

HLSEW
EDITING SYSTEM

by

DAVID BARTLETT AARONSON

B.S., Troy State University, 1977

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

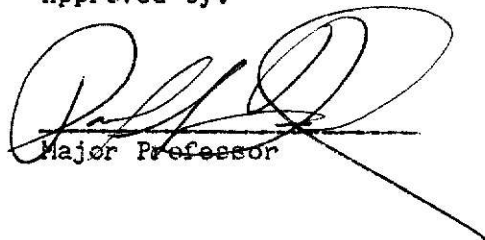
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, KS

1982

Approved by:



Major Professor

SPEC
COLL
LD
2668
.R4
1982
A22
C.2

A11200 189059

i

ACKNOWLEDGEMENT

I would like to take this opportunity to thank Dr. David Gustafson for the many hours of guidance, support, and review rendered.

Also, I want to extend thanks to my wife, Barbara, and daughter, Heather, without who's love and patience this report could never have been done.

TABLE OF CONTENTS

<u>Chapter</u>	<u>Page</u>
I. INTRODUCTION	1
Scope and Purpose	1
Constraints	2
Review of Other Approaches	4
II. DESIGN	7
Preliminary Considerations	7
The Line Editor	11
The Screen Editor	13
Command Interpreter	14
III. IMPLEMENTATION	17
The Line Editor Unit	17
The Screen Editor	29
The Command Interpreter	29
IV. TESTING	30
V. ENHANCEMENTS AND EXTENSIONS	32

APPENDIX A. HLSEW User's Manual	A-1
APPENDIX B. Screen Editor Source Code	B-1
APPENDIX C. Line Editor Source Code	C-1
APPENDIX D. Program Test Data	D-1
REFERENCES	R-1

FigurePage

1. A Screen Editor Front End to a Line Editor	10
2. Line Editor Design	12
3. Screen Editor Design	15
4. HLSEW High Level Design	16
5. Logical Organization of Line Editor	19
6. Line Editor Flow of Control	21
7. INSERT TEXT Flow of Control	22
8. Logical Division of Screen Editor	25
9. The Screen Handler	26
10. Editing Procedure Processes	28
11. Enhanced Terminal Access	33
12. Enhanced Module Communications	34

CHAPTER I

INTRODUCTION

Purpose

The Computer Science Department of Kansas State University is currently in the process of developing a High Level Software Engineering Workstation (HLSEW). This interactive workstation will be the software implementation of an "intelligent terminal", designed to aid in program development. It will consist of five modules: (1) The Command Interpreter, (2) The Software Engineering Analyzer, (3) The Translator, (4) The Storage System and (5) The Editor.

The command interpreter serves as the main driver for the program and provides user access to the other modules. The analyzer provides the programmer with Halstead's and McCabe's software metrics throughout the development of his program. These metrics consist of information such as the anticipated volume, the complexity, and the implementation level. The programmer will be issued these measures at the completion of every block or can request them at any time during program use. The translator converts a program written in a program design language (PDL), which supports CASE statements, DO WHILE, and REPEAT UNTIL syntax, into compilable COBOL source code. The storage system serves as the data filer. All information is mapped and accessed to secondary storage by the storage system on a line by line basis. The editor serves as the tool which the programmer utilizes to create and edit text. The purpose of this report is to describe the design and implementation of the editor for the HLSEW.

A custom editor is required because the HLSEW specification

dictated that text be handled by the storage system in a very particular manner. Each line of text, as it is entered would also be stored. This allows the storage system to maintain a reference point within the program and provides the capability to analyze each line of code as it is entered. Most editors do not handle text in this manner and would require modification. Modifying the UCSD Editor to handle text on a line by line basis required the source code which was not available.

Constraints

Initial constraints on the Editor included size of the program, portability, power of the editor, user interface, extensibility, and flexibility. Giving consideration to each of these areas was essential.

One of the primary considerations is size, because the system has to be run on a microcomputer. It is anticipated that the Editor, the Storage System, the host operating system and portions of the user's program will have to reside in main memory simultaneously. As many of today's microcomputers are limited to 64 K bytes of main memory, each module has to be as compact as possible.

The design process has to take into account the desirability of moving the software from one machine to another. Although complete portability is very unlikely, because of the great disparity in existing hardware, portability has to be kept in mind and as many portable features, as possible utilized.

The user interface is also of utmost importance. The Editor, which serves as the user's window to the system, has to be "friendly". A

"friendly" interface is defined as one which allows the work to get done with the least amount of strain and annoyance to the user. The user should be guided into correct actions, politely prevented from incorrect actions, and never made to feel lost or abandoned.

A consideration which usually conflicts with the friendliness of the interface is the power of the editor. To get the "most for your money" is always desirable and in this case, to pick the most powerful set of instructions to fit in the limited size memory available became a primary consideration.

Extensibility is also important. As the microcomputer industry grows, an increasing number of machines are becoming available with larger main memories. If the restriction of size were lifted, it is desirable to be able to extend the Editor's power by adding additional procedures which would utilize the primitives designed into the system.

The constraint which is often ignored is flexibility. It is anticipated that many, and varied, types of equipment may be encountered. Because of this fact, the editor has to be flexible enough to operate in most any environment. For example, it has to be able to operate with either hard copy output devices (e.g., teletypewriters) or CRT type displays.

Some additional concerns are understandability and modifiability. It is important that the code be understandable and modular so that future modifications can be made with minimal effort. Understandability and modifiability go hand in hand and are essential to a good product. In addition, consideration has to be given to efficiency and reliability.

Review of Other Approaches

The basic approach to editors is usually from either of two directions, the line oriented editor or the screen oriented editor. The line oriented editor, which is more or less the traditional editor, is primarily designed for hard copy devices. Lines of text are often referred to by line numbers and commands are issued to perform manipulations on text lines using a notational syntax (e.g., DELETE 23,30, indicating the deletion of lines 23 through 30). On the other hand, a screen editor is designed for use with CRT type displays, and provides the user with a window into his text file. Changes are made directly on the screen with immediate feedback and the user always views the current version of the text. This reduces the chance of error and spares the user the overhead of explicit requests to view lines.

