "better" text processing system was undertaken to demonstrate the feasibility and utility of such a comprehensive system. Toward this end, a survey of the current state-of-the-art in text processing was carried out. Based on this research, the design of a "better" text processing system is presented.

This system is an interactive, integrated, screen-oriented, context-oriented text processing system consisting of both a text editor and formatter. It is designed to be easy-to-learn, easy-to-use, easy-to-enhance, and easy-to-maintain. Intended system users are non-computer scientists, e.g. secretaries, business executives, engineers, students, etc., who may or may not have programming experience. This system has been designed to aid in the preparation of documents and/or programs. The system has been implemented on existing Naval Postgraduate School (NPS) hardware using available software.
Four features characterize this system. First, it is fully integrated in that the text editor and formatter are not separate entities but complementary parts of a single program. Secondly, this system is interactive and provides the user with immediate feedback. Third, it is screen or display-oriented. A visual display, on a cathode ray tube (CRT)-equipped terminal, serves to provide both the editing and formatting feedback. The screen-orientation eliminates the need for many hardcopy printouts as the screen depiction is exactly how the final, finished product will appear. Finally, manipulation of the text through the screen window for perusal and modification is context-oriented.

B. TEXT EDITOR DEFINITION

Although the term "text editor" is widely used, it is necessary to clearly define its meaning here. In general, an editor is an interactive computer program that allows the user to create and modify a text file. This file is simply a sequence of character data. An editor is used to write documents such as term papers, theses, letters, etc., and programs. The text editor performs the following tasks:

- provides access to the text stream from both file storage and terminal input,
- adds text to the text file,
- distinguishes between words and spaces to split
and build lines of text, and
- alters the text at a given location based on
  user specifications.

Text editors are widely used because corrections and
modifications to the text file are easily made as opposed
to re-entering a complete updated version of the text.

Most editors provide functions to insert, delete, copy,
move, scan and substitute text within the text file. A
text editor also provides facilities to manipulate the text
file so that it may be created, destroyed, saved or
updated. Displaying and/or printing out the text is easily
accomplished. In addition, the user is able to locate a
particular location, anywhere in the file, in several
different ways. The user can scan through the displayed
text by moving the cursor, by requesting the display of
specified lines or by initiating a context search in which
the editor attempts to match a specified sequence or
pattern of characters.

There are two distinct types of text editors that
impact upon this project. The first category is line-
oriented and the second is context-oriented. These
categories are described in the following paragraphs.

1. **Line-Oriented Editors**

   Line-oriented editors are those in which the text is
   logically divided into lines. A line is a character
sequence that is delimited within the file by a special line marker at its beginning or end. Lines should not be confused with sentences. This type of editor is distinguished by the assignment of either a permanent or a relative line number to each and every line of text. These line numbers serve as user reference points. Editor commands include line numbers to indicate where the desired action is to occur. The major advantage of line-oriented editors is their ability to be used on both CRT and hardcopy terminals and their tolerance of a slow communication rate. In addition, many secretaries and typists feel that a physical line is the most logical and reasonable entity to manipulate while editing. Two other factors that sell line-oriented editors are the disruption of touch-typing habits on key-defined function editors and the completely context-free operation of character-oriented editors.

2. Context-Oriented Editors

Context-oriented editors are so termed because there are no associated line numbers for reference purposes. Instead the text file is referenced relative to the existing text by specifying a particular marker or sequence of characters in a context search, or by the use of absolute x-y addressing if available on the CRT terminal. These files are usually divided into pages which may or may not be further broken down into lines. Context editors are
not generally implemented on hardcopy terminals or CRT terminals with a low baud rate because too much time elapses while the page of text is printed out or displayed.

C. TEXT FORMATTER DEFINITION

A text formatter or processor is a computer program which allows the user to manipulate the text file to produce a specified output format. Formatters are used in document preparation. The processor keeps track of the page format as defined by the page margins, number of lines per page, and line spacing specifications which are all subject to change. It provides page numbering, indentation, headings and footers, and an underlining capability. A typical text formatter, [5, 7, 10], performs the following tasks:

- accesses the text file from storage,
- distinguishes between format commands and text,
- provides reasonable and general default format specifications,
- redefines the format specifications as directed by the user,
- splits and concatenates lines of the original text file to produce lines conforming to the output specifications,
- fills text by putting as many words as possible within a line.
- outputs blank lines or skips lines,
- provides for page ejects or skips to the top of a new page,
- outputs irregular, non-filled lines of text, e.g. titles, when directed by the user, and
- allows the line to be centered, right-justified (aligned along the right margin), or underlined.

Text formatters allow drastic format changes to be accomplished by the user to improve the document's final appearance and thus its readability.

Many text processors adjust the lines of text which means the line is both filled and right-justified by inserting extra blanks within the line. Sophisticated formatters, in addition to the previously mentioned features, may provide spelling checks, hyphenation, footnote production and generation of a table of contents or indices. Text processors may or may not be interactive. Non-interactive formatters require the user to visualize the effect of the format command and later verify the desired result by printing out a hardcopy listing of the document. Interactive formatters do not require this mental transformation and verification process by the user because the command result is immediately apparent.

Before proceeding on to the survey of text processing, it is appropriate to define a text processing capability.
Systems possessing a text processing capability are able to perform the previously mentioned tasks of both an editor and formatter. For the purposes of this thesis, word processing and text processing are synonymous.
II. TEXT PROCESSING SURVEY

The initial step in this thesis was to survey the current state-of-the-art in text processing. In this survey, the commercial marketplace was reviewed. In addition, ten existing NPS text editors and processors were observed. The survey goal was to determine the most desirable features of a text processing system.

A. COMMERCIAL TEXT PROCESSING

Although text processing has been around since the mid-sixties at academic and research institutions, it was not until the mid-seventies that it became commercially attractive. The technological advances of the early and mid-seventies brought about a rapid decrease in both computer size and price. These factors made them very attractive to perform a variety of tasks, particularly text processing, in an automated office. FORTUNE's "Trends in Computing--Applications for the 80's" (201), describes automation's impact in this manner, "In the white collar factory, the office, the computer and its microprocessor-based offspring are inciting a revolution of epic proportions with word and text processing...." Computer vendors of both hardware and software are scrambling to establish themselves in the emerging office automation market. The following examples of this scramble were cited
in the FORTUNE article [20]. Data General has added text processing to the commercial capabilities of its Eclipse Computers. Nano has given all of its VS computers a text processing capability through "mailword". Germany's Nixdorf disclosed a second generation of distributed data systems that perform text processing, batch and interactive communications, data entry, and local file and data base processing. Dunn, in COMPUTER DECISIONS [4], provides this estimate of the size of the text processing market, "Revenues from word processing have jumped from $936 million in 1977 to $1.5 billion in 1978 when 120,000 units were acquired, and are projected to reach 600,000 units, grossing $6 billion, in 1983."

Two factors causing management's acquisition of text processing facilities are the secretarial shortage and the low productivity of white collar and clerical employees. Dunn, in "The Office of the Future Part II" [5], details the secretarial problem in this way, "...last year [1978] 60,000 secretarial jobs went unfilled, though salaries are rising more than 10% a year. By the mid-1980's the secretarial shortfall could reach the quarter of a million mark." Wilds, in "The Smart Way to Pick Word Processors" [24], provides an example of the productivity impact of text processing. A typical productivity increase occurred at J. C. Penny Co. where the installation of text processing equipment cut turnaround time on typed documents.
from "as long as a week" to an average of "next day".

There are two principal products sold by text-processing vendors. Stand-alone text processors with price tags of $6,000 to $12,000 each are the most popular. An alternative, the purchase of a word-processing package for CPU's, makes text processing another data processing application. These packages cover a considerable price range up to $75,000. Independent suppliers, rather than computer manufacturers, provide most of this software. A representative sampling of text processing software packages, derived from the January 1980 issue of COMPUTER DECISIONS [19], packages would include the following:

- **WORD-ONE**—used with IBM 360/40 or above, 370/135 or above, 4300 series, National and Amdahl systems; sold by Bowne Information Systems for $65,000;

- **AZ-TEXT**—used with Data General's Advanced Operating System and Eclipse System; sold by Data General Corporation for $4,000;

- **Datapoint Word Processing**—used with Datapoint 3800, 1800 and 1500 series processors; sold by Datapoint Corporation for $750;

- **Word-11**—used with DEC RSTS/E timesharing executive; sold by Data Processing Design, Inc. for $7,500.
There are many text processing systems available for microcomputers. The following packages are a few of those available for INTEL-based 8080 and Z-80 microcomputers. Small Business Applications, Inc. markets Magic Wand at $400. The Software Store has MWP (Mini Word Processing System) for $195. The TRS-80 Microprocessor can use either The Electric Pencil, for $100 or $150, or Scripset, for $69 or $99. These software packages can be economically integrated into existing systems without the purchase of special hardware.

As previously mentioned, stand-alone or dedicated systems are the most popular. The vendors of these systems market a complete hardware/software package consisting of a workstation and the support software. These systems usually have specialized terminals. "Clustered, or shared resource, word processing systems (which rely on minis and micros) are the fastest growing word processing segment" according to FORTUNE (20). CPT, Lanier, AM Jacquard, Wang and Xerox are only a few of the vendors marketing these systems.

B. COMMERCIAL TRENDS

Several interesting developments are occurring in text processing. First, CRT-displays are inheriting the market from nondisplay systems. Text processors that use hardcopy terminals exclusively are being recognized as less productive and less attractive. Also, significant price
decreases make display systems more attractive. Second, there is a trend toward resource sharing and joint text processing and data processing. In this area, AXXA Corporation tackles executive and secretarial productivity with its new dual workstation, supporting text processing, electronic mail, calculation, data processing and file access. Xerox Office Products announced the 860 Advanced workstation that performs text processing, system bookkeeping and data processing. Finally, the new systems are multifunctional. Text processing is integrated with telecommunications, facsimile transmission, numeric processing, electronic mailing, data entry and access, and other functions such as search/sort, filing, statistics generation, etc. [20, 25]

C. EXISTING NPS TEXT EDITORS AND FORMATTERS

To evaluate the current state-of-the-art on campus, a review of ten editors, formatters or word processing systems was undertaken. These systems were either used by the author or observed in use. User's Manuals and other system documentation were reviewed. The following systems were examined:

(1) CRT Text Editor NED, 1977, used on the PDP-11/70, [21];

(2) AMOS/2 Editor, 1972, used on the AGT-10, [61];
(3) EDT Editor, 1977, used on the PDP-11/50, [7];

(4) Interactive Display Editor Vi, 1979, used on the PDP-11/50, [13];

(5) NROFF text processor, 1974, used on the PDP-11/50, [16];

(6) TED editor, 1978, used on the 8080 and Z-80, [21];

(7) PRO formatter, 1978, used on the 8080 and Z-80, [21];

(8) UCSD PASCAL Screen-Oriented Editor, 1978, used on the 8080 and Z-80, [22];

(9) UNIX Text Editor Ed, 1975, used on the PDP-11/50, [23]; and


These systems covered the entire spectrum of text editors and formatters from the simple, relatively unsophisticated, general-purpose models to complex, sophisticated, specialized systems. They also provided a fairly representative historical development of text processing systems, from the non-interactive, line-oriented NROFF formatter (1974) to the interactive, display-oriented, context editor Vi (1980), and the integrated, interactive, screen-oriented context WPS-8 Word Processing System (1978).
Of the ten systems examined, the WPS-8 was the only
stand-alone text processing system. The other systems were
run on general-purpose computer systems. Terminal
characteristics varied considerably among the reviewed
systems. ED, NED, NROFF and WPS-8 utilize special graphics
terminals. NED and WPS-8 require special function keys.
UNIX's Ed and Vi, PRO, TED, and UCSD's editor, can be used
on a wide range of terminals, both intelligent and dumb,
display or hardcopy. AMOS/2 uses a teletype.

Of the editors surveyed, AMOS/2, EDT, UNIX's Ed, and
TED were line-oriented. The context-oriented editors were
NED, Vi, UCSD's editor and the WPS-8. NED and UCSD's
editor performed some formatting functions such as margin
setting, filling, centering, right justification, etc. Vi
and WPS-8 had the most extensive command sets. The command
vocabulary was, unfortunately, so extensive that the user
had to frequently refer to system documentation for help.
Some of the commands were very similar looking but
performed widely different functions. The special function
keys on the NED and WPS-8 terminals facilitated the editing
process. Editor commands were generally one character
followed by line numbers where appropriate. Many commands
were mnemonically indicative of the action performed.
Cursor positioning was used on the context-orientededitors. NED allowed the editing of more than one file at
the same time, as did UNIX's Ed. An additional feature
provided by NED was the capability to divide the screen up into as many as ten completely separate and independent edit areas.

Of the formatters surveyed, PRO and NROFF were non-interactive while WPS-8 was interactive. PRO is used as a post-processor for files edited with TED, and provides many formatting features including footnoting. NROFF and WPS-8 provide almost any formatting feature ever thought of. Commands were generally two letters in all three systems and frequently had numeric arguments. NROFF and TED are sometimes time-consuming to use because of their non-interactive nature and the user's failure to correctly visualize command results.

D. CONCLUSIONS

This chapter has provided a brief survey of the state-of-the-art in text processing. The evolution of text processing over the last ten or so years has been traced. Non-interactive, hardcopy systems have been replaced with interactive, softcopy or CRT-based systems. Line-orientation has been replaced by context-orientation. In the past, editors and formatters were separate systems but now they are fully integrated into a comprehensive text processing system. Special hardware, terminals and micro- and mini-based stand-alone systems, have been developed and are quite common. Not only do the text processing systems of today perform editing and formatting duties but they are
also capable of performing a wide range of other functions, such as system bookkeeping and data processing. Multi-functional text processing has become an integral part of today's automated office.
III. DESIGN METHODOLOGY

A. PRODUCT DEFINITION

Before any design could be contemplated, the final product had to be envisioned and defined from a user's point of view. The product of this effort is an integrated, interactive, screen-oriented text processing system. SCOPE, Screen-Oriented Processor and Editor, performs the previously defined functions of both a text editor and formatter. It is implemented on the existing PDP-11 mainframe, using existing terminals and software. SCOPE is general-purpose and is not intended to perform graphics or uncommon formatting functions, e.g. the generation of two columns of text necessary for some publications. Thesis preparation was a consideration in the selection of desired formatting features.

From the ten NPS systems reviewed, the following were chosen as models for this thesis:

(1) the Vi Interactive Display Editor,
(2) the UCSD PASCAL Screen-Oriented Editor,
(3) the text editor TED, and
(4) the text processor PRO.

In addition, a UCSD PASCAL Formatter, written by the author, served as a model.
The goals, the design and the development process of SCOPE will be discussed in the remainder of this chapter.

B. DESIGN GOALS

Further research was undertaken to determine the elements of a good design. The following paragraphs detail design goals, set for other design and implementation projects, that were applicable to this system. These references, "Overview of the Functional Features and Software Design for a Display Word Processor" [14]; Kernighan and Plaugher's chapters on editing and formatting [15]; "Text Handling in an Automated Office" [17]; and "Development of an Experimental Display Word Processor for Office Applications" [18]; served as the basic source documents.

In SOFTWARE TOOLS [15], Kernighan and Plaugher discuss editing and formatting design factors. The following points were made in their discussion. Human engineering (the analysis of man's habits and work patterns in the attempt to optimize the man-machine interface) considerations are identified as the primary design factor. For this reason, an interactive system has streamlined and terse communications with the user and is easy-to-use. Experienced users were found to prefer concise commands so brevity and mnemonic value are counterbalanced. They recommend that most formatting occur automatically. Error recovery is identified as the second major design factor.
and briefly described. The system cannot quit when a user enters an erroneous command but must recover gracefully. Valuable information must be protected from loss due to user error. Reliability is identified as another important factor. Also, the design must be well-organized because of the size of the program. A flexible design is desired to allow for enhancements of more advanced features.

"Development of an Experimental Display Word Processor for Office Applications" [18] furnished the following design information. This software design was heavily influenced by considerations of response, ease-of-use and extensibility. A key objective in the design was to emulate the typewriter in order that the user could instantly and accurately be shown the status of the document being edited or formatted. Minimization of operator fatigue was considered to be an important factor. Displays were used to simplify the user's job and the 'soft copy' eliminated the sluggish response of a hardcopy terminal. Unlike 'blind' systems where the operator must keep a mental image of the desired finished output and maintain knowledge of the location of material in order to edit or format, this display-based system performed these activities for the operator. The results of keyboard editing and formatting actions were instantaneously displayed. The keyboard and the soft copy provided by the display served as the primary interface for user/system
dialogue. User error was reduced by differentiating between text and command input. Although the command repertoire was extensive, users were found to rely upon only a few basic commands.

In addition to the factors mentioned in the previous paragraph, "Overview of the Functional Features and Software Design for a Display Word Processor" [14] discussed further efforts made to aid the user. Information on the CRT requiring operator interaction was displayed using menus. A message area, "status and alarm area", was used to display messages to the user which indicated the current command, system status and error conditions. The HELP display provided information pertaining to the user's current state, e.g. current valid commands.

"Text Handling in an Automated Office" [17] expressed the design views found in the following paragraphs. The design aims in this project were to achieve a good human interface and an efficient implementation. A good human interface meant that the system was easy-to-use and easy-to-learn and could be basically described as 'friendly and forgiving'. An efficient implementation meant that the provided services were adaptable (easily modified or expanded), many concurrent users could be supported and machine resources were utilized efficiently. Emphasis was placed upon streamlining the human interface, making
services adaptable to changing requirements and utilizing computer resources efficiently.

Features that enhanced the human interface were a unified command structure, a message area and a uniform mechanism to store and display text. A good human interface was achieved when actions common to different services were initiated by the same commands. Full-word wraparound and rudimentary text formatting, such as default margin settings, double spacing, etc., to make the text more readable, were provided during text insertion to improve the human interface. Full-word wraparound automatically forces a word which would extend into the right margin to be moved to the next line down. The message area was set aside for command entry, i.e. to display and edit commands during their composition; error messages; machine status; etc. It was protected, i.e. no user services could be invoked in it. The cursor served as the pointer to the text being edited. Commands for moving the cursor and positioning the text window, i.e. the block of text currently displayed in the screen, were uniform across all services.

Efficient utilization of computer resources was obtained by storing as little information about the screen as possible, avoiding duplication of code and minimizing the communication between screen and the computer. This system avoided duplication of executable code by unifying
the various constructs, operations and data structures
common to a number of services and implemented them as a
common pool of subroutines. In addition, each service had
its own special purpose routines. This design strategy
made code more compact, but also—even more importantly—
made the system more adaptable. Existing services
could be modified by adding new special purpose routines or
by enhancing the common routines. The same commands in
each service performed a common action making the user's
job much easier.

Based on these observations, the following design goals
for the SCOPE system were set to achieve a friendly,
forgiving and easy-to-use system:

1. to utilize the display or screen as the
   editing and formatting vehicle,
2. to provide extensive error-checking and
   graceful error recovery,
3. to develop a uniform command structure of
   brief mnemonic commands,
4. to incorporate a display message area for
   system-user communications,
5. to assist the user through the presentation of
   menus and a current command status display,
6. to develop a reliable program,
7. to facilitate program adaptability and maintainability,
8. to utilize computer resources efficiently, and in addition,
9. to provide hardware independence in the use of printers and terminals, i.e. to specifically eliminate the requirement for specialized hardware.

The overall SCOPE design is simple and straightforward. Program reliability and maintainability were much more important than the considerations of speed and storage. This does not mean, however, that processor and storage resources were inefficiently utilized. Tradeoff decisions during the design process were made in line with the following list of priorities, in order of importance; ease-of-use, program reliability, program maintainability and finally efficient utilization of machine resources.

C. THE DESIGN PROCESS

The design process proceeded in phases. It was iterative and incremental in nature. The initial phase dealt with the consideration of terminal characteristics. Phase Two defined the overall system structure and capabilities. Subsequent phases dealt with the definition of new features and sometimes with the redefinition and enhancement of existing ones.
1. Terminal Characteristics

The first step in realizing the desired product was the consideration and selection of a minimum set of terminal characteristics necessary for a screen-orientation. Only CRT-equipped terminals were considered for this reason. Specifically, two dumb terminals, the ADM-3A [1] and the DM-1520 [8], were studied because they were used in the implementation. In addition, the DM-2500 [9] was considered to add a greater degree of generalization and hardware independence. Ideally, the terminal used would have an extensive keyboard with many special function keys, e.g., delete, insert, etc., and a CRT screen large enough to display a full page of output text at one time. This system, however, is not intended to run on the ideal terminal but on a large number of widely available dumb and intelligent terminals.

To achieve the previously mentioned design goals of generality and hardware independence, the terminal characteristics chosen for inclusion in the minimum required set are present on most dumb terminals. The following terminal characteristics are necessary for successful implementation of this project:

a. standard ASCII character set (upper and lower case, numerals and punctuation marks),
b. nondestructive cursor movement in four directions (up, down, right and left),
c. CRT screen dimensions that allow the display of 12 or more lines with 60 or more characters per line,
d. communication rate of at least 1200 baud,
e. hardware scrolling feature (cursor movement past the bottom line on the screen causes all the above lines to move up one line),
f. absolute x-y addressing,
g. clear screen,
h. carriage return,
i. bell (used to alert user when an error is encountered), and
j. control code generation.

Upon entry into the system, the user will indicate which terminal type is being used. The system will then make the necessary ASCII code assignments to variables holding the codes for such display functions as clearing the screen or moving the cursor non-destructively to the right. In addition, the user will indicate the printer type to be used for the generation of hardcopy output.

2. SCOPE Features

The next phase in the design process was spent defining the features or capabilities to be included in
SCOPE. The text processing systems reviewed in section II.D. were the sources for the chosen features. Each major feature is listed and briefly described in this section. For purposes of clarity, the features to be included in the SCOPE design are grouped into six categories.

a. File-Handling

SCOPE needed to be able to create new files, save files and access previously saved files. A hardcopy printout of the processed document should be readily available and may or may not be produced directly on the printer. A listing file may be created and sent directly to the printer or saved in storage for multiple hardcopy production at the user's discretion. SCOPE generated files are incompatible with other UNIX editors and formatters due to embedded format commands. For this reason, SCOPE should allow the user to put files together in various ways within the buffer and to write out partial buffer contents as a file. Any SCOPE file may be added at any location within the buffer as long as there is enough space in the buffer. In addition, any segment of text in the buffer may be copied out and saved as another separate SCOPE file. These two features should facilitate the "cut and paste" aspects of document reorganization.

b. Screen-Orientation

SCOPE should be a screen-oriented or display-based system. The screen will provide the user
with a 'soft copy' of the finished document. The visual display will serve as the editing and formatting vehicle. The user should be made immediately aware of the results of any editing or formatting commands. No guesswork should be involved. A screen-orientation should take advantage of the visual nature of the editing and formatting process.

Cursor positioning will determine the text displayed to the user in the text window on the screen and will pinpoint the location of most command actions. A reasonably fast system response is required to maintain the correspondence between the user's commands and the cursor movement. A slow response, typical of a busy timesharing system, disrupts this correspondence so that user entries appear to be ignored by the system thereby confusing the user and causing him to introduce errors by duplicating entries and receive unwanted results.

Cursor movement should occur in one of four directions at a time, right and down for forward movement, and left and up for backward movement through the text. Movement of the cursor should proceed a character at a time or a line at a time. Wraparound from the end of a line to the beginning of the next when going forward, or from the beginning of a line to the end of the previous one when going backward should be provided. A carriage return should place the user at the first character of the next line down. Other commands should be available to speed
movement through the text by scrolling through the text a screenful of lines at a time or a page of lines at a time, where a page reflects the hardcopy page length. Context should be maintained during rapid movement through the file by overlapping or repeating a few lines from the last screen display of text when scrolling a screen at a time. No overlap need occur when paging through the text. In addition, the user should be able to jump to the beginning or end of the file upon command.

c. System-User Communications

SCOPE should be an interactive text processing system. The screen should be the primary interface for system/user dialogue. Being interactive, system response time will be a major determining factor in SCOPE's success. The user will require instantaneous echoing of his input for verification purposes and should be able to make corrections by erasing input characters. System-user communications are not minimized in this design because many SCOPE operators will be casual users. Such users will not want to spend a lot of time relearning the system.

A message area will be a specially defined-area on the screen for the input and display of user commands, the display of error messages, the display of current command status, i.e., which commands are currently available to the user, and the display of prompts and menus. This message area should be protected from other system features.
except in the case of the terminal's hardware scrolling. The user should be coached during a SCOPE session through the use of menus describing the available options and prompts describing the command action and how it is initiated.

There should be a uniform command structure of short mnemonic commands. The same command should cause the same action across the provided services, e.g. a 'q' will allow the user to exit or quit any service. A relatively small set of powerful commands should be available. Text input and command input should be differentiated for the user. It is important to keep the user informed of processing actions not apparent to him, e.g. writing of a file, by signaling their successful or unsuccessful completion. System messages to the user should be conversational in tone and understandable. A cryptic message only serves to confuse and alienate the user and to introduce error.

d. Error Handling and Recovery

SCOPE should attempt to prevent user error as much as possible by providing extensive error-checking. As mentioned previously, slow response time will increase user input error as typographic errors become more frequent. User command and text input should be carefully screened to recognize invalid commands or nontext characters as soon as possible. However, since all user errors cannot be
anticipated, the design of the system should attempt to minimize the impact of those that do occur. In general, error handling should consist of providing the user with a descriptive error message, forcing active user recognition of the error condition and finally re-prompting to provide information about valid command options available. Graceful error recovery should be attempted and accomplished except in the case of a catastrophic error occurring to the base system. The user should be protected from the loss of text by requiring positive confirmation of its deletion or the provision of backup files.

e. Editing

SCOPE should provide editing features to facilitate the composition of documents and programs. These editing features may be invoked during initial creation or after creation and storage. During the editing session, the user should be able to move the cursor or issue scrolling commands for forward and backward movement through the text to pinpoint the location of the edit action. A standard/default set of formatting specifications should be in effect unless the user sets other specifications. The following list of specific features should be available through the SCOPE editor.

- Insert--should allow the user to add text to the buffer before the character at the cursor. Input text should be screened for invalid entries and displayed on
the screen. The user should be able to erase input
characters. If the text is being filled, automatic
full-word wraparound should be provided at the end of the
line. User acceptance or rejection should determine
whether the added text remains in the buffer. The user
should be protected from buffer overflow.

- **Delete**—should allow the user to remove
text from the buffer. A character or an entire line may be
deleted at a time. Cursor movement should indicate where
the deletion occurs. Up or down movement should delete a
line, while right or left movement deletes a character.
The direction of deletion, forward or backward, should be
consistent during the deletion process. Contradictory
cursor movement should cause the last deleted line or
character to be restored to the screen and in effect negate
the deletion process. Actual removal/deletion of the text
should only occur within the buffer upon positive user
confirmation. Until this confirmation, the text will only
"appear" to be deleted on the screen.

- **Locate**—should allow the user to pinpoint
the first location of a specified pattern or string of
characters if it occurs within the text. This global
search should begin at the current cursor position and
proceed in a forward direction. Only a successful search
will reset the cursor position to the beginning of the
"located" pattern. In the event of an unsuccessful search,
the cursor and the text window remain unchanged.

- Replace—should allow the user to substitute one string of characters for another regardless of their respective lengths at every occurrence of the target within the remainder of the text. This global substitution should begin at the current cursor position and proceed in a forward direction only. The user should be protected from buffer overflow.

- Exchange—should allow the user to substitute one string of characters for another of the same length at the current cursor position. As the cursor moves forward the user input character will overwrite the existing buffer contents.

- Copy—should allow the user to duplicate parts of text within the buffer. The user should be able to delimit the material to be duplicated and position the cursor at its new location. Copied text should be placed in front of the character at the cursor. The user is protected from buffer overflow.

- Move—should allow the user to reposition a segment of text within the buffer. The user should be able to delimit the text to be moved and position the cursor at its new location. The delimited text should be placed in front of the character at the cursor.

- Extract—should allow the user to write out a portion of the text to another SCOPE file. The user will
delimit the body of text to be copied to another file and then initiate the process. The original text should remain unaffected in the buffer.

- Add—should allow the user to add another SCOPE file to the contents of the buffer. The cursor position will indicate the position of the addition. The new text is placed before the character at the cursor. The user should be protected from buffer overflow.

f. Formatting

SCOPE should provide formatting features to improve the final document or program's appearance and thereby increase its readability. The formatting features in the design were principally determined by the requirements of thesis preparation. During the formatting session, the cursor should mark the position of the formatting action. Formatting specifications may be set before text creation and can be modified at any time. The formatting specifications should be in effect from their initial setting/entry point until changed or nullified. Most formatting will be provided automatically on the basis of a standard/default set of specifications derived from the NPS Thesis Manual. The following list of specific features should be available through the SCOPE formatter.

- Margin Setting—should allow the user to set or change the left, right, paragraph, top, bottom and page margins at any time.
- Page Length—should allow the user to set or change the number of lines that can be written to the hardcopy output.

- Line Spacing—should allow the user to set or change the space between lines of text. Single, double and triple spacing should be provided.

- Skip Lines—should allow the user to add extra spacing between lines by skipping a specified number of lines or writing a specified number of blank lines.

- Page Numbering—should allow the user to number or not number the pages of text. The user should also be able to increment the page number on a specific page.

- Page Eject—should allow the user to end a page at will and skip to the top of a new page.

- Fill—should allow the user to put as many words as possible on a line of text and will be the default mode. Filling should not apply to irregular lines such as titles or running headers and footers.

- No Fill—should allow the user to place particular words on a line and create irregular lines of text.

- Center—should allow the user to center a line of text.

- Right Justify—should allow the user to align a line of text along the right margin.
- Underline—should allow the user to underline characters within the text. Punctuation marks should not be underlined.

- Adjust—should allow the user to generate filled text aligned along the right margin. Extra blanks will be inserted between words to achieve this result.

- Indent—should allow the user to indent or move the text to the right a specified number of spaces.

- Tab—should allow the user to set tab stops.

- Table-Handling—should allow the user to easily construct tables or columns of text using the tab key.

- Titles—should allow the user to indicate five different types of titles or headers, first through fourth-order and other. The system should automatically format the first through fourth-order titles as per the thesis manual, while the other type is centered on a new page. Header numbering and lettering should be considered for possible inclusion.

- Running Headers and Footers—should allow the user to write a specified message across the top or bottom of every page of the document. The user should also be able to specify where within the top and bottom margin the header or footer is located.

- Paragraph/No Paragraph—should allow the user to define a new paragraph or return the text to
non-paragraph status.

- Footnote—should allow the user to input the information to be contained in the footnote which will then be automatically positioned by the system in accordance with Thesis Manual directions.

D. TESTING

Ease of testing was an important design consideration. An incremental approach should be generally taken in testing. In this method each module is designed, coded and tested by itself and then added to other tested, working modules. Incremental testing greatly simplifies debugging by limiting the scope of the errors encountered. It also allows the discovery of logic and implementation errors early enough to prevent their repetition in other modules and thus eliminates the need for major redesigns once underway.

Two debugging tools were designed to aid the test process. One tool was a procedure which displayed the text and format commands within the buffer upon the screen. The second was a program which provided a listing of the raw file. Both of these tools replace the format commands with special or text characters to allow their display/listing because the control codes themselves are unprintable and could generate unwanted results, e.g. a form feed.
E. DESIGN APPRAISAL

Why is the SCOPE design "better" than other existing systems? A brief comparison of SCOPE and the reviewed systems highlights the critical differences. SCOPE's integrated editor and formatter concept is superior to separate editor and formatter packages, like TED and PRO or Ed and NROFF, because the user performs both editing and formatting while executing a single system. Because of its interactive nature, SCOPE is easier and faster to use than NROFF or PRO. A user can, with SCOPE, change a format specification and immediately see the results rather than having to visualize them. It is no longer necessary to produce a listing or a display in a separate process for verification purposes.

SCOPE provides more extensive formatting features than are available in NED, Vi, or the UCSD PASCAL editor, and supports these features with on-line user aids. The user assistance provided by SCOPE is more comprehensive than that of any of the reviewed systems. The conversational prompts and error messages coach the user through the text creation process. The learning curve for effective system utilization is significantly reduced for the casual users, who should be able to operate SCOPE with a minimum of external documentation.