The Department of Computer Science at Kansas State University utilizes a program called PEDIT on its Interdata 8/32 computer. PEDIT is a line oriented editor written in approximately 3300 lines of PASCAL code [KANS79]. Although this editor is very powerful, the commands are somewhat clumsy. Also, the user is usually tied to some sort of hard copy (printout) for reference. Examples of commands include: UP, DOWN followed by an integer, indicating relative movement within the text file; INSERT, DELETE, and CHANGE, usually followed by a set of arguments; and COPY, which is used to move blocks of text. There is currently an effort to design a screen editor for this system, the size of which, should exceed 4000 lines of PASCAL code.

A very powerful screen editor and one which is becoming the de facto standard for microcomputers, is part of the UCSD P-System. The UCSD P-System consists of a filer, editor, compiler and operating

system. UCSD Pascal is compiled to P-Code, which is in turn either interpreted by the operating System or run directly on the specific microcomputer. This allows the compiled P-Code to be transported to any machine with the UCSD System. The editor is a full screen editor which provides great flexibility and power. Its user interface is excellent. Prompt lines are always displayed, indicating the options available to the user and the commands are almost all single keystroke commands [BOWL80].

Intel Corporation developed an editor called CREDIT. CREDIT is a screen text editor and line text editor rolled into one [GRAP80]. The screen is divided into two parts. One part uses cursor positioning and you perform the operations normally associated with a screen editor. The other part is a line oriented, scrolling display. The user can move between the two windows at will, facilitating both types of editing. Example screen commands include: REPLACE, which types over existing text with replacement new text; DELETE, which either deletes one character or can delete all characters between two specified boundaries; and PAGE, which gets the next screenful of text. Example line oriented commands are PRINT, FIND, DELETE and SUBSTITUTE, which are each followed by a set of arguments.

A novel design for an editor was implemented by the Department of Computer Science at The University of Arizona [FRAS79]. It consists of a line editor and a screen editor front end. The screen editor front end manages the information on the screen, interprets the commands from the user and passes them, in appropriate form, to the line editor to do the actual editing. This approach offers all the advantages of a screen editor without the normally large volume of code associated with their

implementation. It also enhances the portability of the system because the terminal specific code (the screen front end) is kept to a minimum. In addition, development time for a screen front end, which utilizes a host line editor, is much less than for a complete screen editor.

All of these editors have one thing in common that would not be acceptable to the HLSEW Editor. They manipulate their text files as whole units and when there is a file to be edited or read, the entire file is loaded into main memory. On the other hand, each offers some ideas which were useful in designing the HLSEW Editor.

CHAPTER II

DESIGN

Preliminary Considerations

The dilemma encountered when starting to design this editor was whether to choose a small, powerful line oriented editor or a big, powerful screen oriented editor. To resolve this problem several questions had to be addressed.

What constitutes a good editor? A good editor should be both easy to learn and easy to use (or powerful). These terms are difficult to define. Generally, the editor is easy to learn if the interface is friendly, and it is easy to use if it is sufficiently powerful enough to accomplish desired tasks with a minimum of labor.

Many criteria are important to a good user interface. Simplicity deserves special attention because insufficient attention in designing a simple user interface is the most common cause of bad interfaces [SNEE78]. A simple interface provides only the minimal set of operations required to accomplish the mission and does not confuse the user with extraneous or seldom used operations. In addition to being simple, the editor must give the user feedback and prompt the next command, especially if the user is not secure in his knowledge of the editor. Commands should output enough text to identify all changes that have been made and precisely indicate source text position. The user needs protection from common mistakes. The editor should be able to prevent typos in commands from destroying the file or parts of it. Finally, for a good interface the user must be able to access any part of the source text anytime during the edit session. This may seem

trivial, but anyone who has had to bounce back and forth editing among two or more files knows the value of this feature.

The goal of ease of use often conflicts with ease of learning. The power of an editor is usually directly proportional to its difficulty to learn, therefore some tradeoffs are required. At a minimum, the editor has to handle the creation of text, movement through a text file, and insertions and deletions on a line by line basis. Any addition to this minimal set has to be weighed against the impact to the interface. As the power of the editor can be significantly increased by a command which can find target strings of text and replace them with new strings, without diminishing the interface, it was also included. With this minimal set of features in mind, the next problem had to be tackled.

How can the goals of the implementation be accomplished and yet remain within the constraints specified in Chapter 1? Many of the features mentioned above indicate a screen oriented editor, yet the fact that text has to be handled one line at a time indicates that a line oriented editor would be more suitable. Another factor which points to a line oriented editor is the flexibility required to run on different types of output devices. A full screen editor is of little use if your terminal is a hard copy device. The limited size of main memory available lends itself to a line oriented editor (since screen editors are usually more voluminous than line editors), as does the requirement for portability. Screen editors are usually more tied to the hardware than are line oriented editors because of terminal dependent features which must be exploited by the editor. It became evident at this point that some compromise had to be made.

The idea of a line editor with a screen front end, as described in

Chapter 1 became an intriguing possibility and deserved more investigation. Utilizing a line oriented editor as the base for the system allows it to meet the requirements of flexibility, allows it to deal with the storage system one line at a time, and hopefully keeps the size of the project down. A screen handler used as a front end to the line editor enables the interface with the user to be as friendly as possible, and at the same time increases the power of the editor. This, in fact, is the route that was taken; a screen editor front end designed to manage the information on the screen, interpret commands from the users and pass them to the line editor to do the editing. Figure 1 is a conceptualized drawing of a line editor with a screen front end, joined to form an editing "System".