SCOPE's limited and simple, yet powerful, command structure is designed to be less confusing to the user than
many of the others have proven to be. The mnemonic commands are intended to be easier to understand, remember and to use than those of earlier systems.

Finally, unlike NED and WPS-8, SCOPE does not require a special terminal with specialized function keys and/or an enlarged CRT screen but can be used on a wide range of dumb and intelligent terminals.

In conclusion, the SCOPE design is a "better" design because it attempts to incorporate the most attractive features of the reviewed systems while avoiding the limiting aspects of earlier systems.
IV. IMPLEMENTATION

This section provides general information about the SCOPE system. Program details may be obtained from the program listing (Appendix B) provided. The listing also provides system documentation. An initial introduction to the system is provided along with compilation and execution instructions. A list of abbreviations used throughout the program is included. Each function/procedure is briefly described in the program. Mnemonic names, modularity, straightforward logic and code as well as appropriate comments facilitate program understanding. Each function/procedure is short enough to appear on one page. Indentation and the PASCAL keywords aid in the tracking of program flow. Finally, most variables within the SCOPE program are global in nature.

The SCOPE implementation presented here is characterized by its isolation of implementation details and its modularity. The isolation of as much as possible of the actual implementation from the program as a whole is recommended by Plaugher and Kernighan [15]. This recommended technique facilitates the optimization process or the selection of a new strategy because the desired result can be obtained by simply altering low level buffer management routines. The second attribute of this
implementation, its modularity, is recommended by Sharma in [17]. Modularity makes the provided services more adaptable to changing requirements. There is a common pool of general-purpose subroutines that are called by many of the services as well as special-purpose routines.

During the implementation process the UNIX system provided two handicaps. The first difficulty was the slow response time. UNIX suffers measurable degradation whenever graphics-related programs are executed and/or multiple users are on the system. The second difficulty had to do with the special terminal conditions under which SCOPE operates, which are explained in section D. These conditions were a nuisance when trying to execute commands and programs outside of SCOPE. The frequent and necessary changes of the terminal conditions were not only inconvenient but tended to increase user error.

A. PROGRAMMING LANGUAGE SELECTION

An important consideration in the system implementation effort was the selection of a suitable programming language. A high level language was sought. Desirable features included structured constructs, flexible and extensive data structures, and extensive error-checking and diagnostic facilities. In addition, the language had to be able to distinguish between every character input at the keyboard—especially control codes which serve as commands. Secondary selection criteria were availability
(unfortunately the new PL/I compiler by Digital Research was not available) and transportability. PASCAL met all of the above mentioned criteria and was chosen. Specifically, the Berkeley Version 1.1 [10] PASCAL implementation was selected because of its attractive run-time options. A PDP-11/50 minicomputer under the UNIX timesharing system [23] served as the host computer.

8. PROGRAM LOGIC AND STRUCTURE

Software engineering principles [12] were applied in developing the program structure. The top-down design approach was followed. Major functions to be performed were analyzed and decomposed into smaller, more easily managed subfunctions. Functional modules were designed to be subprograms, i.e. functions or procedures, and were limited to one page. During the analysis and decomposition phase, SOFTWARE TOOLS [15] and Triyono's thesis [21] provided invaluable assistance.

Coding and debugging were simplified and overall program clarity improved by always selecting the simplest and most straightforward technique to accomplish any "tricky" or complex actions. In addition, a single prompting procedure and a separate error message procedure were set up to centralize system communication with the user. Finally, tradeoff decisions were made whereby program simplicity, reliability and maintainability were considered more important than speed, storage or overall
efficiency.

C. DATA STRUCTURES

The Berkeley version of PASCAL does not provide string manipulations and it does not allow the reading of strings from text files. In addition, it allows only scalars--integer, characters, or boolean values--to be passed as subprogram arguments. Finally, set implementation is such that sets may not be declared as constants, nor do sets defined to contain character variables assigned using the standard function CHR behave correctly. Despite these shortcomings, data structure selection was not a difficult phase. In order to facilitate changes the actual data structure is hidden from higher level modules by a low level buffer management function designed only to manipulate the data structure.

Within this system there are two principal data structures. The first is the "raw" text file as it exists in storage when not being acted on. It is a sequence of character data. It consists of the text characters and format command/characters. Integers, reals, booleans, strings and unstructured data types may not be included in this file. This file contains a beginning and end of text character or marker. Within the text, the lines and the paragraphs are set off by a line or paragraph marker respectively. Although this line orientation is not strictly necessary it does simplify many operations,
particularly the mapping from screen to buffer explained in
Section IV.C. The additional system overhead in
maintaining these line markers is a small price to pay for
the simplification. A paragraph marker, once set by the
user does not change its relative position in the text
unless the user resets/deletes it. A line marker, on the
other hand, may be frequently shifted if as many words as
possible are being placed in a line, i.e. filled text, or
text is inserted or deleted before the line marker. Line
and paragraph markers occur at the beginning of a line or a
paragraph.

The format commands, other than markers, embedded
within the text indicate special processing that is to
occur to the raw text before display or production of a
listing can occur. A format command's range of effect can
vary from only one character to the entire file. If the
text file is created using a non-standard format, the
characters in the file immediately following the beginning
of text marker are grouped together and called a "format
record". This record only contains those user format
specifications that differ from the standard menu provided.
It should be noted that the raw file is useless unless the
embedded format commands are masked out and processed. The
standard PASCAL [11] procedures READ and WRITE and function
EOF (end-of-file) are used to manipulate data within the
text file.
Space usage was carefully considered. The line and paragraph markers and format commands could add a significant amount of overhead to the system by taking up unnecessary space that could be used for text. This is not the case, however, because most markers and format commands, however, overlay what would be a blank between words. Extra blanks associated with indentation, tabulation and the end of a line can be eliminated or compressed. Within a line two spaces following the punctuation marks, '.'., ':' or ';', and '?' will not be compressed because the mapping is simplified. Two procedures PROCESSBUF and PROCESSRAW eliminate and compress extra blanks as well as insure that punctuation requiring two spaces is followed by two spaces where appropriate.

The most important data structure within the SCOPE system is the text buffer. Figure 4-1 is a diagram of the SCOPE buffer. Text and format commands, such as OM, reside in this data structure while undergoing user operations. This buffer is a large array of characters of approximately 30,000 bytes. There are several integer pointers indexing this array. BUFINDX is the current location of operation within the buffer. INSBEG and INSEND are pointers which delimit the INSERT AREA within the buffer. BUFLIMIT is the last position in the array. CH is always the current character at BUFINDX.
Movement through the buffer is accomplished via the GFTRUFCCHAR (Direction) function, except in the INSERT mode, where Direction is +1 for forward movement, 0 for no movement, and -1 for backward movement. The text character immediately preceding the INSREG location appears on the CRT screen at the cursor. Moving forward, i.e. right, down or scrolling down, through the text is performed in the following manner:

GETRUFCCHAR(+1) which means that

\[
\begin{align*}
\text{BUFINDX} & := \text{BUFINDX} + 1; \\
\text{INSREG} & := \text{INSREG} + 1; \\
\text{INSEND} & := \text{INSEND} + 1; \\
\text{BUFFER}[\text{BUFINDX}] & := \text{BUFFER}[\text{INSEND}]; \\
\text{CH} & := \text{BUFFER}[\text{BUFINDX}].
\end{align*}
\]

Moving backward, i.e. left, up or scrolling up, through the text buffer is performed in this manner:

GETRUFCCHAR(-1) which means that

\[
\begin{align*}
\text{BUFFER}[\text{INSEND}] & := \text{BUFFER}[\text{BUFINDX}]; \\
\text{INSEND} & := \text{INSEND} - 1; \\
\text{BUFINDX} & := \text{BUFINDX} - 1; \\
\text{INSREG} & := \text{INSBEG} - 1; \\
\text{CH} & := \text{BUFFER}[\text{BUFINDX}].
\end{align*}
\]
GETBUFCHAP(0) allows the system to re-reference the current buffer location. All of the buffer pointers remain unchanged. GETBUFCHAP cannot be used in the INSERT MODE for reasons explained in Section IV.H.

READRAW handles the buffer initialization for both new and existing files. A file must be read in and initialized before EDIT or FORMAT processing. Buffer initialization during the creation of a new file involves entering the beginning and end of text markers, entering an initial paragraph marker and setting the INSBEG, INSEND and BUFINDX pointers. Remember that the current INSBEG and INSEND are empty. Initialization for an existing file is somewhat more complicated. First, the raw text file is read into the buffer character by character where the first character goes into the first buffer position, the second in the second position, and so on. After the file is read, the text must be shifted so that the INSERT AREA is correctly positioned. The last character read in is moved to the second to the last buffer position, the preceding one to the third to the last position, and so on until the beginning of text (bot) character is reached. It is not moved. The pointers, INSBEG, INSEND and BUFINDX are set and in this manner the INSERT AREA is defined. This INSERT AREA is the key component within the buffer. Text insertion, deletion, copying and movement is accomplished by resetting INSBEG and INSEND. More details of these
operations may be found in the program listing. Format commands and line and paragraph markers within the buffer are totally transparent to the user who only sees them reflected on the screen.

Several other arrays are used within this system. The running header and footer lines are each stored in an 80 byte array of characters. Two other arrays, of 20 integers each, are used in the TABULATION and ADJUSTLN routines. The column number(s) of the tab location(s) is(are) stored in TABLOCNS. Similarly, the relative location of blanks within a buffer line of text are saved in BLKLOCNS. These stored values indicate the sites for the insertion of extra blanks within the text line to achieve right justification with filling.

0. INPUT/OUTPUT HANDLING

The use of a high level language such as PASCAL eliminates the messy details of input/output (I/O) handling. I/O to and from the screen is accomplished using the standard PASCAL READ and WRITE procedures on the text files INPUT (keyboard) and OUTPUT (CRT screen).

Correct SCOPE operation requires the setting of certain I/O options on the current terminal. The UNIX command "stty raw -echo nl" sets the necessary options. RAW means that no erase, kill, interrupt are EOT characters are in effect and that the parity hit is passed back. NL tells the system not to automatically generate a carriage return.
and line feed for a new line and that only a line feed will end a line. It also prevents the use of the READLN and WRITELN procedures. ECHO prevents the system from echoing back input characters to the CRT screen.

SCOPE filters the input characters and selectively echoes back or writes to the CRT screen, preventing the display of unprintable control characters. A typical case is the handling of cursor movement in which the user generates right, left, upward or downward movement of the cursor. INCHAR is the character read in from the user.

In addition to READ and WRITE, the RESET and REWRITE procedures are used in I/O to and from other files. INFILE and OUTFILE are text files assigned specific names by the user, e.g. CHAP1.o and LISTING.o. The '.o' indicates a PASCAL file. RESET is used to open existing files for reading, while REWRITE is used to open and create a new file. INFILE is used in the reading of raw text files and OUTFILE in the saving of raw files and the production of listings. READLN and WRITELN cannot be used on either of these files because the UNIX eoln (end-of-line) character does not exist in SCOPE generated files. UNIX writes OUTFILE in blocks of 512 bytes. The Berkeley PASCAL function FLUSH is required to dump the last incomplete block of output to the waiting file.
The lack of string manipulations causes all I/O to be done a character at a time which is somewhat tedious. DISPLAYLN performs the necessary processing to display a line of text on the CRT screen. LISTLN produces a line of text for a hardcopy listing. Both of these procedures call routines which process any embedded format commands and only portray the "processed" text as opposed to the "raw" text in the buffer. LUSERINPUT and SUSERINPUT, for long and short user input from the terminal, perform the READ function. They allow the user to enter a string of characters with the option of erasing any mistakes by typing a "control e" before hitting the carriage return. LUSERINPUT allows up to 14 characters to be input and is used in the naming of files. It automatically checks each entry for a '.P' in the name and appends it where necessary. SUSERINPUT allows up to 4 characters to be entered at one time. Two procedures are used because strings must be padded with extra blanks to their full declared length. The slow response time, due to UNIX degradation, was a problem. Not only is it disconcerting to be typing several words ahead of what is being displayed on the screen, but immediate error correction becomes difficult. It is no longer a matter of simply erasing one or two characters that are incorrect. Instead, the user might have to erase and lose several words to get back to the error. In addition, during the erase action when the
cursor is moving backwards a delay can cause the user to inadvertently erase more than intended.

Because only character data is read and written by the SCOPE system there is no way of directly entering integer values for such items as page numbers or margin settings. To overcome this deficiency, CONVRTNUM transforms a character number into its equivalent integer value. Up to a three digit number may be converted. It currently handles only positive values.

E. THE SCOPE SYSTEM

Implementation proceeded in an incremental fashion. The initial phase implemented a minimum set of system features. The minimum set of features allowed the production of a simple paper but not of this thesis. This skeletal system was a barebones approach with many options eliminated, e.g. the basic formatting specifications setting up the page layout (margins, lines per page and line spacing) were defined as constants. During the analysis of system features, the basic system design evolved into four nested levels or modes. At the highest or outermost level, i.e. global mode, the system reads or writes files, edits, formats or leaves the SCOPE system based on the input user commands. The global mode (SCOPE DRIVER/MAINLINE) is the driver of the entire system. Figure 4-2 is a pseudo code representation of the SCOPE DRIVER.
The second level consists of the following essential system features:

a. read in an existing file or create a new file,
b. write the raw file out to storage,
c. edit the text file,
d. format the text file in a reasonable manner,
e. prepare a hardcopy listing of the file, and
f. exit from the system.

Each of the above features, except system exit, is a major module within SCOPE. Within this level lies the third level of modules or subfunctions, e.g. INSERT and DELETE for EDIT. At the innermost and lowest level are the functions to manipulate the data structures and other support routines, e.g. cursor movement and mapping. Cursor movement and mapping are key modules which are discussed in Section IV.G. The movement of the cursor on the CRT screen must be correctly reflected in movement through the text buffer. There must be a one-to-one correspondence between the character at the cursor and this character's location within the text buffer.

Entry to lower levels, e.g. subordinate modules, is accomplished by calling the required procedure or function. Return to a higher level is usually accomplished via a "quit" or "control q" command.
F. COMMAND STRUCTURE

It is important at this point to reiterate the chief design goal of this system which is to develop a friendly and forgiving tool. Users will probably not be using this system on a daily basis and so will not be intimately familiar with commands. Such casual users will not utilize the system if a lot of time must be spent learning and relearning the command set. For this reason, it was considered imperative to have short, simple, easy-to-remember and easy-to-use commands. The command set is also consistent throughout the different SCOPE modes of operation.

This is an interactive, screen-oriented system and these features are exploited. Three lines at the top of the CRT screen are set aside permanently as a message area for the display of prompts and error messages and the entry of some user commands. Entry and exit from any of the three outer SCOPE levels causes a prompt to be generated in the message area. This prompt tells the user which commands are currently available to him for execution and how to initiate execution.

Commands are mnemonic and one character in length. The command is usually the first letter of the action to be performed. Some commands, e.g. page number, require entry of a numeric argument. Many commands are dependent upon cursor movement. In some instances the control key is used
in conjunction with another character to differentiate
between a command and a text character, e.g. in the insert
mode. The EDIT and FORMAT commands are described in more
detail in sections IV.H, IV.I.

G. CURSOR MOVEMENT AND MAPPING

Establishing and maintaining the correct relationship
between the screen display and buffer was the most
difficult aspect of the SCOPE implementation. There has to
be a one-to-one correspondence between the character shown
on the screen and its location within the buffer. Format
commands are transparent to the user but must be reflected
on the CRT screen, e.g. a paragraph marker is shown by
indenting the line. Cursor mapping is carried out by the
MAPCURSOR function which processes the format command and
sets up the screen display as necessary. SETCURSOR is
another mapping function which comes into play during the
scrolling actions.

There are two methods of cursor movement available and
both are presented in figure 4-3. Movement within a single
line is fairly simple but wrapping around to the previous
line, or jumping multiple lines complicates the picture
quickly. Cursor movement is simplified by making it a
line-oriented process. STARTLN and NEWLINE are key
functions in determining the correct user line location.
GETCHAR is used to produce movement one character to the
left or right. JUMPTOMARKER allows the user to jump to the
beginning or the end of the currently active file. Further
details about the cursor mapping and movement are available
in the program listing.

H. EDIT

EDIT is probably the most important module within the
SCOPE system. It is used to move the cursor and thus scan
through the body of text as well as to make modifications
to the text. The EDIT commands are listed in Figure 4-4.
The upper case letters are used to allow the same beginning
letter to be used, e.g. x for xchange and X for Xtractfile.
In addition, the capitals indicate file manipulations where
the lower-case commands only deal with the buffer. The EDIT
commands closely follow the design, with the
exception of SETMARKER which is used by almost all of the
other EDIT features to delimit the body of text to be
searched, copied, moved, extracted, etc. Many of the EDIT
features remain to be implemented.

The initial implementation involved only INSERT and
DELETE. INSERT adds text directly to the buffer through
the INSERT AREA. Because GETBUFCHAR, the low level buffer
management function works on the outside of the INSERT
AREA, it can not track the new characters entered into the
INSERT AREA. All inserted and deleted text must be
processed by PROCESSBUF to ensure that the line markers are
correctly positioned. This processing occurs in paragraph
increments because a paragraph is the smallest text
segment. During the text insertion process, certain formatting commands are entered directly by the user into the buffer. Irregular lines during the fill mode must be ended with a break character. Titles may be labeled at this time. In addition, the fill, no fill, adjust, adjust off and the tab commands may be entered.

Along with INSERT and DELETE, an XCHANGE function was set up and used. This allowed the replacement of many strings of the same length. Because the string is the same length during an XCHANGE there is no need to process the the modified text which is more efficient than reading and processing at least one paragraph to reposition line markers if necessary. Several users have enjoyed the automatic wraparound feature which speeds up the input process.

I. FORMAT

FORMAT implementation has been limited to titles, centering, tabs and other essential features for thesis preparation. The following paragraphs describe the intended SCOPE FORMAT implementation.

The Format mode can be entered at any point during the editing session to change the output specifications. It also moves the cursor to scan the text, and in addition, some commands depend upon the cursor for proper execution. The user moves the cursor to the position from which the format command will begin to have effect, enters the
correct format command and is then prompted for the correct command arguments. Formatting changes are immediately reflected on the screen for user examination.

Figure 4-5 outlines the FORMAT command structure. User commands are alphabetic characters but the format commands in the buffer are control codes. All the format commands were assigned control codes so that they could be easily distinguished from text characters. The decimal ASCII code is the argument for the PASCAL CHR function which is used to assign and identify all the format commands within the buffer. The keyboard control characters are used infrequently by the user during the editing and formatting process. Certain commands, e.g. break, adjust, adjust off, fill, no fill, paragraph marker and tab may be entered directly by the user. Within the buffer, the numeric arguments for any format command, immediately follow the command and are themselves followed by a break character so that they may be easily distinguished from text numerals. Whenever possible, the input command (command syntax) is mnemonically indicative of the action to be performed, e.g. 'u' for underline. The slashes in certain commands indicate that the command is complex, has multiple arguments and that the user will go through a series of prompts during command entry. Margins are a good example in which the user makes three different entries to set a specific margin.
Menus such as those presented in Figures 4-6 and 4-7, are displayed upon user request to describe the current output specifications. Figure 4-6 will be displayed unless the user has made some format changes. Figure 4-7 will be displayed, if any of the default specifications have been altered.

J. WRITE RAW FILE

This module is the simplest of the main modules. It writes out every text character and format command, except a rubout character, within the buffer to the user-specified file. The rubout character is used for the deletion of characters or spaces when it is inconvenient to manipulate the INSERT AREA to do so. In this manner SCOPE files are saved for future use.

K. LIST PROCESSED FILES

Hardcopy printouts or listings of the text file are required whenever the screen display is not adequate, e.g. proofreading is often more easily accomplished on paper. This process is hardware dependent because printers have various drivers, carriage control mechanisms and top-of-form alignments. To provide some degree of hardware independence, SCOPE includes a procedure PRINTER which queries the user about the printer and makes appropriate margin adjustments.
The actual printing process is carried out in two steps. In the first step, the format commands within the buffer are processed and the requisite characters, e.g., blanks for margins or carriage returns and line feeds for line and paragraph markers, along with text characters are written to a user-specified output file. This output file contains only text characters, i.e., blanks, alphabetic, numeric and special character data, and carriage returns and line feeds. The second step of the printing process must be done outside of SCOPE execution and is accomplished using the Shell command "cat filename.o;lor" [23]. Unless multiple copies are being made, this method wastes secondary storage and should be changed before a production status is achieved. A direct one-step listing generation is more efficient.

L. SCREEN DISPLAY VICE LISTING

Although the design called for an exact duplicate of the final hardcopy output on the screen, that is not the case in this implementation. There were several reasons for this design departure. Figure 4-R is a diagram of the CRT Screen Layout. Since the area of the screen available for the display of text is limited (only 21 lines here), double-spacing only allows the user to see a small segment of text (11 lines). Better use of the screen occurs when the text is single-spaced (21 lines of text are then displayed). Figures 4-9 and 4-10 illustrate this point.
The screen depiction also does not show adjusted lines of text. Mappin gthe cursor to the buffer contents becomes much more difficult and requires the maintenance of information about each displayed line if wraparound and other features are continued in their present form. The top and bottom margins which include the running header and footer and page number do not appear on the screen. Finally, the representation of underlining is difficult because the underline character overwrites the text character at the same position.
V. FOLLOW-ON WORK AND ENHANCEMENTS

Although the SCOPE system was used in the production of this thesis, more implementation and testing must be carried out before the system is ready for release to general users. The SCOPE program contains untested code segments, e.g. ADJUSTLN. Several important editing features require implementation. The COPY feature has not been fully tested. MOVE has not been implemented. The COPY, MOVE and XTRACTFILE features need to carefully check the format specifications in effect at the original segment location. This may mean going back to the beginning of the file and checking all the format commands between the beginning and the start of the text being acted on. The LOCATE feature to do pattern searches can be implemented using the algorithm in reference (3). To be successful, LOCATE must identify a pattern that has embedded format commands. REPLACE will use LOCATE to find the patterns to be replaced.

Many of the FORMAT specifications are currently fixed within the program and cannot be modified directly by the user. FORMAT command sequences were designed but not implemented for commands requiring arguments. In addition, table-handling and footnote mechanisms must be completed. The adjusting of text must be implemented for hardcopy
orintouts. Running headers and footers should be added.

Upon completion of the implementation, systematic testing of the SCOPE system should be conducted. A user-oriented SCOPE Manual should be developed. A UNIX Shell program should be developed for system initialization and termination. This program would handle the setting of the terminal stty options and initiate execution.

Once in a production mode, "bugs" uncovered by the users will have to be corrected. Surveys of SCOPE users should be conducted to determine suggestions for streamlining prompting and command sequences, overcoming the most common user errors, and enhancing the system. An efficiency analysis, using profiles, should be conducted to determine the most suitable areas of code for optimization.

This system is intended for use in the preparation of programs as well as documents, but so far no work has been done in that area. The format commands, being control codes, will have to be masked out or processed before a SCOPE file can be used as input to any compiler.

There are many possibilities for system enhancement. SETMARKER could be expanded to allow the user to set other markers within the file to pinpoint special locations. The adjusting of text lines on the screen and the associated mapping to successfully handle the extra embedded blanks would be a worthy project. Underlining could be represented on the screen as another line of blanks and "-"
characters underneath the required position.

The page concept on the screen where the page margins are actually shown delimited on the screen with the "!" character down each side for instance and the top and bottom of the page indicated by dashes ("-" ) across the screen could be introduced. In this manner the screen becomes a more exact copy of the final output document. Carrying this further, the page scrolling mechanism would not only scroll through one hard copy equivalent of a page but allow the user to request a particular page by number.

An UNDO command allowing the user to easily recover from his last action might be helpful. Another valuable feature would be the provision of a document index containing pertinent information about the document such as its title, date of last update, size, etc. This index would be readily accessible to the user from the command level. The introduction of a macro-processing capability would be another possibility for enhancement. Finally, common document formats could be stored as templates to be easily accessible to the user.

This design could be implemented on other systems. The INTEL-based 8080/Z-80 microprocessors are prime candidates using the new PL/I compiler by Digital Research. Another candidate for SCOPE implementation is the IBM system scheduled to be installed at the W. R. Church Computer Center in the fall of 1980.
VI. CONCLUSIONS

The design criteria for a "better" text processing system were discussed and a design was developed. This design was based upon an examination of currently available systems and a review of the design criteria set forth by recognized designers such as Kernighan and Plaugher and Xerox personnel. SCOPE's screen orientation, interactive nature and integration of services are the keys to this better design. The implementation demonstrates the feasibility and value of such a comprehensive system. It is hoped that the design criteria presented here can serve as a guide in the development, evaluation and selection of commercial text processing systems, as well as presenting a basis for the design of new text handling systems.
the above buffer contents would appear on the screen as:

```
SCOPE System.
```

where '=' represents the cursor position.

Figure 4-1. SCOPE Buffer Layout
begin  {SCOPE DRIVER/MAINLINE}

initialize variables;
terminal;
printer;
setdefaults;
setformatcmds;
clearscreen;
repeat
promot;
userinput(cmd);
if valid cmd then
  case cmd of
    'e': edit;
    'f': format;
    'l': listprocessed;
    'a': stop := true;
    'r': readraw;
    'w': writeraw;
  end (case)
else (not a valid command)
  errormsg
until stop;
end.

Figure 4-2. SCOPE Driver

71
Method I

↑
Move up 1 line, staying in same column or the one closest to the left.

→
Move right 1 column, at end of line go to first char of next line down.

↓
Move down 1 line, staying in same col or the one closest on the right.

←
Move left 1 column, if beginning of line go to last char of previous line.

Method II

cr
Move to first char of next line down.

'U'
Scroll up, move up approximately $\frac{3}{4}$'s of the screen with 5 lines of overlap.

'D'
Scroll down, move down approximately $\frac{3}{4}$'s of the screen with 5 lines of overlap.

'h'
Go to the beginning of the file.

'e'
Go to the end of the file.

Figure 4-3. Cursor Movement
<table>
<thead>
<tr>
<th>Command Syntax</th>
<th>Command Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Addfile</td>
</tr>
<tr>
<td>c</td>
<td>copy</td>
</tr>
<tr>
<td>C</td>
<td>Cursor Movement</td>
</tr>
<tr>
<td>d</td>
<td>delete</td>
</tr>
<tr>
<td>i</td>
<td>insert</td>
</tr>
<tr>
<td>l</td>
<td>locate</td>
</tr>
<tr>
<td>m</td>
<td>move</td>
</tr>
<tr>
<td>q</td>
<td>quit</td>
</tr>
<tr>
<td>r</td>
<td>replace</td>
</tr>
<tr>
<td>s</td>
<td>setmarker</td>
</tr>
<tr>
<td>x</td>
<td>exchange</td>
</tr>
<tr>
<td>X</td>
<td>extractfile</td>
</tr>
</tbody>
</table>

Figure 4-4. SCOPE EDIT Commands
<table>
<thead>
<tr>
<th>Decimal Code</th>
<th>Command Syntax</th>
<th>Formatting Command</th>
<th>Keyboard Control Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Adjust</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>h</td>
<td>Break</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Center Line</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>UNUSED [UNIX end of file]</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>Page Eject</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>m/n/#</td>
<td>Fill</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>h/h,f/1,m,r/#</td>
<td>Page Number Margin</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>tab key</td>
<td>Running Header/Footer</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>line feed key</td>
<td>Tab</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>UNUSED [UNIX ']</td>
<td>UNUSED [UNIX ']</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>i/#</td>
<td>Indent</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>l</td>
<td>Line Spacing</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>carriage return</td>
<td>Line Marker</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>n++,#</td>
<td>Page Number</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>o</td>
<td>Fill Off</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>control p</td>
<td>Paragraph Marker</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>m/t/#</td>
<td>Top Margin</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>r</td>
<td>Right Justify Line</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>s/#</td>
<td>Skip Lines</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>t/#</td>
<td>Title</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>u</td>
<td>Underline</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>available</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>m/l/#</td>
<td>Left Margin</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>x</td>
<td>Adjust Off</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>control v</td>
<td>Beginning of text Marker</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>control z</td>
<td>End of text Marker</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>b</td>
<td>Begin Delimiter Marker</td>
<td>I</td>
</tr>
<tr>
<td>28</td>
<td>n/b/#</td>
<td>Bottom Margin</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>e</td>
<td>End Delimiter Marker</td>
<td>I</td>
</tr>
<tr>
<td>30</td>
<td>m/r/#</td>
<td>Right Margin</td>
<td>varies</td>
</tr>
<tr>
<td>31</td>
<td>m/o/#</td>
<td>Paragraph Margin</td>
<td>varies</td>
</tr>
</tbody>
</table>

Figure 4-5. SCOPE FORMAT Command Structure
DEFAULT/STANDARD FORMAT MENU

Bottom Margin 9 ins = 1 1/2"
Left Margin col13 = 1 1/4"
Page No. Margin 6 ins = 1"
Paragraph Margin col17 = Lmargin+4
Right Margin col172 = 1 1/4"
Top Margin 6 ins = 1"

Note: 10 chars/inch and 6 ins/inch.

<table>
<thead>
<tr>
<th>Setting</th>
<th>On Screen</th>
<th>On Listing Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjust Mode</td>
<td>off</td>
<td>on</td>
</tr>
<tr>
<td>Fill Mode</td>
<td>on</td>
<td>on</td>
</tr>
<tr>
<td>Line Spacing</td>
<td>single</td>
<td>double</td>
</tr>
<tr>
<td>Max lines</td>
<td>21</td>
<td>66</td>
</tr>
<tr>
<td>Window Size</td>
<td>21</td>
<td>N/A</td>
</tr>
<tr>
<td>Tab</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 4-6. SCOPE FORMAT Menu (Default)
<table>
<thead>
<tr>
<th>CURRENT FORMAT MENU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom Margin</td>
</tr>
<tr>
<td>Left Margin</td>
</tr>
<tr>
<td>Page No. Margin</td>
</tr>
<tr>
<td>Paragraph Margin</td>
</tr>
<tr>
<td>Right Margin</td>
</tr>
<tr>
<td>Top Margin</td>
</tr>
</tbody>
</table>

Note: 10 chars/inch and 6 ins/inch.

<table>
<thead>
<tr>
<th></th>
<th>On Screen</th>
<th>On Listing Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjust Mode</td>
<td>off</td>
<td>off</td>
</tr>
<tr>
<td>Fill Mode</td>
<td>off</td>
<td>off</td>
</tr>
<tr>
<td>Line Spacing</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>Max lines</td>
<td>21</td>
<td>66</td>
</tr>
<tr>
<td>Window Size</td>
<td>10</td>
<td>N/A</td>
</tr>
<tr>
<td>Tab</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 4-7. SCOPE FORMAT Menu (Current)
Figure 4-8. CRT Screen Layout
Although the design called for an exact duplicate of the final hardcopy output on the screen, that is not the case in this implementation. There were several reasons for this design departure. Figure 4-8 is a diagram of the CRT Screen Layout. Since the area of the screen available for the display of text is limited (only 21 lines here), double-spacing only allows the user to see a small segment of text (11 lines). Better use of the screen occurs when the text is single-spaced (21 lines of text are then displayed). Figures 4-9 and 4-10 illustrate this point. The screen depiction also does not show adjusted lines of text. Mapping the cursor to the buffer contents becomes much more difficult and requires the maintenance of information about each displayed line if wraparound and other features are continued in their present form. The top and bottom margins which include the running header and footer and page number do not appear on the screen. Finally, the representation of underlining is difficult because the underline character overwrites the text character at the same position.

Figure 4-9. Sample SCOPE Screen Display #1
Although the design called for an exact duplicate of the final hardcopy output on the screen, that is not the case in this implementation. There were several reasons for this design departure. Figure 4-8 is a diagram of the CRT Screen Layout. Since the area of the screen available for the display of text is limited (only 21 lines here), double-spacing only allows the user to see a small segment of text (11 lines). Better use of the screen occurs when the text is single-spaced (21 lines of text are then displayed). Figures 4-9 and 4-10 illustrate this point.