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

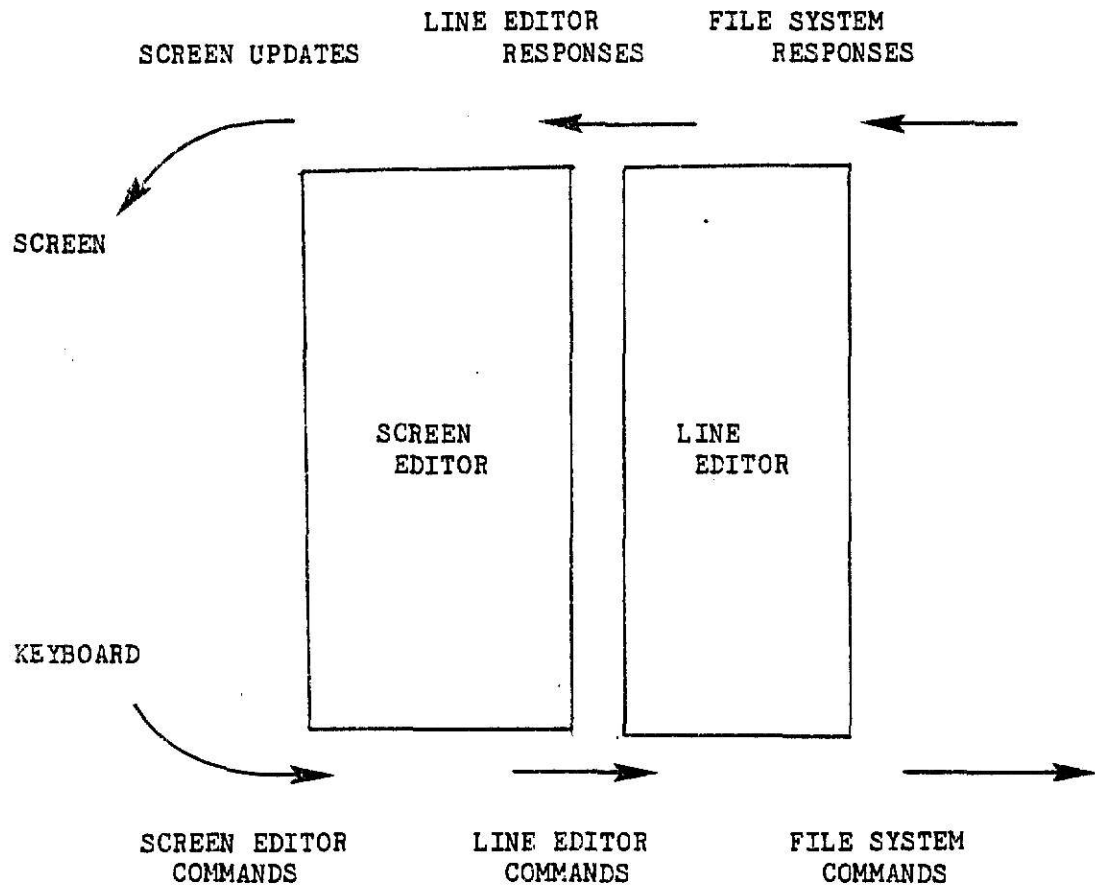


Figure 1. A Screen Editor front end to a Line Editor

The Line Editor

The Line Editor portion of the System was designed first. In addition to the minimal set of activities (Insert, Delete, List, Find & Replace), that were stated earlier, some additional goals were established. The ability to verify changes made during find and replace operations should be available as an option to the user. Because of the sensitivity to column changes in COBOL, some sort of tab setting function is necessary and along with this tab setting function, the ability to change the tab character is required. Error messages are required and should be written from a central procedure. Finally, some help to the inexperienced user should be available. Procedures to accomplish these capabilities were included in the design.

The requirement for flexibility dictates that this Line Editor has to be able to stand alone (without the screen front end), in the event it were to be used on a hard copy device. This caused the design of the Line Editor to include some procedure which reads commands from the keyboard and controls the editing procedures. The Line Editor also has to communicate with the Storage System and the Screen Editor. It must be able to fetch and store lines of text from the File System and receive commands from the Screen Editor. All output from the Line Editor has to be in the normal scrolling fashion, characteristic of line oriented devices. Figure 2 represents the initial design of the Line Editor.