The screen depiction also does not show adjusted lines of

Figure 4-10. Sample SCOPE Screen Display #2
SCOPE, SScreen-Oriented Processor and Editor, is an interactive, integrated, screen-oriented, context-oriented text processing system. It performs the functions of both a text editor and formatter. The following paragraphs provide a brief description of the intended SCOPE System.

While using SCOPE, the file that is edited or formatted is displayed on the CRT screen which acts as a "window" on the text. This file exists in a stream format, i.e., the file is a continuous stream of character data. In this character stream, the lines of text are defined by a line marker or a paragraph marker which precedes each line or paragraph respectively. Along with the text and markers the stream also contains format commands which cannot be confused with either the text or markers.

The cursor position on the screen serves to pinpoint the text to be processed. Cursor movement occurs in one of four directions: down and right for forward movement and up and left for backward movement. Movement through the body of text is accomplished character by character, line by line, a screenful of lines at a time, or by a page of lines where a page reflects the hardcopy printout length. Rapid movement through the text is performed using commands, e.g., 'D' for scroll down, which allows the user to move forward.
through the equivalent of 3/4's of the screen.

During the editing session, text is inserted in free format with full-word wraparound at the end of each line. The system worries about each end of line thereby freeing the user to concentrate only upon inputting as rapidly as possible. Blocks of text may be copied from one place to another within the same file and to another file. Deleted material is not removed from the buffer until the user positively acknowledges this action thus preventing some unintentional errors. Context searches are also available. Table construction is facilitated by the use of tabs. The format specifications are in effect during the edit process.

In the formatting mode the user immediately sees the processed results upon the CRT screen. The cursor position again determines where the formatting action is to occur. Menus are provided to the user for the selection of desired formatting specifications. There is a default menu that provides the user with a reasonable page layout as defined by the NPS Thesis Manual. Hardcopy printouts of the document may be generated on various models of printers. Centering, right justification, i.e. alignment along the right margin, underlining, running headers and footers and footnoting will be provided.
SCOPE Execution Commands

\% stty raw -echo n1
\% ox scope {does not appear on screen}

Enter the terminal type as follows:
1 for adm-3a, 2 for dm1520, 3 for dm2500

Print listing on UNIX printer--y or n?

COMMAND MODE >
edit format listing o)ut r)eadfile w)ritefile

SCOPE Compilation Commands

\% od -o filename.o {to compile source}
\% -z {to generate profile counters}
\% -o {necessary to turn off stmt}

\% mv obj scope {to correctly name the object module}

\% oxo -j2 filename.o {provides formatted oam listing}

\% oxo -z filename.o {to generate profile limit counter}
program scope(inout, outout, infile, outfile);

(* abbreviations used in SCOPE in alphabetical order...*)

addr : address
b, beg : beginning
b : bottom
blk : blank
buf : buffer
c : center
ch, char : character
chg : change
cmd : command
cnt, cntr : count, counter
col : column
const : constant
convrt : convert
cr, c : carriage return
D : scrolldown
del : delete
dir : direction
e : end
in : inout
i, indx : index
init : initialize, initialization
inc : increment
ins : insert
jmp : jump
l : left
lf : line feed
lm : line marker
l : line
locn : location
m : marker
max : maximum
msg : message
neg : negative
no, num : number
o : of
p, para : paragraph
pq : page
pm : paragraph marker
pons : positive
r : right
t : top, text
u : under, up
U : scrollup
var : variable

end abbreviations *)
const
overlap = 5; (*no of ins to be recopied during scrolling*)
buflimit = 30000;
maxchars = 14; (*limit on user input of long length*)
maxchars = 4; (*limit on user input of short length*)
msaIns = 3; (*no. of lines in msa area*)
line1 = 1;
line2 = 2;
line3 = 3;
neq1 = -1;
pos1 = +1;
zero = 0;
cole = 8;
coll = 12;
col24 = 24;
var

different variables and constants defined:

- **emotybuf**, **fileread**, **namefile**, **stop**: boolean;
- **fill**, **markersok**, **alistind**: boolean;
- **begmset**, **endmset**, **newmset**: boolean;
- **ch**, **cmdchar**, **entchar**, **hell**, **botchar**, **down**: char;
- **accept**, **erase**, **quit**, **om**, **lm**, **lf**, **cr**, **clear**: char;
- **bm**, **em**, **nm**, **inchar**, **home**, **left**, **up**, **right**: char;
- **centerlnch**, **underlnch**, **titlech**, **pjectch**: char;
- **indentch**, **headerch**, **footerch**, **fillch**, **rubout**: char;
- **pgnumch**, **filoffch**, **skiplnch**, **breakch**: char;
- **rtjustifych**, **lmarginch**, **rmarginch**, **tabch**: char;
- **tmarginch**, **bmarginch**, **omarginch**, **pomarginch**: char;
- **adjustch**, **adjustoffch**, **inspacingch**: char;
- **{escape, rubout}{char don't work in raw mode}

- **buffer**: array [1..buflimit] of char;
- **cmd**: array [1..smaxchars] of char;
- **name**: array [1..lmaxchars] of char;
- **footbuf**: array [1..80] of char;
- **headbuf**: array [1..80] of char;
- **underlnbuf**: array [1..80] of char;

- **infile**, **outfile**: text;

- **cmdset**, **edcmdset**, **cursormoveset**: set of char;
  (cursormoveset is only for documentation purposes because this Pascal doesn't allow chr defined variables in a set to be recognized properly.)

- **n**, **bufindx**, **insend**, **insbeg**, **number**: integer;
- **lineno**, **colno**, **lastcol**, **lastln**, **pageno**: integer;
- **termno**, **blnoffset**, **tlnoffset**, **rmargin**: integer;
- **bmargin**, **lmargin**, **ogmargin**, **omargin**: integer;
- **tmargin**, **maxqlns**, **inspacing**: integer;
- **delbeg**, **delend**, **j**, **i**, **beatext**: integer;
- **begindx**, **end indx**, **rewindx**: integer;
- **charcnt**, **centerofpq**, **tab**, **tacnt**: integer;

- **blklocns**: array [1..20] of integer;
- **tabllocns**: array [1..20] of integer;
function bot: boolean;
(* this func indicates that the beginning of the text has been reached when it is true.*)
begin
  bot := ch = botchar
end; (* bot *)

function boin: boolean;
(*this func indicates that the char (ch) read from the buffer or file is the beginning of line marker, i.e. im.*)
begin
  boin := ch = im
end; (* boin *)

function boo: boolean;
(*this func indicates that the char (ch) read from the buffer or file is the beginning of paragraph marker, i.e. om.*)
begin
  moo := ch = om
end; (* moo *)

function break: boolean;
(*this function indicates that the char (ch) read from the buffer or file is the break char.*)
begin
  break := ch = breakch
end; (* break *)

function eot: boolean;
(*this func is analogous to the eof and eoln funcs and indicates the end of the text has been reached.*)
begin
  eot := ch = eotchar
end; (* eot *)
function formatchar: boolean;
(* this func is true when ch is a format command or part of one.*)

begin
  (* formatchar *)
  if (ord(ch) >= 0) and (ord(ch) <= 31) then
    formatchar := true
  else if ch in ['1', '2', '3', '4', '5'] then begin
    (check for numeric part of format commands)
    ch := getbufchar(-1);
    formatchar := ch in [titlech, inspacingch];
    ch := getbufchar(+1)
  end;
  formatchar := false
end; (* formatchar *)

function textchar: boolean;
(* this func is true when ch is a valid displayable text char only. Line/para markers and format commands cause a false value to be returned.*)

begin
  (* textchar *)
  textchar := (ord(ch) >= 32) and (ord(ch) <= 126)
end; (* textchar *)
function getbufchar(inc: integer): char;

(*this func gets a char from the buffer. It is either the predecessor, same or successor to the last char returned.*)
(* inc has the values +1,0,-1.*)

const
  neg1 = -1;
  zero = 0;
  pos1 = +1;

begin
  case inc of
    neg1:
      begin
        buffer[insend] := buffer[bufindx];
        insend := insend - 1;
        bufindx := bufindx + 1;
        insbeg := insbeg+1;
      end;
    zero:
      null; (* no change within the buffer *)
    pos1:
      begin
        bufindx := bufindx + 1;
        insbeg := insbeg + 1;
        insend := insend + 1;
        buffer[bufindx] := buffer[insend];
      end
  end; (* case *)
  getbufchar := buffer[bufindx];
end; (* getbufchar *)
function lnlenoth: integer;
(*this func counts the chars, including
spaces w/in the current line of the buffer and
expects the buffer position to be at l1m or om.*)

var
cnt: integer;

begin (*lnlengt h*)
cnt := -1;
repeat
  ch := getbufchar(+1);
cnt := cnt + 1
until newline or eot;
lnlength := cnt;
{return to beginning of line}
repeat
  ch := getbufchar(-1);
cnt := cnt - 1
until cnt < 0
end; (*lnlength*)
function newline: boolean;
(*this func is true whenever an \texttt{ln}, \texttt{om},
or format cmd indicating a new line is
encountered.*)
begin
  (* newline *)
  newline := ch in (breakch, om, lm, titlech, ojectch)
end;
(* newline *)

function center: integer;
(*this func returns the starting position
for the centering of a line.*)
begin
  (* center *)
  center := centerofog - lnlength div 2
end;
(* center *)
procedure terminal;

(* this procedure sets the terminal constants, i.e.
ascii codes based on the user input terminal type.*)

var
  ans: char;

procedure adm3a;

(* this proc assigns the unique adm-3a terminal codes.*)

begin
  home := chr(30);
  clear := chr(26);
  up := chr(11);
  right := chr(12)
end;

(* adm3a *)

procedure dm1520;

(* this proc assigns the unique dm-1520 terminal codes.*)

begin
  home := chr(19);
  clear := chr(31);
  up := chr(28);  (* dm1520 *)
  right := chr(28)
end;

(* dm1520 *)

procedure dm2500;

(* this proc assigns the unique dm-2500 terminal codes.*)

begin
  home := chr(2);
  clear := chr(30);
  up := chr(26);
  right := chr(28)
end;

(* dm2500 *)
procedure termcodes;

(*this proc assigns the ascii terminal codes that are common to all the terminals used w/ this system.*)

begin

  lastcol := 80;
  lastln := 24;
  bell := chr(7);
  cr := chr(13);
  if := chr(10); (* down *)
  down := chr(32);
  (* note that if/down appears as ' ' on UNIX *)
  rubout := chr(127);

  (rubout from terminal doesn't work in raw mode)
  (escape := chr(27); doesn't work in raw mode)
  left := chr(8);
  accept := chr(1); (*ctrl a *)
  erase := chr(5); (*ctrl e *)
  quit := chr(17) (*ctrl a *)
end; (* termcodes *)

92
begin
  (* terminal *)

  write(chr(26), chr(12), chr(30));
  write(' enter the terminal type as follows: ');
  write(chr(13), chr(10));
  write('1 for adm-3a, 2 for dm1520, 3 for dm2500');
  write(chr(13), chr(10), ' > ');
  repeat
    read(ans)
  until (ans = '1') or (ans = '2') or (ans = '3');
  termcodes;
  case ans of
    '1':
      begin
        adm3a;
        termno := 1
      end;
    '2':
      begin
        dm1520;
        termno := 2
      end;
    '3':
      begin
        dm2500;
        termno := 3
      end
  end (* case *)
end; (* terminal *)
procedure setdefaults;
(*this proc sets the default values for
a standard 8 1/2" x 11" sheet of paper
e.g. margins, etc.*)

begin
(* setdefaults *)
    fill := true;
    bmargin := 9; (* 1 1/2" *)
    lmargin := 13; (* 1 1/4" *)
    ommargin := 6; (* 1" *)
    omarin := 17; (* 4 spaces in \frac{1}{4} from lmargin *)
    rmargin := 72; (* 1 1/4" *)
    tmargin := 6; (* 1" *)
    maxcols := 66;
    inspacing := 2;
    centerofeq := (rmargin - lmargin) div 2 + lmargin;
    tab := 4;
    n := 1;
    repeat
        tablochs[n] := lmargin + 1 + tab * n;
        n := n + 1
        until (n = 21) or (tab * n >= 80)
end; (* setdefaults *)

94
procedure setformatcmds;

(*this proc assigns the ascii codes for the format cmds and markers used w/in this system*)

begin
(* setformatcmds *)
  adjustch := chr(1); (* ctrl a *)
  breakch := chr(2); (* ctrl b *)
  centerinch := chr(3); (* ctrl c *)
  tmarginch := chr(4); (* ctrl d *)
  ojectch := chr(5); (* ctrl e *)
  fillch := chr(6); (* ctrl f *)
  omarginch := chr(7); (* ctrl g *)
  headerch := chr(8); (* ctrl h *)
  tabch := chr(9); (* ctrl i *)
  indentch := chr(11); (* ctrl k *)
  inspacingch := chr(12); (* ctrl l *)
  lm := chr(13); (* line marker = cr *)
  pnumch := chr(14); (* ctrl n *)
  filloffch := chr(15); (* ctrl o *)
  om := chr(16); (* para marker = ctrl p *)
  footerch := chr(17); (* ctrl q *)
  rjustifvch := chr(18); (* ctrl r *)
  skipinch := chr(19); (* ctrl s *)
  titlech := chr(20); (* ctrl t *)
  underinch := chr(21); (* ctrl u *)
  num := chr(22); (* new marker = ctrl v *)
  marinch := chr(23); (* ctrl w *)
  adjustoffch := chr(24); (* ctrl x *)
  botchar := chr(25); (* ctrl y *)
  eotchar := chr(26); (* ctrl z *)
  bm := chr(27); (* begin marker = ctrl [ *)
  bmarginch := chr(29); (* ctrl \ *)
  em := chr(29); (* end marker = ctrl ] *)
  rmarinch := chr(30); (* varies *)
  pmarginch := chr(31); (* varies *)
end; (* setformatcmds *)
DESIGN AND IMPLEMENTATION OF AN INTEGRATED SCREEN-ORIENTED TEXT--ETC

JUN 80  L A TALMAGE

END
DATE
TIME
OFTC

UNCLASSIFIED
procedure xyaddr(col, row: integer);

(*this proc initiates absolute x-y addressing on
the crt screen for adm-3a and dm-1520 terminals.
termnum is 1 for the adm-3a and 2 for the dm-1520.
col is the actual column number on the screen
from 1 to 80 where the display is to occur.
row is the actual row number on the screen
from 1 to 24 where the display is to occur.*)

begin (*xyaddr *)
    case termno of
        1: (*adm-3a*)
            write(chr(27),chr(61),chr(row+31),chr(col+31));
            (*31 is the offset for ea. value to set the
correct ascii code.*)
        2: (*dm-1520*)
            write(chr(30), chr(col + 31), chr(row + 31));
        3: begin (*dm-2500*)
            (*the offsets vary in setting the correct ascii code*)
            if col <= 32 then
                col := col + 95
            else if (col >= 33) and (col <= 64) then
                col := col + 63
            else if (col >= 65) and (col <= 80) then
                col := col + 31;
            write(chr(12), chr(col), chr(row+95))
        end
    end (*case *)
end; (*xyaddr *)
procedure suserinput(nochars: integer);
(*this proc gets nochars of user input. A
'crl' terminates short user input.*)

begin
  (*suserinput *)
i := 1;
  repeat
    read(inchar);
    if inchar = erase then begin
      write(left, ' ', left);
      i := i - 1
    end else begin
      if inchar <> cr then begin
        write(inchar);
        if i <= smaxchars then
          cmd[i] := inchar;
        i := i + 1
      end
    end
  until (inchar = cr) or (i = smaxchars + 1);
  if i <= nochars then
    for n := i to smaxchars do
      cmd[n] := ' ';
  else
    for n := nochars + 1 to smaxchars do
      cmd[n] := ' ';
end; (*suserinput *)
procedure luserinput(nochars: integer);(*this proc gets nochars of user input. A<cr>terminates long user input.*)begin
  i := 1;
  repeat
    read(inchar);
    if (inchar = erase) and (i > 1) then begin
      write(left, ' ', left);
      i := i - 1
    end else begin
      if inchar <> cr then begin
        write(inchar);
        if i <= lmaxchars then begin
          name[i] := inchar;
          i := i + 1
        end
      end
    end
  until (inchar = cr) or (i > lmaxchars + 1);
if namingfile then
(*check for .p in filename*)
if (name[i - 2] <> '.') or (name[i - 11] <> 'o') then begin
  if i <= nochars - 1 then begin
    name[i] := 'o';
    name[i + 1] := 'o';
    i := i + 2
  end else begin
    name[nochars - 1] := 'o';
    name[nochars] := 'p'
  end
end;
if i <= nochars then
  for n := i to lmaxchars do
    name[n] := ' ';
else
  for n := nochars + 1 to lmaxchars do
    name[n] := ' '(*luserinput*)
procedure convrtnum(i: integer; var num: integer);
(*this proc converts the char number in the string starting at index i into an integer value.*)

function unit(charno: char): integer;
(*this func returns the unit integer value of the char number.*)
begin
  unit := ord(charno) - ord('0')
end;

begin
  n := 0;
  repeat
    n := n + 1
    until cmd[i + n] = ' ';
  case n of
    1:
      num := unit(cmd[i]);
    2:
      num := unit(cmd[i]) * 10 + unit(cmd[i + 1]);
    3:
      num := unit(cmd[i]) * 100 + unit(cmd[i + 1]) * 10 + unit(cmd[i + 2])
  end
end; (*convrtnum*)
procedure blankin( rownum: integer );
(*this proc causes a line of blanks to be printed extending from the left screen boundary to the 75th col on the right so that the bell doesn't ring each time.*)
begin
  xyaddr(1, rownum);
  write(' ');
  write(' ');
  write(' ');
end;
(* blankin *)

procedure msclear;
(*this proc blanks out the first 3 lines of the crt screen, i.e. the msg area.*)
begin
  blankin(line1);
  blankin(line2);
  blankin(line3)
end;
(* msclear *)

procedure pause;
(*this proc makes the user aware of error msgs and prompts by requiring his interaction w/ the system.*)

var
  ans: char;
begin
  xyaddr(col24, line3);
  write(' <space> to continue ');
  repeat
    read(ans)
    until ans = ' ';
end;
(* pause *)