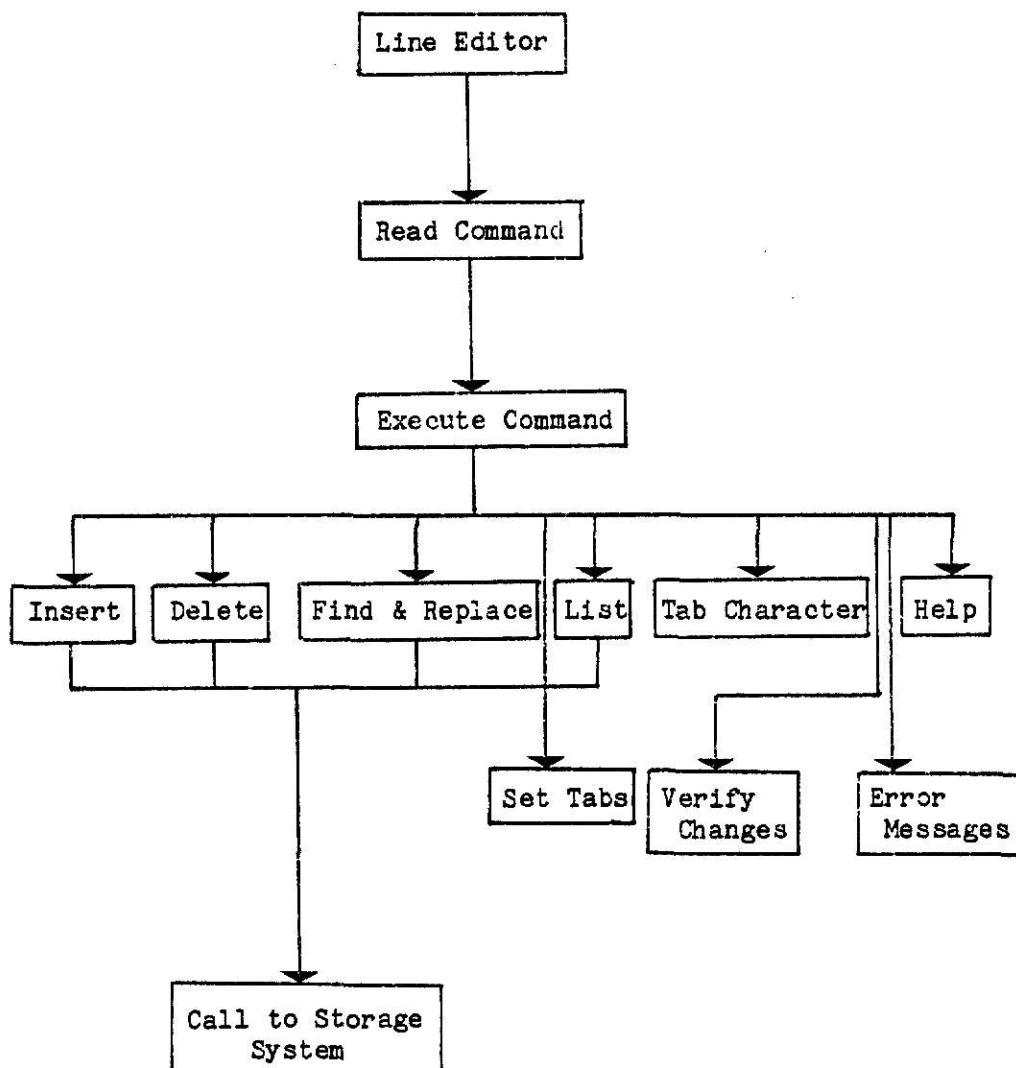


Figure 2. Line Editor Design

The Screen Editor

Since the Line Editor is going to do most of the editing work, all that is required of the Screen Editor is to manage the output to the screen, interpret commands from the keyboard, and make calls to the Line Editor. These functions essentially provide the friendly interface, which is one of the primary goals. However, there are additional functions which the Screen Editor can accomplish because of its unique properties.

Besides providing a friendly interface to the user, the screen display adds another dimension which should be exploited. The user sees his text as it appears in the file. He is now able to move a cursor among the text characters and perform operations as required. In order to take advantage of this, several capabilities were added to the Screen Editor design, which would extend the power of the System. First, the user is able to exchange text on the screen with new text. The ability to position the cursor and replace the character on which the cursor is sitting is a valuable tool. Second, the user is able to insert not only lines of text, but insert text within a line. That is, to position the cursor someplace on the screen and insert characters in a line causing characters to the right of the insertion to be moved, as new ones are entered. Finally, the extension of character deletes, rather than only line deletes, is very useful. Again, moving the cursor to the appropriate position and deleting characters, causing the remainder of the line to shift as deletes are made, adds to the power of the System. All of these extensions to the Line Editor allow the user to visualize his changes within the context of the surrounding text and decide on their acceptance prior to causing the changes to be stored. This

reduces errors and improves performance. Figure 3 portrays the initial design of the Screen Editor.

Command Interpreter

In order for either the Screen Editor or the Line Editor to be invoked upon entry into the system, another layer of software is needed. This outer layer initially greets the user and determines the type editing to be done. By specification, it would also act as the driving routine for the other modules in the HLSEW. Figure 4 shows the complete HLSEW high level design.