100
procedure prompt(promptnum: integer);
(*this proc writes out the prompt indicated by promptnum in the msg area on the crt screen.*)

begin
  msgclear;
  xyaddr(col12, line1);
  case promptnum of
    1:
      begin
        write('EDIT MODE q)uit A)ddfile X)tractfile ');
        xyaddr(col12, line2);
        write('C)ursormove');
        xyaddr(col12, line3);
        write('c)opy d)elete i)nsert l)ocate m)ove');
        xyaddr(col12, line4);
        write('r)emove s)etmarker x)change');
        xyaddr(colino, rowno)
      end;
    2:
      begin
        write('INSERT MODE');
        xyaddr(col10, line2);
        write('ctr1 a to accept; ctrl e to erase; ');
        write('ctr1 q to quit')
      end;
    3:
      begin
        write('FORMAT MODE')
      end;
    4:
      begin
        write('COMMAND MODE >');
        xyaddr(col18, line2);
        write('e)dit f)ormat l)isting q)uit ');
        write('r)eadfile w)ritefile');
        xyaddr(col18 + 22, line1)
      end;
    5:
      begin
        write('DELETE MODE');
        xyaddr(col18, line2);
        write('ctr1 a to accept; ctrl q to quit')
      end;
    6:
      begin
        write('COPY MODE')
      end;
  end;
end;
7:
begin
  write('MOVE MODE')
end;
8:
begin
  write('processing completed.')
  xyaddr(col18, line2);
  write('delete markers set in buffer--y or n ?')
  xyaddr(col18, line3);
  write('>')
end;
10:
begin
  write('buffer full !!!')
  xyaddr(col18, line2);
  write('cntrl a)accept or cntrl q)uit ?')
end;
11:
begin
  write('does file already exist--y or n ?')
  xyaddr(col18, line2);
  write('>')
end;
12:
begin
  write('enter initial page no. followed by <cr>')
  xyaddr(col18, line2);
  write('>')
end;
13:
begin
  write('print listing on UNIX printer--y or n ?')
  xyaddr(col12, line2);
  write('>')
end;
14:
write('at end of file')
15:
write('listing being produced')
16:
begin
  write('SETMARKER MODE --eqin E)nd N)ewlocn')
  xyaddr(col12 + 2, line2);
  write('move cursor then cntrl s to setmarker ')
  write('cntrl q to quit')
  xyaddr(col18, line3);
  write('text between/including B) and E) chars ')
  write('is placed before the cursor.')
end;
begin
  write('LOCATE MODE')
  end;
18:
begin
  write('REPLACE MODE')
  end;
19:
write('at beginning of file');
20:
write('XCHANGE MODE');
21:
write('Addfile');
22:
write('Xtractfile');
23:
begin
  write('Cursor Movement arrows');
  xyaddr(col8, line2);
  write('<cr> U)scroll up D)scroll down page');
  xyaddr(col8, line3);
  write('b)eginning of file e)nd of file')
end
end (* case *)
end (* prompt *)
procedure errormsg(msgnum: integer);

(*this proc prints out the error msg indicated by msgnum in the msg area on the crt screen.*)

begin (*errormsg*)
  write(bell);
  msqclear;
  xyaddr(col12, line1);
  case msgnum of
    1: write('invalid command');
    2: begin
       write('moving the cursor is meaningless w/ '); 
       write('no text present in the buffer.');
    end;
    3: write('readfile before entering EDIT MODE');
    4: write('invalid EDIT command');
    5: write('readfile before entering FORMAT MODE');
    6: write('invalid INSERT command');
    7: write('all inserted text has been erased');
    8: write('invalid DELETE command');
    9: begin
       write('deletion direction must be consistent, i.e.'); 
       xyaddr(col18, line2); 
       write('go right and down or go left and up.');
    end;
   10: write('invalid SETMARKER command');
   11: write('begin marker already set');
   12: write('end marker already set');
   13: write('new locn marker already set');
end;
write('begin marker must occur before end marker');
write('new locn cannot be located in the delimited text');
write('begin marker has not been set');
write('end marker has not been set');
write('new locn marker has not been set');
write('insufficient buffer space for copy to occur');
write('please indicate the title type--1, 2, 3, 4');
write('not implemented');
write('readfile before entering SETMARKER mode');
write('buffer full!!!!!!!')
end (* case *)
pause
end (* errmsg *)
procedure printer;
(* this proc makes necessary adjustments to margins for the UNIX printer.*)
begin
  (* printer *)
  repeat
    prompt(13);
    suserinput(1)
    until (cmd[1] = 'y') or (cmd[1] = 'n');
  if cmd[1] = 'y' then begin
    (* UNIX printer's driver allows the user to write only 63 lines to a og vice 66.*)
    tlnoffset := 2; (* 2 ln at the top *)
    binoffset := 2 (* 2 ins at the bottom *)
  end else begin
    tlnoffset := 0;
    binoffset := 0
  end
end; (* printer *)
procedure getnumericmd;
(*this proc gets the numeric part of a format command and converts it to an integer.*)
begin
(* getnumericmd *)
i := 0;
repeat
    i := i + 1;
    ch := qetbufchar(+1);
    cmd[i] := ch
until break;
    cmd[i] := ' ';
    convrtnum(i - 1, number)
end;

procedure jptomarker(markerchar: char; inc: integer);
(*this proc moves backward or forward thru the buffer until the marker, e.g. bot is reached. inc indicates the direction of the move, i.e. +1=>forward and -1=>backward.*)
begin
(* jptomarker *)
    ch := qetbufchar(0);
    if ch <> markerchar then
        repeat
            ch := qetbufchar(inc)
            until ch = markerchar
    end;

(* jptomarker *)
procedure filename(calledby: char);

(*this proc requests user input of the name of
the file to be processed. Up to 14 chars, including
the '.p' for a PASCAL file, consisting of alpha and
numeric chars and the period are allowed.*)

begin
  msqclear;
  xyaddr(col12, line1);

  case calledby of
    'A':
      write('name the file to be added to the buffer,');
      'I':
      write('name the listing file,');
      'R':
      write('name the rawfile to be read into the buffer,');
      'W':
      write('name the raw file saving the buffer contents,');
      'X':
      write('name the file to be extracted from the buffer,')
  end; (*case
  xyaddr(col18, line2);
  write('enter the filename.p, up to 14 chars, ');
  write('followed by <cr>');
  xyaddr(col18, line3);
  write('>' );
  namingfile := true;
  luserinput(maxchars);
  namingfile := false;
  write('the filename is ', name);
  pause
end; (* filename *)
function startln(nolines: integer): char;
(*this func finds the beginning of the line to be
displayed by either backtracking or skipping forward
through the buffer by counting the number of line and
paragraph markers encountered and comparing this total
w/the specified nolines. startln returns an im or pm.
nolines values indicate the following:
- \text{n} backward \text{n line(s)},
0 same line,
+\text{l} forward \text{n line(s).} *)
begin
(* startln *)
if nolines <= 0 then begin (* go backwards *)
(* to find start of current line *)
ch := getbufchar(0);
if not newline and not bot then repeat
ch := getbufchar(-1)
until newline or bot
end;
if (nolines < 0) and not bot then repeat
repeat
ch := getbufchar(-1)
until newline or bot;
nolines := nolines + 1
until (nolines = 0) or bot
else if (nolines > 0) and not eot then (* go forward *)
repeat
repeat
ch := getbufchar(+1)
until newline or eot;
nolines := nolines - 1
until (nolines = 0) or eot;
if bot then begin repeat
ch := getbufchar(+1)
until newline
end else if eot then repeat
ch := getbufchar(-1)
until newline;
startln := ch
end; (* startln *)
procedure setcursor(nocol, nolines: integer);

(* this proc maps the cursor col and row to the correct position in the buffer. Nocol is the col locn of the text char in the line of text. See startln above for nolines description. *)

begin (* setcursor *)
  ch := startln(nolines);
  if nocol = lmargin then begin
    if bolln then
      colno := lmargin + 1
    else if bollp then
      colno := pmargin + 1
    else if ch = tabch then
      process tab;
    (* not to 1st char in line in the buffer *)
    repeat
      ch := getbufchar(+1)
    until textchar or eot
  end else begin
    if bolln then
      i := lmargin
    else if bollp then
      i := pmargin
    else if ch = tabch then
      i := lmargin + tab;
    repeat
      ch := getbufchar(+1);
      if textchar then
        i := i + 1
    until (i = nocol) or newline or eot;
    (* if there are not enough chars in the line then pt to the last one. *)
    if newline or eot then
      repeat
        ch := getbufchar(-1)
      until textchar;
    colno := i
  end
end;

(* setcursor *)
procedure adjustln(lnctr: integer);
(*this proc inserts extra blanks in a ln of words
so that the last char of the last word is on the
right margin. Each ln is divided into 2 parts at
midword. lbblkcnt and rbblkcnt are the # of blanks
inserted to the left and right of midword. endblkcnt
is the total # of blanks to be inserted in the ln.
blklocns indicates where w/in the ln blanks are located.
inc is the offset in blklocns due to previous blank
insertion. lend is the blank position preceding midword.*)

var
   endblkcnt, i, inc, lbblkcnt, rbblkcnt: integer;
   midword, wordcnt, lnsize, lend: integer;
   blklocns: array (1..201 of integer);

procedure cntwords;
(*this proc counts the # of words in a ln of text
and finds out where blanks may be inserted.*)
begin
   (* cntwords *)
   ch := startln(-1);
   wordcnt := 0;
   i := 1;
   if lnsize > centerofag + 3 then begin
      while i <= lnsize do begin
         ch := getbufchar(+1);
         if ch = ' ' then begin
            (check for 2 spaces together)
            if (wordcnt>=1) and (blklocns[wordcnt]<i-1) then begin
               wordcnt := wordcnt + 1;
               blklocns[wordcnt] := i;
            end
         end else if wordcnt < 1 then begin
               wordcnt := 1;
               blklocns[1] := i;
            end
         i := i + 1
      end;
   end;
   wordcnt := wordcnt + 1
end (* cntwords *)
procedure leftside;
(*this proc adjusts the left half of the line
by inserting extra blanks to the left of midword*)

begin
(* leftside *)

  i := 1;
  inc := 0;
  lend := blkcnt * (midword - 1);
  ch := startln(0);
  j := 1;
  repeat
    ch := getbufchar(+1);
    write(outfile, ch);
    if (inc < lend) and (blkcnt > 0) then begin
      if j = blklocns[i] then begin
        for n := 1 to blkcnt do begin
          j := j + 1;
          write(outfile, ' ')
        end;
        i := i + 1;
        inc := inc + blkcnt;
        blklocns[i] := blklocns[i] + inc
      end
    end;
    j := j + 1;
  until inc >= lend
end;
(* leftside *)
procedure rtside;
(*this proc adjusts the right half of the line
by inserting extra blanks to the right of midword*)

begin (* rtside *)

i := midword;
repeat
ch := getbufchar(+1);
write(outfile, ch);
if (j < wordcnt) and (rb1kcnt > 0) then begin
  if j = blkolcns[i] then begin
    for n := 1 to rb1kcnt do begin
      j := j + 1;
      write(outfile, ' ')
    end;
    i := i + 1;
    inc := inc + rb1kcnt;
    blkolcns[i] := blkolcns[i] + inc
  end
  j := j + 1
until j > lnsize
end;

(* adjustln *)

begin (* adjustln *)

lnsize := lnlength;
endblkcnt := lmargin - (lnsize + lmargin);
if endblkcnt > 0 then begin
ctwords;
  if wordcnt > 0 then begin
    if incnt mod 2 = 0 then begin
      l1blkcnt := endblkcnt div (wordcnt - 1);
      r1blkcnt := l1blkcnt + 1;
      midword := wordcnt + (l1blkcnt*(wordcnt - 1) - endblkcnt)
    end else begin
      r1blkcnt := endblkcnt div (wordcnt - 1);
      l1blkcnt := r1blkcnt + 1;
      midword := endblkcnt + 1 - r1blkcnt * (wordcnt - 1)
    end;
    leftside;
  rtside
end
end
end
(* adjustln *)

113
procedure displayln;

(*this proc causes one line of up to 80 chars to be displayed on the crt screen at the line/row position indicated by rowno.*)

begin (*displayln *)
  if boIn then
    colno := lmargin + 1
  else
    colno := omargin + 1;
  xvaddr(colno, rowno);
  repeat
    ch := getbufchar(+1);
    if textchar then begin
      write(ch)
    end
  until newline or ent
end; (*displayln *)
procedure displayscreen(col, nolines, row: integer);
(* this proc causes up to (lastln - msglns), usually (24-3) or 21, lines to be displayed on the crt screen. nolines is described in startln. col and row indicate the final cursor position.*)
var
  bufInCnt, incnt: integer;
begin
  (* displayscreen *)
  inspacing := 1;
  write(clear);
  bufInCnt := 0;
  incnt := 0;
  rowno := 1 + msglns;
  ch := startln(nolines);
  repeat
    displayln;
    bufInCnt := bufInCnt + 1;
    incnt := incnt + 1;
    rowno := rowno + 1;
  case inspacing of
    1: null;
    2:
      if rowno < lastln then begin
        incnt := incnt + 1;
        rowno := rowno + 1;
      end;
    3:
      begin
        incnt := incnt + 2;
        rowno := rowno + 2
      end
  end (* case *)
  until (incnt >= lastln - msglns) or eot;
  if eot then
    bufInCnt := bufInCnt - 1;
  if row = 1 then
    setcursor(col, -bufInCnt);
  else if rowno = lastln then
    setcursor(col, 0)
  else
    setcursor(col, -bufInCnt - row);
  if inspacing = 1 then
    rowno := row + msglns;
  else
    rowno := row + inspacing - inspacing + 1 + msglns;
  xyaddr(colno, rowno)
end; (* displayscreen *)
procedure scrollup;

(*this proc causes the user window to move
backwards thru the text far enough to
move 16 lines, 3/4's of a screen, or
until bot is reached. It provides for an
overlap, i.e. the top 5 lines will become
the bottom 5 lines, for continuity.*)

var
  uplns: integer;

begin
  uplns := overlap - lastln + msqlns - rowno;
  displayscreen(lmargin, uplns, lastln-msqlns-overlap-1)
end; (*scrollup *)

procedure scrolldown;