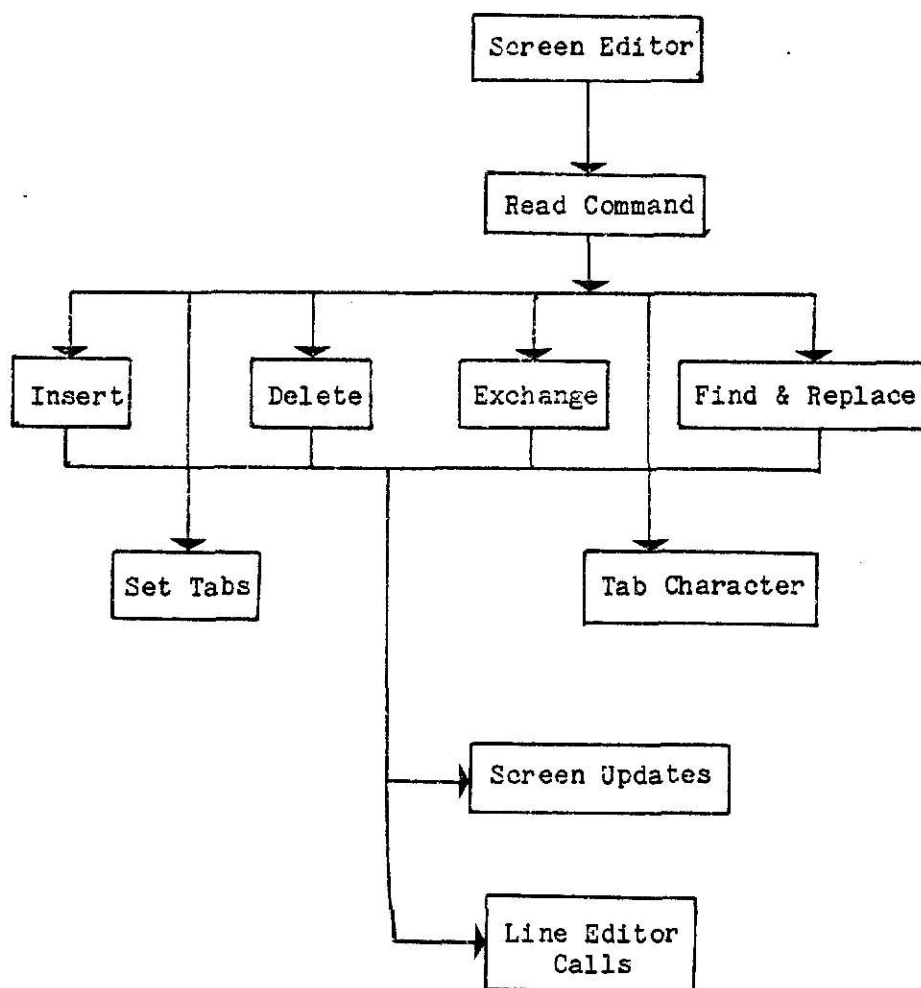


Figure 3. Screen Editor Design

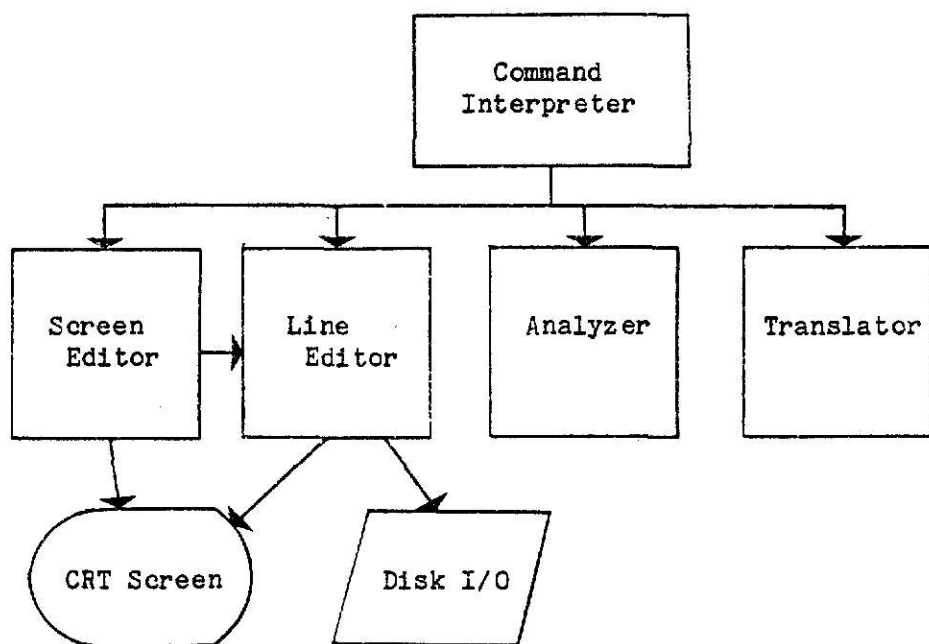


Figure 4. HLSEW High Level Design

CHAPTER III

IMPLEMENTATION

The Editing System for the HLSEW is implemented on the PIQ/3 Microcomputer in UCSD Pascal. It consists of approximately 1800 lines of source code, which includes some internal documentation and 10,500 bytes of object code. UCSD Pascal supports the separation of procedures and functions, or groups of them, from the main program. This is useful when creating programs too large for the edit buffer. These groups are compiled separately and are known as "Units". They are linked with the main program prior to execution. Because this implementation exceeded the size of the editing buffer, the Line Editor was implemented as a unit to be used by the Screen Editor.

The Line Editor Unit

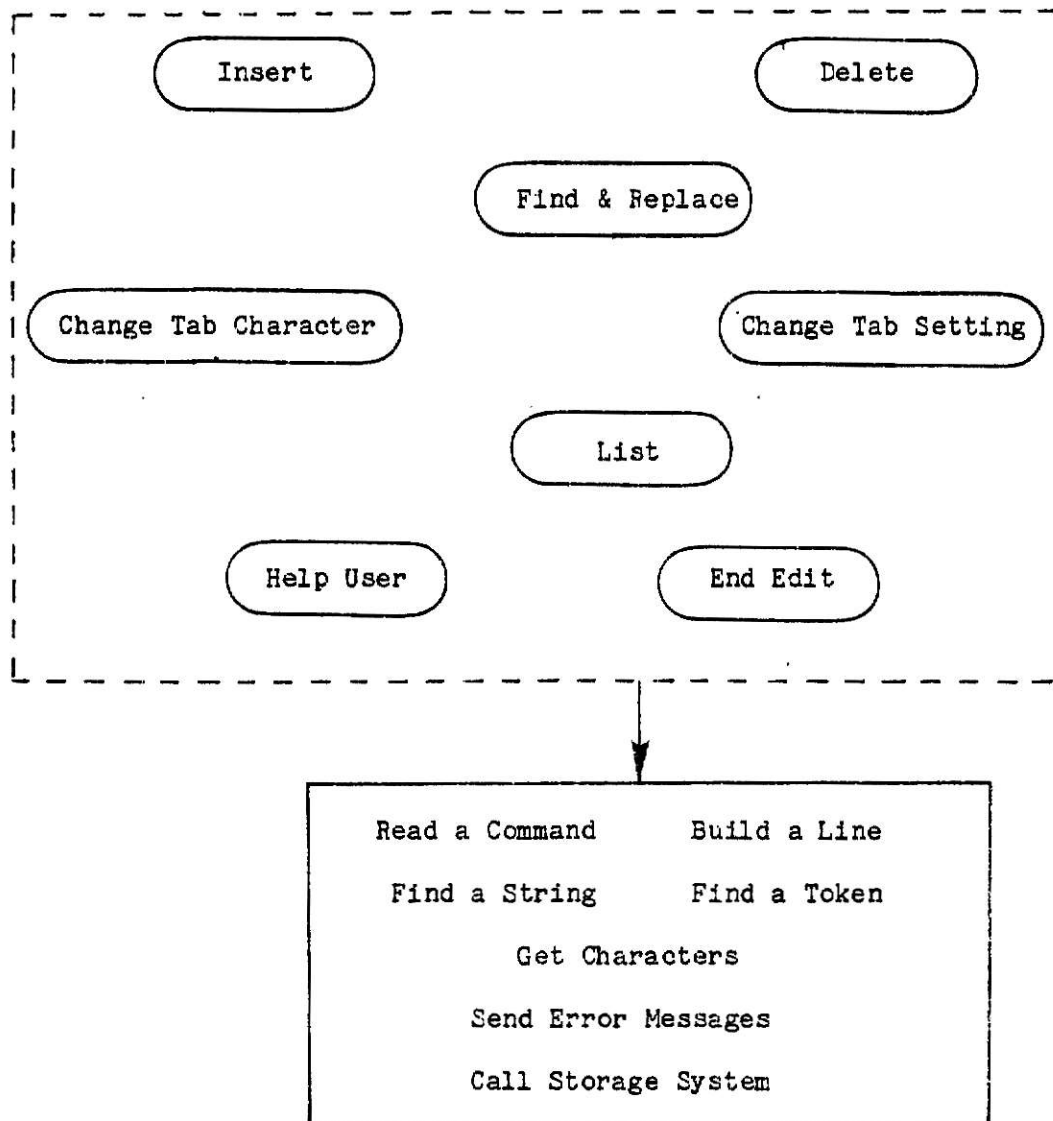
The Line Editor Unit is the foundation of the HLSEW Editing System. It is approximately 1000 lines of code and contains the core editing procedures. It communicates with the Storage System and has the capability to stand on its own as an Editor.