(*this proc causes the user window to move
forward thru the text buffer far enough to
move 16 lines, 3/4's of a screen, or until
eot is reached. It provides for an overlap,
i.e. the last 5 lines will become the top 5.*)

var
  downlns: integer;

begin
  downlns := lastln - msqlns - overlap + rowno;
  displayscreen(lmargin, downlns, overlap - 1)
end; (*scrolldown *)
procedure chqrowno(inc: integer);

(*this proc causes the row number incremented to be or decremented by 1. It uses the terminal's HW scrolling feature when at the bottom in on the CRT.*)

begin
  (* chqrowno *)
  if (inc = -1) and (rowno = 1) then
    scrollup
  else
    case incspacing of
      1:
        if rowno < lastIn then
          rowno := rowno + inc
        else begin (rowno=lastIn)
          write(1f);
          displayln
        end;
      2:
        if rowno < lastIn then
          rowno := rowno + 2 * inc
        else begin (rowno=lastIn)
          write(1f, 1f);
          displayln
        end;
      3:
        if rowno < lastIn - 2 then
          rowno := rowno + 3 * inc
        else begin (rowno=lastIn-2)
          write(1f, 1f, 1f, 1f);
          displayln
        end
    end (* case *)
  end; (* chqrowno *)
procedure mapcursor(inc: integer);
(*this proc duplicates the cursor movement
on the screen as movement thru the buffer
so that there is a one-to-one correspondence
between the cursor char and the buffer char.*)
(*inc has the values +1 and -1.*)

begin
  if bop or boll then begin
    case inc of
      neg1:
        begin (* wraparound to end of previous ln*)
          ch := startin(-1);
          if boll then
            colno := lmargin + llength + 1
          else
            colno := omargin + llength + 1
        end;
      pos1: (* go forward to next line *)
        if boll then
          colno := lmargin
        else
          colno := omargin
      end; (* case *)
    charowno(inc)
  end else if bot then  (* other format commands *)
    oromot(19)
  else if eot then
    oromot(14)
  else
    null
end; (* mapcursor *)
function getchar(inc: integer): char;

(*this func returns only displayable text
chars from the buffer. It ignores the format
commands embedded in the buffer.*)
(* inc has the values +1 and -1 *)

const
    neg1 = -1;
    pos1 = +1;

begin
    (* getchar *)
    case inc of
        neg1:
            repeat
                ch := getbufchar(-1);
                if formatchar then
                    mapcursor(-1)
                else
                    colno := colno - 1
                until textchar or bot;
        pos1:
            repeat
                ch := getbufchar(+1);
                if formatchar then
                    mapcursor(+1)
                else
                    colno := colno + 1
                until textchar or eot
            end; (* case *)
        if bot or eot then
            repeat
                ch := getbufchar(-inc)
            until textchar;
        end;
    end;

end;

(* getchar *)
procedure titleomfix(var pmchars: integer);
(*this proc changes the paragraph margins to conform to thesis manual specifications for indentation.*)

begin
  case ch of
  '1': pmargin := 17;
  '2': pmargin := 17;
  '3': pmargin := 21;
  '4': pmargin := 25;
  '5': pmargin := 17
  end; (* case *)
  pmchars := rmargin - pmargin + 1
end; (* titleomfix *)
procedure processbuf(start, finish: integer);

(*this proc processes the buffer after text insertion, deletion or format changes, e.g. rmargin or lmargin. Only the paragraph(s) involved in text changes will be reprocessed.*)

var
  lchars, pmchars, nochars, cntr: integer;
  halt: boolean;

procedure onlyread(stop: char);
(*this proc prevents processing of specially formatted text such as titles, etc.*)

begin (*onlyread *)
  repeat
    ch := getbufchar(+1)
    until (ch = stop) or eot;
  halt := true
end; (*onlyread *)
begin
  (* processbuf *)
  lmchars := rmargin - lmargin + 1;
  pmchars := rmargin - pmargin + 1;
  halt := false;
  ch := startln(0);
  if bufindx > start then begin
    repeat
      ch := startln(-1)
      until boa and (bufindx <= start) or bot;
    end;
    if bot then
      repeat
        ch := qetbufchar(+1)
        until newline
      end;
    if boa then
      nochars := pmchars
    else
      nochars := lmchars;
    repeat
      cntr := 0;
      repeat
        ch := qetbufchar(+1);
        if textchar or boa then begin
          cntr := cntr + 1;
          if boa then begin
            ch := ' ';
            buffer[bufindx] := ch
          end;
        end else if ch = filloffch then
          onlyread(fillch)
        else if ch = titlech then begin
          ch := qetbufchar(+1);
          titlepmfix(pmchars);
          onlyread(pm)
        end else if ch = tabch then
          nochars := nochars - tab
        until newline or (cntr = nochars) or boa or eot;
    end
  end
  if not (eot or boa) then begin
    (* backtrack to space before word *)
    repeat
      ch := qetbufchar(-1)
      until ch = ' ';
      buffer[bufindx] := lm;
      nochars := lmchars
    end else if boa then
      nochars := pmchars
    else if halt then
      halt := false
  end
  until boa and (bufindx >= finish) or eot
end; (* processbuf *)
procedure movecursor;

(*this proc causes the cursor to move on the screen as well as mapping the cursor movement to the movement thru the buffer.*)

begin

(* movecursor *)

if inchar = right then
  (* move forward 1 col *)
  ch := getchar(+1)
else if inchar = left then
  (* move backward 1 col *)
  ch := getchar(-1)
else if inchar = up then begin (* move up 1 line *)
  setcursor(colno, -1);
  charowno(-1)
end
else if inchar = down then begin (* move down 1 line *)
  setcursor(colno, +1);
  charowno(+1)
end
else if inchar = 'U' then (* backward 3/4's screen *)
  scrollup
else if inchar = 'D' then (* forward 3/4's screen *)
  scrolldown
else if inchar = 'b' then begin
  (* go to the beginning of file *)
  imotomarker(botchar, -1);
  displayscreen(lmargin, 0, +1)
end
else if inchar = 'e' then begin
  (* go to the end of file *)
  imotomarker(eotchar, +1);
  displayscreen(rmargin, -(2 * overlap) + 1, 2 * overlap)
end
else if inchar = cr then begin
  (* move to 1st col of next line down *)
  setcursor(lmargin, +1);
  charowno(+1)
end
else if inchar = 'o' then
  null;
xyaddr(colR, lastln);
write(ch);
xyaddr(colno, rowno)
end;

(* movecursor *)
procedure readraw;

(*this proc reads the raw file, if present, into the buffer and reorganizes the buffer into the correct structure. if no file is present the buffer is set up for the creation of a new file.*)

procedure newbufinit;
(*this proc initializes the buffer for the creation of a new file.*)

begin

(* newbufinit *)
(*botchar:formatrec:om: insert area :eotchar:empty*)
(*begindx: ... :insbeg: :insend: :buflimit*)

begindx := 1;
buffer[begindx] := botchar;
buff[begindx + 11 := pm;
bufindx := begindx + 1;
insbeg := begindx + 2;
insend := buflimit + 2;
buffer[buflimit + 1] := eotchar
end;

(* newbufinit *)
procedure processraw;
(*this proc processes the raw file by counting off the
no. of chars to be in a line after the margin spaces
are considered and includes the proper spacing consid-
erations. A processed line is preceded by a lm or pm
if it is a paragraph. Previously set lm's are blanked*)
var
  cntr, lmchars, nochars, pmchars: integer;
begin (* processraw *)
  pmchars := rmargin - omargin + 1;
  lmchars := rmargin - lmargin + 1;
  nochars := lmchars;
  bufindx := 0;
  j := 0;
  while not eof(infile) do begin
    cntr := j;
    repeat
      read(infile, ch);
      bufindx := bufindx + 1;
      if fill then
        if textchar or boln then begin
          cntr := cntr + 1;
          if boln then
            ch := ' ';
        end else if boln then begin
          nochars := pmchars;
          j := 0
        end else if ch = tabch then
          nochars := nochars - tab
        else if ch = titlech then begin
          buffer[bufindx] := ch;
          bufindx := bufindx + 1;
          read(infile, ch);
          titlepmfix(pmchars)
        end;
        buffer[bufindx] := ch
      until newline or (cntr = nochars) or eof(infile);
    if not (newline or eof(infile)) then begin
      if ch = ' ' then begin
        buffer[bufindx] := lm;
        j := 0
      end else begin (backtrk to ' ' before word)
        j := 0;
        repeat
          j := j + 1
        until buffer[bufindx - j] = ' ';
        buffer[bufindx - j] := lm
      end;
      nochars := lmchars
    end
  end
end; (* processraw *)
procedure oldbufinit;
(*this proc initializes the buffer after an existing file has been read in.*)

var
bufend: integer;

begin (*oldbufinit *)
(*botchar:formatreciom:insert area:textarea:botchar:empty*)
prompt(14);
processraw;
bufend := bufindex;
j := buflimit - 1;
i := bufend;
repeat
  buffer[j] := buffer[i];
  buffer[i] := ' ';
  i := i - 1;
  j := j - 1
until i = 2;
insend := j;
begindx := i - 1;
insbeg := begindx + 1
end;
(* oldbufinit *)

begin (* readraw *)
setdefaults;
write(clear); {for previous reads)
fileread := true;
repeat
  prompt(11);
suserinput(1);
  ch := sstring[1]
until (ch = 'y') or (ch = 'n');
if ch = 'n' then begin {empty new file}
  emptybuf := true;
  newbufinit
  end else begin {nonempty existing file *}
  emptybuf := false;
  filename('r');
  reset(infile, name);
  oldbufinit;
  bufindex := begindx + 1;
displayscreen(1margin, 0, 1)
  end
end;
(* readraw *)
procedure writeraw;
(*this proc writes the buffer back to storage.*)
begin
    (* writeraw *)
    filename('w');
    rewrite(outfile, name);
    jmotomarker(botchar, -1);
    ch := getbufchar(0);
    write(outfile, ch);
    if not eot then
        repeat
            ch := getbufchar(+1);
            if not (ch in (rubout, bm, em, nm)) then
                write(outfile, ch)
            until eot;
        flush(outfile)
end;
procedure listorocessed;

(this proc processes the buffer and writes out the processed text to the user-indicated file for printing w/ no copy saved.)

var
  lncnt: integer;

procedure blankline(nolines: integer);

(* this proc writes out a blankline of processed text to the output file.*)

begin
  (* blankline *)
  for I := 1 to nolines do
    (* blankline *)
    write(outfile, cr, lf)
  end;

procedure endpage;

(*this proc performs the end of page processing necessary for page numbering, proper margins, etc.*)

begin
  (* endpage *)
  repeat
    (* endpage *)
    blankline(1);
    lncnt := lncnt + 1
  until lncnt >= maxpgins - omargin;
  if pageno <> 0 then begin
    (* endpage *)
    write(outfile, pageno: centerofpq);
    (* endpage *)
    lncnt := lncnt + 1;
    (* endpage *)
    pageno := pageno + 1
  end;
  (* endpage *)
  blankline(maxpgins - lncnt - blineno)
end;
procedure writetitle(indentn: integer);
(*this proc writes out one line, a title*)

begin (* writetitle *)
  for i := 1 to indentn do
    write(outfile, ' ');
  repeat
    ch := getbufchar(+1);
    if not formatchar then
      write(outfile, ch)
    until newline;
  write(outfile, cr, lf)
end; (* writetitle *)
procedure processesTitle;
classification as 1st order, 2nd order, etc.,
according to thesis manual instructions;(*)
begin
(* processesTitle *)
    ch := getbufchar(+1);
    case ch of
    '1':
        begin
            blankline(tmargin + 3);
            omargin := 17;
            writetitle(center);
            blankline(2);
            Incnt := tmargin + 6
        end;
    '2':
        begin
            if (Incnt = tmargin) or (Incnt = tmargin + 6) then
                Incnt := Incnt - 1
            else
                blankline(1);
                omargin := 17;
                writetitle(1margin);
                blankline(1);
                Incnt := Incnt + 3
        end;
    '3':
        begin
            omargin := 21;
            writetitle(1margin + 4);
            blankline(1);
            Incnt := Incnt + 2
        end;
    '4':
        begin
            omargin := 25;
            writetitle(1margin + 8);
            blankline(1);
            Incnt := Incnt + 2
        end;
    '5':
        begin
            endpage;
            blankline(tmargin + 3);
            omargin := 17;
            writetitle(center);
            blankline(2);
            Incnt := tmargin + 6
        end
    end (case)
end; (* processesTitle *)
procedure listln;

(*this proc writes one line of processed text to the output file.*)

begin (* listln *)
  if ch = titlech then begin
    processtitle;
    atitle := true
  end else begin
    if boln then
      for i := 1 to lmargin do
        write(outfile, ' ')
    else if boh then
      for i := 1 to pmargin do
        write(outfile, ' ')
    else if ch = breakch then
      for i := 1 to lmargin do
        write(outfile, ' ')
    else if ch = pagech then
      endpage;
  repeat
    ch := getbufchar(+1);
    if not newline then
      if textchar then
        write(outfile, ch)
    until newline or eot;
    write(outfile, cr, lf)
  end
end; (* listln *)
begin
  atitle := false;
  inspacing := 2;
  alisting := true;
  filename;
  rewrite(outfile, lstring);
  jmotomarker(botchar, -1);
  repeat
    ch := getbufchar(+1)
    until newline;
  promot(12);
  suserinput(3);
  convrtunm(1, pageno);
  centerofog := (rmargin - lmargin) div 2 + lmargin;
  promot(15);
  tmargin := tmargin - tlnoffset;
  bmargin := bmargin + blnoffset;
  pmargin := pmargin + blnoffset;
  repeat
    blankline(tmargin);
    lncnt := tmargin;
    repeat
      listin;
      if not atitle then
        case inspacing of
        1:
          lncnt := lncnt + 1;
        2:
          begin
            blankline(1);
            lncnt := lncnt + 2
          end;
        3:
          begin
            blankline(2);
            lncnt := lncnt + 3
          end
        end (* case *)
      else
        atitle := false
        until eot or (lncnt > maxpoints - bmargin);
        endpage
        until eot;
        pause;
        alisting := false
  end;

(* listprocessed *)

132
markersok := true;

begin

  case calledby of
    'A':
      {Addfile}
      mtest5;
    'c':
      {copy}
      if begmset and endmset and newmset then begin
        mtest1;
        mtest2
      end else begin
        mtest3;
        mtest4;
        mtest5
      end;
    'm':
      {move}
      if begmset and endmset and newmset then begin
        mtest1;
        mtest2
      end else begin
        mtest3;
        mtest4;
        mtest5
      end;
    'r':
      {replace}
      if begmset and endmset then
        mtest1
      else begin
        mtest3;
        mtest4
      end;
    's':
      {search}
      if begmset and endmset then
        mtest1
      else begin
        mtest3;
        mtest4
      end;
    'x':
      {Extractfile}
      if begmset and endmset then
        mtest1
      else begin
        mtest3;
        mtest4
      end
    end

end {case}

end; (* checkmarkers *)
procedure format;
begin
  prompt(3)
end;

(* format *)

(* format *)
procedure edit;
var
exit: boolean;

procedure copy;
(*this proc allows the user to copy/duplicate
a body of text w/in the buffer as long as the
entire copy will fit into the buffer. The
material to be copied is delimited by 2 cursor
settings. The new locn is also set w/ the cursor.
Note that the copy will overread the cursor locn.*)

var
inc, textlength, start, finish: integer;

procedure copychar;
(*this proc actually performs the char
transfer in the copying*)
begin
(* copychar *)
buffer[insend] := em;
buffer[endindx] := em;
repeat
  insend := insend - 1;
  endindx := endindx - 1;
  buffer[insend] := buffer[endindx]
until endindx = beginx + 1;
buffer[insend - 1] := bm;
buffer[beginx] := bm;
insend := insend - 2
end;
(* copychar *)
begin  
checkmarkers('c');
if markersok then
  if insend - insbeg > endindx - begindx then begin  
textlength := endindx - begindx;
  if bufindx < newindx then
    inc := +1
  else if bufindx > newindx then
    inc := -1
  else
    inc := 0;
repeat
  ch := getbufchar(inc)
until ch = nm;
repeat
  ch := getbufchar(-1)
until not formatchar;
if endindx < insbeg then begin
{...bm!...:lem!...:insend!nm!...}
st...  
  start := begindx;
  finish := insend + 1 - textlength;
  copychar
end else begin
{...insend!nm!...:bm!...:lem!...}
bufindx := insend + 2;
repeat
  ch := buffer(bufindx);
  bufindx := bufindx + (+1)
until ch = em;
end
endelse
end
else
errormsg(19)
end;
bufindx := insbeg - 1;
motomarker(nm, +1);
buffer(bufindx) := nm;
processbuf(start, finish)
end else
errormsg(19)
end;