All the global variables which are used by the Editing System are contained in the Line Editor Unit. The basic global data structure used by the Line Editor is of type LINE, which is an array of 80 characters. This represents a line of text and is used in two instances. The first instance serves as the data structure that text is placed into for shipment to the Storage System. As text is entered from the keyboard it is stored, one character at a time, in the data structure. The second

instance is used to store Editor commands as they are entered from the keyboard. These two data structures are implemented globally to facilitate their passage between procedures. Although a more structured approach might have been to pass them among each procedure as a parameter, it is felt the overhead to do this would have been excessive. Operations on this data structure are insert and delete. The other global variables used are minimal in number. Several integer variables are needed. One holds the current line number and the other two are pointers, for each data structure. A few booleans are also needed to determine such things as whether or not the user was finished editing, whether or not the Line Editor was being used in the stand alone mode, and a flag to determine whether or not all changes should be verified. Also, a character variable, to house the tab character, and a set of integers for the tab settings are needed.

The Line Editor Unit consists of eight procedures which do the core operations of insert, delete, find and replace text, list, change the tab character, change the tab settings, write help messages, and end the edit session. These core procedures are then supported by several utility procedures which provide services such as find a token, find a string, read a command, build a line of text, get characters from the keyboard, put out error messages, and communicate with the Storage System. Figure 5 represents the logical organization of the Line Editor.

Core Editing Procedures



Line Editor Utilities

Figure 5. Logical Organization of Line Editor

In general, the Line Editor waits for commands to be entered from the keyboard. The commands, as they are read, are placed into the data structure, `INPUT_LINE`, by the utility procedure, `READ_COMMAND`. This procedure also determines which core operation would be called by interpreting the first two characters of the command line. Control is then transferred to the core procedure. The core procedure must interpret the rest of the command line. To accomplish this, it utilizes the utility, `FIND_TOKEN`, which scans the command line and returns the next token. The core procedure then matches this token with what is allowed and either continues the parse of command line or calls the error message utility. Once the parse of the command line is completed, the core procedure then performs the operations directed. In most cases, this involves calls to the utilities which deal with the Storage System and either fetch, store or delete lines. Once the operations are complete, control is automatically returned to the driver of the program, which awaits another command. Figure 6 shows the flow of program control for a general command.

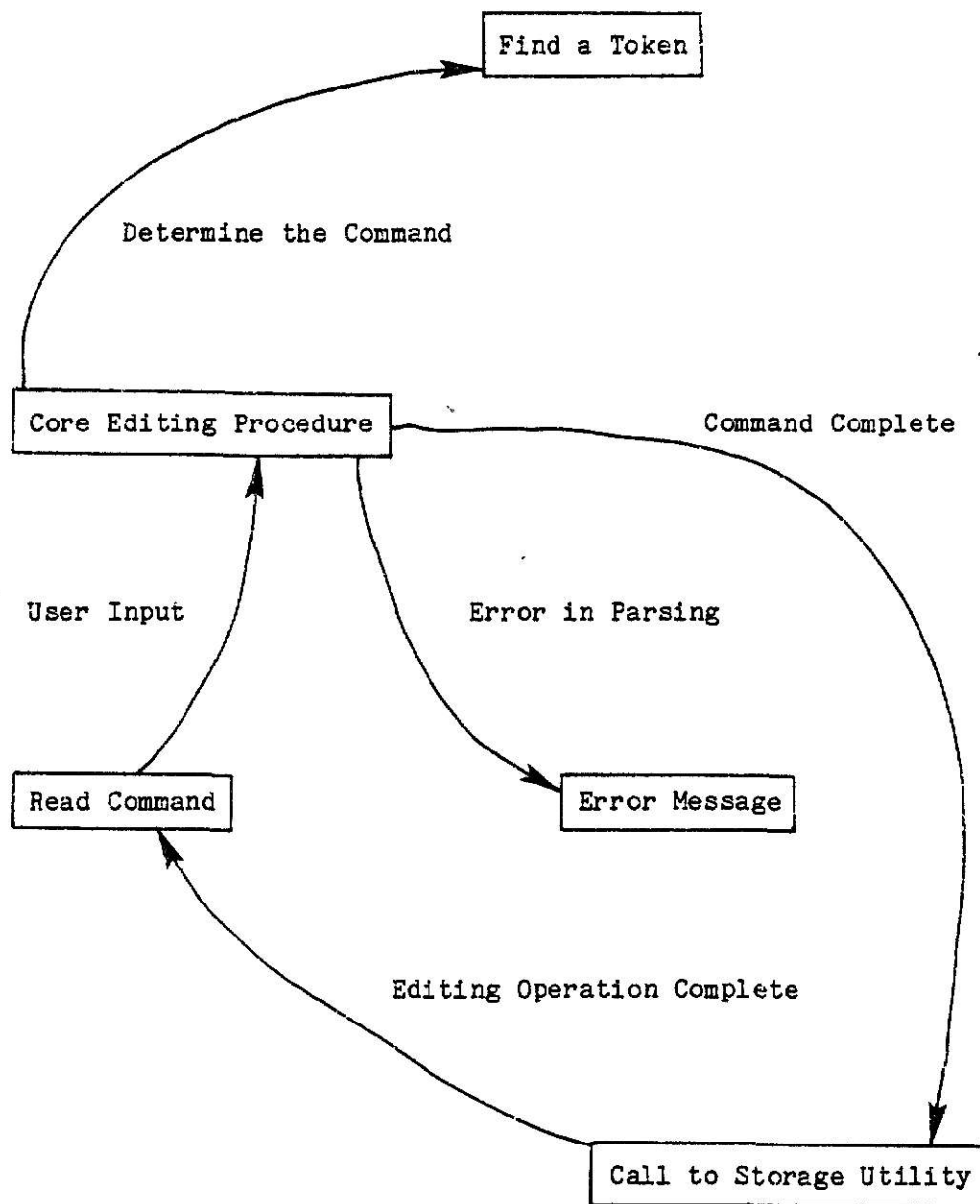


Figure 6. Line Editor Flow of Control

Of notable exception to this general form of operation is the procedure `INSERT_TEXT`. This procedure takes control of the program and calls to the utility `BUILD_A_LINE`. `BUILD_A_LINE` gets input from the keyboard and stores it in the data structure, `TEMP_LINE`, until a carriage return is encountered or the data structure is full. When `BUILD_A_LINE` is finished, a call to the storage utility is made to make the line permanent. Control is maintained by `INSERT_TEXT` and this cycle is repeated until the user indicates he wishes to leave this mode by entering a special control character. Figure 7 shows the flow of control of `INSERT_TEXT`.

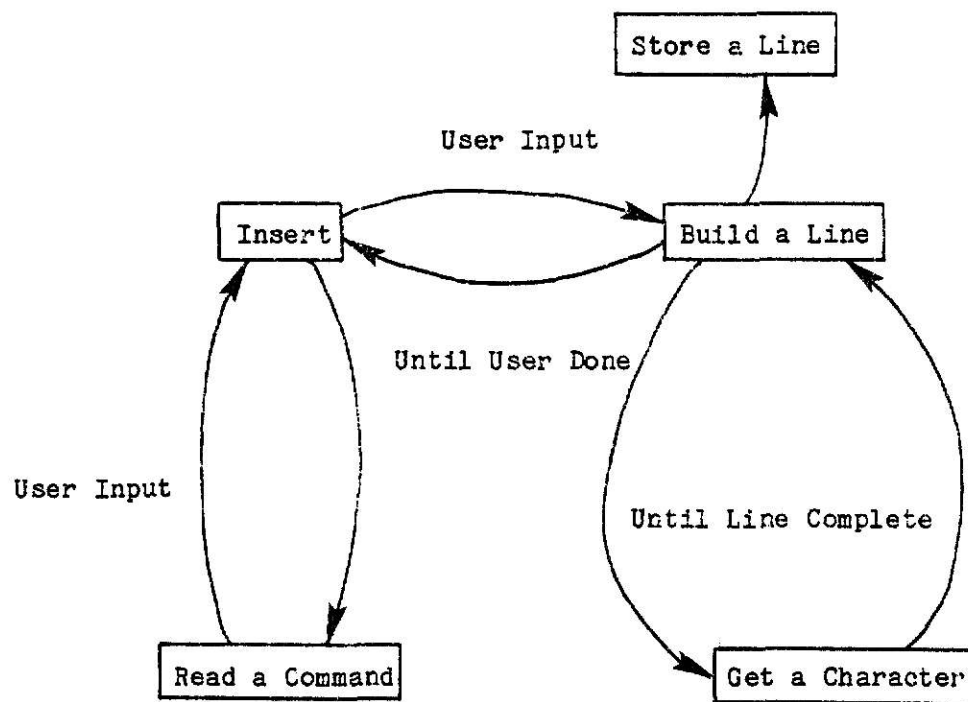


Figure 7. `INSERT TEXT` Flow of Control

The only procedure which manipulates characters within the data structure, TEMP_LINE, is the core editing procedure CHANGE_TEXT. This procedure is responsible for exchanging a target string of text with a substitute string. It first parses the command line to determine the target and substitute strings, and the number of changes to be made. Then a utility procedure is called to find the target string in the current line. If the target string is not found, CHANGE_TEXT continues to fetch a line from the Storage System and call the FIND_STRING utility to examine it. When a target string is found, it is replaced by the substitute string and the new line is then stored.

A source code listing of the Line Editor is provided at Appendix C.

The Screen Editor

The Screen Editor extends the Line Editor by providing a means to handle information on the screen, allow manipulations of text to be done within lines, and portray a "friendly" interface to the user. It is implemented in approximately 800 lines of source code and communicates with the Line Editor to perform core editing operations.

The Screen Editor utilizes the same global variables and data structures as the Line Editor. The only global variables added by the Screen Editor are two integers, ROW and COLUMN, which keep track of the position of the cursor.

The Screen Editor is divided logically into two sections, one to handle the screen and the other to extend the core editing operations. Figure 8 shows this relationship. The "screen handler" section includes procedures to move the cursor right, left, up and down, display menus on the screen, and page text to the screen. In addition, the screen handler must interpret commands, as they are entered from the keyboard, and call the appropriate screen editing procedures. The editing section contains procedures which parallel the Line Editor and make calls to its core editing operations.