(* copy *)
procedure delete;
(*this proc allows the user to delete chars in either the right or left direction by moving the cursor in the desired direction. An entire line of chars is deleted if the cursor is moved up or down. Deletion in the buffer only occurs upon user acceptance. Note that format cmd's w/in the deleted area are also lost.*)

var
  forwrd, backward, finished: boolean;
  stop: integer;

procedure delscreen(dir: char);
(*this proc writes blanks or blank lines to the screen to show how the deletion will appear. Note no deletion occurs in the buffer.*)

begin
(* delscreen *)
{deletion only reflected on screen}
if (dir = right) and not backward then begin
  forwrd := true;
  ch := getchar(+1);
  write(' ');
  ch := getchar(+1);
end else if (dir = left) and not forwrd then begin
  backward := true;
  write(left, ' ', left);
  ch := getchar(-1);
end else if (dir = up) and not forwrd then begin
  backward := true;
  blankln(rowno);
  setcursor(lmargin, -1);
  chrowno(-1); if rowno = 1 then begin scrollup;
    for i := 1 to overlap - 1 do blankln(rowno + i)
  end
end else if (dir = ' ') and not backward then begin
  forwrd := true;
  blankln(rowno);
  setcursor(lmargin, +1);
  chrowno(+1);
end else
  errmsg(9);
  xyaddr(colno, rowno)
end;
(* delscreen *)
procedure delprocess;
(*this proc causes the deleted text to be
removed from the buffer and processes the
remaining text as required.*)

begin
(* delprocess *)
  delend := bufindx;
  if forwrd then begin
    {... delete area! insert area !...}
    {...delbeg!...delend!insbeg!...insend!...}
    bufindx := delbeg;
    repeat
      ch := getbufchar(-1);
      if formatchar then
        delbeg := delbeg - 1
      until textchar;
    insbeg := delbeg;
    processbuf(delbeg, delend)
  end else if backward then begin
    {...delchar! insert area !delete area !...}
    {...delend!insbeg!...insend!......delbeg!...}
    bufindx := insbeg - 1;
    repeat
      ch := getbufchar(-1);
      if formatchar then
        delend := delend - 1
      until textchar;
    insend := delbeg;
    insbeg := delend;
    processbuf(delend, delbeg)
  end
end
(* delprocess *)
begin

(* delete *)

xyaddr(colno, rowno);
finished := false;
forward := false; (also downward)
backward := false; (also upward)
delbeg := bufindx;
repeat
  read(inchar);
  if inchar in (right, left, up, ' ') then
delscreen(inchar)
  else if ch = quit then  (no deletion occurs)
    finished := true
  else if inchar = accept then begin
    {deletion w/in buffer actually occurs}
    delprocess;
    finished := true
  end else begin
    errmsg(8);
prompt(5);
    xyaddr(colno, rowno)
  end
until finished;
if forward then
  stop := delbeg (backward)
else
  stop := delend;
repeat
  ch := getbufchar(-1)
until (bufindx <= stop) and boo or bot;
if bot then begin
  repeat
    ch := getbufchar(+1)
  until bot or boo or eot;
  if eot then
    emptybuf := true
  end;
if emptybuf then
  write(clear)
else
  display(screen(lmargin, -overlap, overlap + 1))
end; (* delete *)
procedure exchange;

(*this proc allows the user to change both format commands and text characters directly in the buffer. It is primarily used for trouble-shooting.*)

begin

(* exchange *)

  repeat
  read(inchar);
  if inchar <> quit then begin
    buffer[bfindx] := inchar;
    ch := getbufchar(+1);
    xyaddr(4, lastln);
    if textchar then
      write(ch)
    else if hop then
      write('%')
    else
      write('\')
  end
  until inchar = quit;
  displayscreen(1margin, 0, +1)
end;

(* exchange *)
procedure insert;
(* this proc causes text to be added to the buffer before the char indicated by the cursor. *)

var
  i, j, inspos, insquit: integer;
  finished: boolean;

procedure inserterase;
(*this proc moves the cursor back to the preceding char and blanks out the last entered char. wrap-
around occurs, in addition the cursor movement is duplicated in the buffer.*)

begin (* inserterase *)
  if bufindx > inspos then begin
    ch := buffer[bufindx];
    (wraparound from new empty line)
    if (ch = fillch) or (ch = filloffch) then begin
      fill := not fill;
      bufindx := bufindx - 1;
      insbeg := insbeg - 1
    end else if not newline then begin
      bufindx := bufindx - 1;
      insbeg := insbeg - 1;
      ch := buffer[bufindx];
      write(left, ' ', left)
    end;
  if newline then begin
    (wraparound to end of previous line)
  i := 1;
  repeat
    ch := buffer[bufindx - i];
    i := i + 1
  until newline;
  if boln then
    colno := i - 1 + lmargin
  else if boln then
    colno := i - 1 + omargin
  else if ch = tabch then
    colno := i - 1 + lmargin + tab;
  rowno := rowno - 1;
  bufindx := bufindx - 1;
  xyaddr(colno, rowno)
end else
  (same line erase)
  colno := colno - 1
end else begin
  errormsg(7);
  prompt(2);
  xyaddr(colno, rowno)
end
end; (* inserterase *)
procedure endline(nowrapprecedented: integer);
 (*this proc removes extra blanks from the end of
 the previous line, sets the lm or om, prepares
 the crt screen to display the next inserted char
 and rotates chars to fill the stripped slots.
 nowrap equal is the # of chars to be written to the
 new line. *)

var
 lastch: char;

begin (* endline *)
 (* remove extra blanks at eoln if present *)
 j := 0;
 repeat
   lastch := buffer[buflindx := j];
   j := j + 1
 until lastch = ' ';
 insbeg := insbeg - j + 2;
 bufindx := bufindx - j + 2;
 (* set marker *)
 if bop then begin
   colno := pmargin + 1;
   buffer[buflindx] := om
 end else begin
   colno := lmargin + 1;
   buffer[buflindx] := lm
 end;
 (* prepare crt screen *)
 if rowno < lastln then begin
   rowno := rowno + 1;
   blankin(rowno)
 end else
   write(11);
 endaddr(colno, rowno);
 (* rotate and write wrapped chars *)
 if nowrapped > 0 then
   repeat
     bufindx := bufindx + 1;
     if j >= 2 then
       buffer[buflindx] := buffer[buflindx + 1];
     write(buffer[buflindx]);
     nowrapped := nowrapped - 1
   until nowrapped = 1

end; (* endline *)
procedure putchar;
(*this proc adds another char to the buffer.*)
begin  (* putchar *)
  bufindx := bufindx + 1;
  buffer[bufindx] := ch;
  insbeg := insbeg + 1
end;  (* putchar *)

procedure insertchar;
(*this proc adds a new char to the text buffer and provides automatic line wraparound. note that the current bufindx contains a char while insbeg is empty.*)
begin  (* insertchar *)
  if fill then
    if colno <= rmargin then
      if ch = tabch then
        colno := colno + tab
      else begin
        write(ch);
        colno := colno + 1;
        putchar
      end
    else if (colno > rmargin) and (ch = ' ') then
      endline(0)
    else begin {nonblank char at eoln}
      if (colno>rmargin) and (ch<> ' ')
      putchar;
      i := 1;  (* # of wrapped chars *)
      (backtrack to space before word)
      repeat
        ch := buffer[bufindx - i];
        i := i + 1;
        until ch = ' ';
      (blank out wrapped chars on screen)
      colno := colno + 1 - i;
      xyaddr(colno, rowno);
      for j := 2 to i do
        write(' ');
      (write wrapped chars on new line)
      bufindx := bufindx - i;
      endline(i);
      colno := colno + i
    end
  else
    (no filling or autowrap)
    putchar;
    (check for full buffer)
    if insbeg = insend - 1 then  (*full buffer*)
      prompt(10)
end;  (* insertchar *)
procedure insformatcmd;
(* this proc adds format commands to the buffer and handles special insert entries, e.g., tables. *)
begin
(* insformatcmdchar *)
if bop then
  endline(0)
elself (ch = cr) and not fill then begin
  (* ignore cr and fill *)
  ch := lm;
  endline(0)
end else if ch = filloffch then begin
  outchar;
  fill := false
end else if ch = fillch then begin
  outchar;
  fill := true
end
end;
(* insformatcmdchar *)
procedure insprocess(action: char);
(* this proc processes the inserted text and surrounding text as necessary. note that emptybuf indicates the creation of a new file. *)
begin
(* insprocess *)
if action = accept then
  action := 'A'
elself
  action := 'Q';
case action of
  'A';
  begin
    if emptybuf then begin
      imotomarker(hotchar, -1);
      processbuf(insoos, buflimit)
    end else begin
      insquit := inshea - 1;
      processbuf(insoos, inshea - 1)
    end;
    emptybuf := false
  end;
  'Q';
  begin
    (* no insertion occurs *)
inbeg := inspos;
  bufindx := insbeg - 1
end
end {case}
end;
begin
if emptybuf then begin
  rowno := 1 + msgins;
  colno := pmargin + 1
end else begin
  (* blank out the end of the line *)
  xyaddr(colno, rowno);
  for i := colno to lastcol do
    write(' ');
  blankln(rowno + 1);
  (* insert before cursor *)
  ch := getbufchar(-1)
end;
xyaddr(colno, rowno);
finished := false;
inpos := insbeg;
repeat
  read(ch);
  if newline then
    insformatcmd
  else if textchar then
    insertchar
  else if ch = erase then
    inserterase
  else if (ch = accept) or (ch = quit) then begin
    insprocess(ch);
    finished := true
  end else begin
    errormsg(b);
    promot(2);
    xyaddr(colno, rowno)
  end
until finished;
if not emptybuf then begin
  repeat
    ch := getbufchar(-1)
    until (bufindx <= insquit) or bot;
  displayscreen(pmargin, 0, +1)
  end else
  write(clear)
end;

(* insert *)
procedure locate;
(*this proc performs a pattern search thru the file for the input pattern string and positions the cursor at the "found" pattern or at the end of the searched area.*)
begin
  errormsg(21)
end;

procedure move;
(*this proc allows the user to move text from one place to another in the buffer. The body to be moved is delimited by 2 cursor settings. The new locn is also set with the cursor. Note the moved material will precede the cursor position*)
begin
  null
end;

procedure replace;
(*this proc will utilize the locate proc to replace a specified pattern w/ another one either once or multiple times w/in the delimited area of text.*)
begin
  errormsg(21)
end;
procedure Addfile;
(*this proc adds the user-specified file to the
buffer contents at the locn indicated by nm*)

var
  addpos, addquit: integer;

begin
  filename('A');
  setmarker;
  checkmarkers('A');
  if markersok then begin
    addpos := bufindex;
    reset(infile, name);
    jmotomarker(botchar, -1);
    jmotomarker(nm, +1);
    read(infile, ch);
    repeat
      read(infile, ch);
      bufindex := bufindex + 1;
      buffer[bufindex] := ch
    until eof(infile) or (bufindex = insend - 1);
    insbeg := bufindex + 1;
    addquit := bufindex;
    if not eof(infile) then
      errmsg(23)
    else begin
      prompt(8);
      repeat
        read(inchar)
      until inchar in ['n', 'y'];
      if inchar = 'y' then
        delmarkers
      processbuf(addpos, addquit);
    repeat
      ch := getbufchar(-1)
    until (bufindex <= addquit) or bot;
    displayscreen(1margin, 0, +1)
  end;
end;  (* Addfile *)
procedure Xtractfile;
(*this proc writes the buffer contents delimited
by bm and im out to a file*)
begin (*Xtractfile *)
  filename('X');
  setmarker;
checkmarkers('X');
if markersok then begin
  rename(outfile, name);
  write(outfile, botchar);
  jmtomarker(botchar, -1);
  jmtomarker(bm, +1);
  ch := getbufchar(+1);
repeat
  write(outfile, ch);
  ch := getbufchar(+1)
until ch = em;
write(outfile, eotchar);
  rorme:e (of late I/O block)
  prompt(8);
read(inchar)
  repeat
  until inchar in ['n', 'y']
if inchar = 'y' then
  delmarkers
end
end; (* Xtractfile *)
begin
if fileread then begin
exit := false;
prompt(1);
repeat
xyaddr(colno, rowno);
read(inchar);
if inchar in
{cr,down,left,right,up,'D','U','D','e','h'} then
if not emptybuf then
movecursor
else
errormsg(2) (* text not present *)
else if inchar in edcmdset then begin
case inchar of
'A':
begin
prompt(21);
Addfile
end;
'C':
begin
prompt(6);
copy
end;
'C':
{Cursor movement}
prompt(23);
'd':
begin
prompt(5);
delete
end;
'i':
begin
prompt(2);
insert
end;
'l':
begin
prompt(17);
locate
end;
(* edit *)
'm':
    begin
      prompt(7);
      move
      end;
'q':
    exit := true;
'r':
    begin
      prompt(18);
      replace
      end;
's':
    begin
      prompt(16);
      setmarker
      end;
'x':
    begin
      prompt(20);
      exchange
      end;
'X':
    begin
      prompt(22);
      Xtractfile
      end
end; (* edit case *)
if not exit then
  prompt(1)
end else begin
  errmsg(4);
  prompt(1)
end
until exit
end else
errmsg(3)
end; (* edit *)
begin

(** mainline of scope **) 

(*this is the driver for the command level and thus SCOPE. It allows the user to read in and process the desired file; edit and format the file in the buffer and to save the file by writing it to the printer or to storage.*)

stop := false;
namingfile := false;
fileread := false;
cmdset := ['r', 'w', 'l', 'e', 'q', 'f'];
edcmdset := ['d', 'i', 'a', 'c', 'm', 'r', 's', 'x', 'A', 'C', 'X'];
cursormoveset := [up, down, right, left, cr, 'b', 'e', 'D', 'U'];
terminal; (* prompt msg in proc *)
printer;
setdefaults;
setformatcmds;
write(clear); (* clearsceen *)
repeat
  repeat
    prompt(4);
suserinout(1);
cmdchar := cmd[1];
if cmdchar in cmdset then 
case cmdchar of
  'r':
    redraw;
  'w':
    writeraw;
  'l':
    listorrocessed;
  'e':
    edit;
  'q':
    stop := true;
  'f':
    format
end (* decode case *)
else
  errormsg(1)
until stop;
write(clear)
end.
LIST OF REFERENCES


8. Elite 1520 Technician Manual; Datamedia Corporation; Pennsauken, New Jersey.

9. Elite 2500 Instruction Manual; Datamedia Corporation; Pennsauken, New Jersey.


13. Joy, W. N.; An Introduction to Display Editing with VI; University of California, Berkeley; April 1979.


21. Triyono; "Design Methodology for an "Easy-to-Use" Text Processor;" M. S. Thesis; Naval Postgraduate School; Monterey, California; June 1978.


<table>
<thead>
<tr>
<th>No.</th>
<th>Initials and Title</th>
<th>Address or Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Defense Technical Information Center</td>
<td>Cameron Station, Alexandria, Virginia 22314</td>
</tr>
<tr>
<td>2.</td>
<td>Library, Code 0142</td>
<td>Naval Postgraduate School, Monterey, California 93940</td>
</tr>
<tr>
<td>3.</td>
<td>Department Chairman, Code 052</td>
<td>Department of Computer Science, Naval Postgraduate School, Monterey, California 93940</td>
</tr>
<tr>
<td>4.</td>
<td>LCDR Frank Burkhead, USN</td>
<td>22306 Capote Drive, Salinas, California 93908</td>
</tr>
<tr>
<td>5.</td>
<td>Professor Lyle Cox, Code 052C1</td>
<td>Department of Computer Science, Naval Postgraduate School, Monterey, California 93940</td>
</tr>
<tr>
<td>6.</td>
<td>Captain Lisa Talmage, USMC</td>
<td>Office of Manpower Utilization, Building 2009, Quantico, Virginia 22134</td>
</tr>
</tbody>
</table>