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH THE ORIGINAL
PRINTING BEING
SKEWED
DIFFERENTLY FROM
THE TOP OF THE
PAGE TO THE
BOTTOM.**

**THIS IS AS RECEIVED
FROM THE
CUSTOMER.**

Screen Handling Procedures

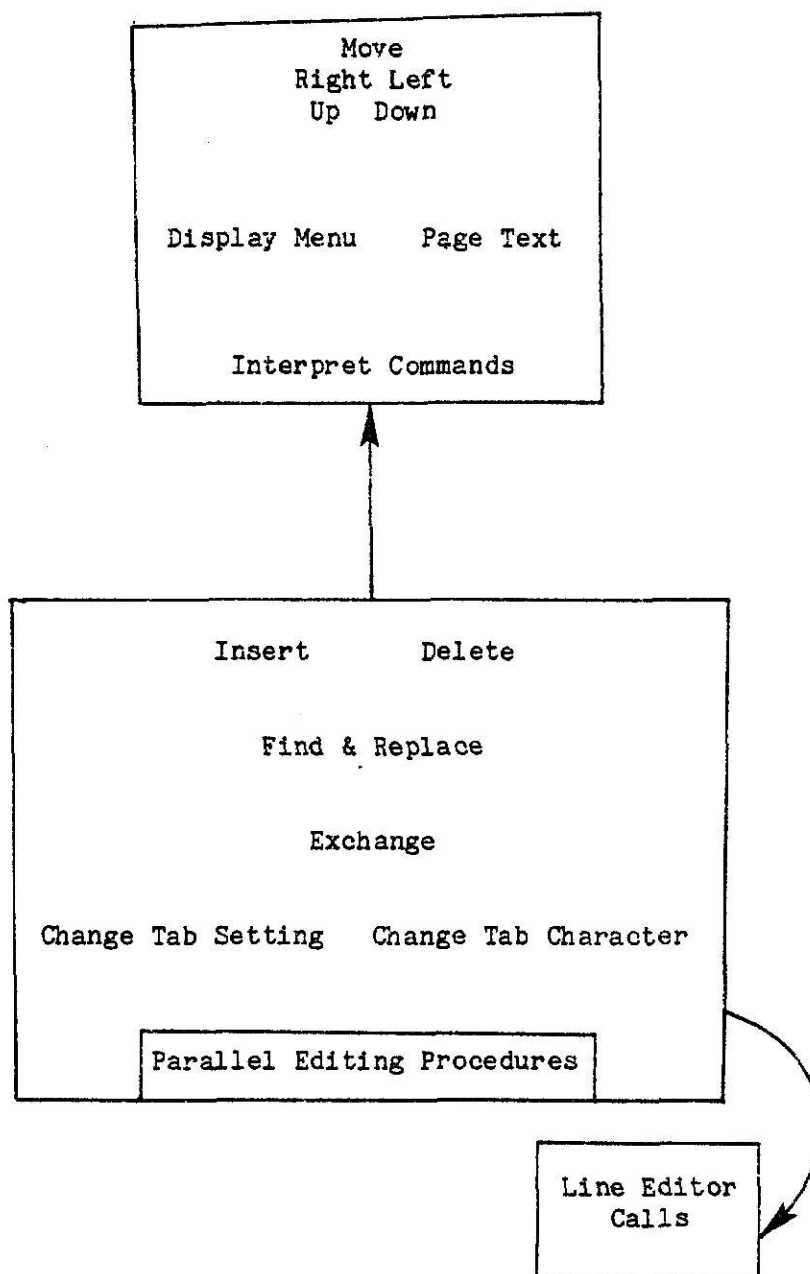


Figure 8. Logical Division of Screen Editor

The screen handler section provides the "friendly" interface to the user in several different ways. First, the screen handler displays command menus. These menus are layered, so that a response to one will cause another to appear, if appropriate. This serves to guide the user through the editing steps. The screen handler "watches" the keyboard for a response to the menus. It expects only a discrete set of inputs from the user and inputs outside of this set is ignored. This acts as a filter to eliminate bad input and prevents many editing errors. The responses are mostly single keystroke commands, which reduces the overhead to the user. The cursor can be positioned anywhere in the text and when the screen limits are reached the cursor is either moved to the successive line or a new screenful of text is displayed. This allows the user access to any part of his text file. Figure 9 is a representation of the screen handler.

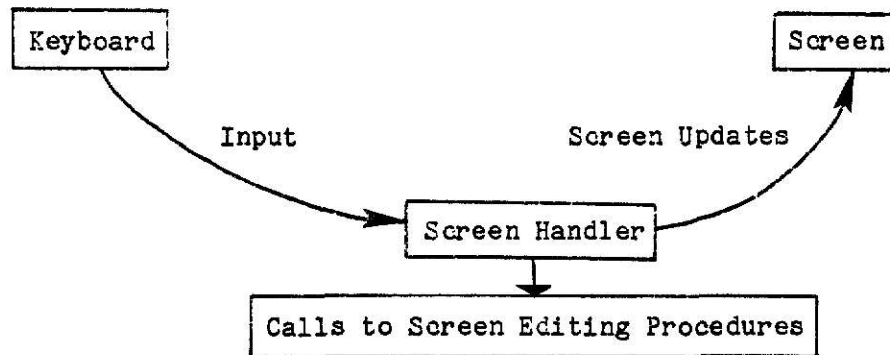


Figure 9. The Screen Handler

The section which interfaces with the Line Editor is responsible for determining the editing changes that are being made on the screen and "packaging" a command line which the Line Editor will recognize. Generally, this is accomplished in the two manners, operations on complete text lines and operations within a given text line. The particular editing procedure called, sends to the screen handler the discrete set of input that it is willing to accept. Initially, when the screen handler returns valid input to the editing procedure, the screen is updated and the mode of editing (complete line or within a line) is established. If the editing mode is complete lines then a counter is initiated to determine the number of lines concerned. The number or range of lines is placed in the global data structure, INPUT_LINE, which houses commands for the Line Editor. The appropriate core editing procedure is then called via the Line Editor to perform the required work.

If the editing is to be done within a line, then the line on which the cursor is sitting is retrieved from the Storage System via the Line Editor. As input is received, the screen is updated and temporary changes are made to the data structure containing the line of text. Upon completion of operation, the user issues a special control character and the line of text is shipped to the Line Editor for storage. Figure 10 illustrates both modes of editing.

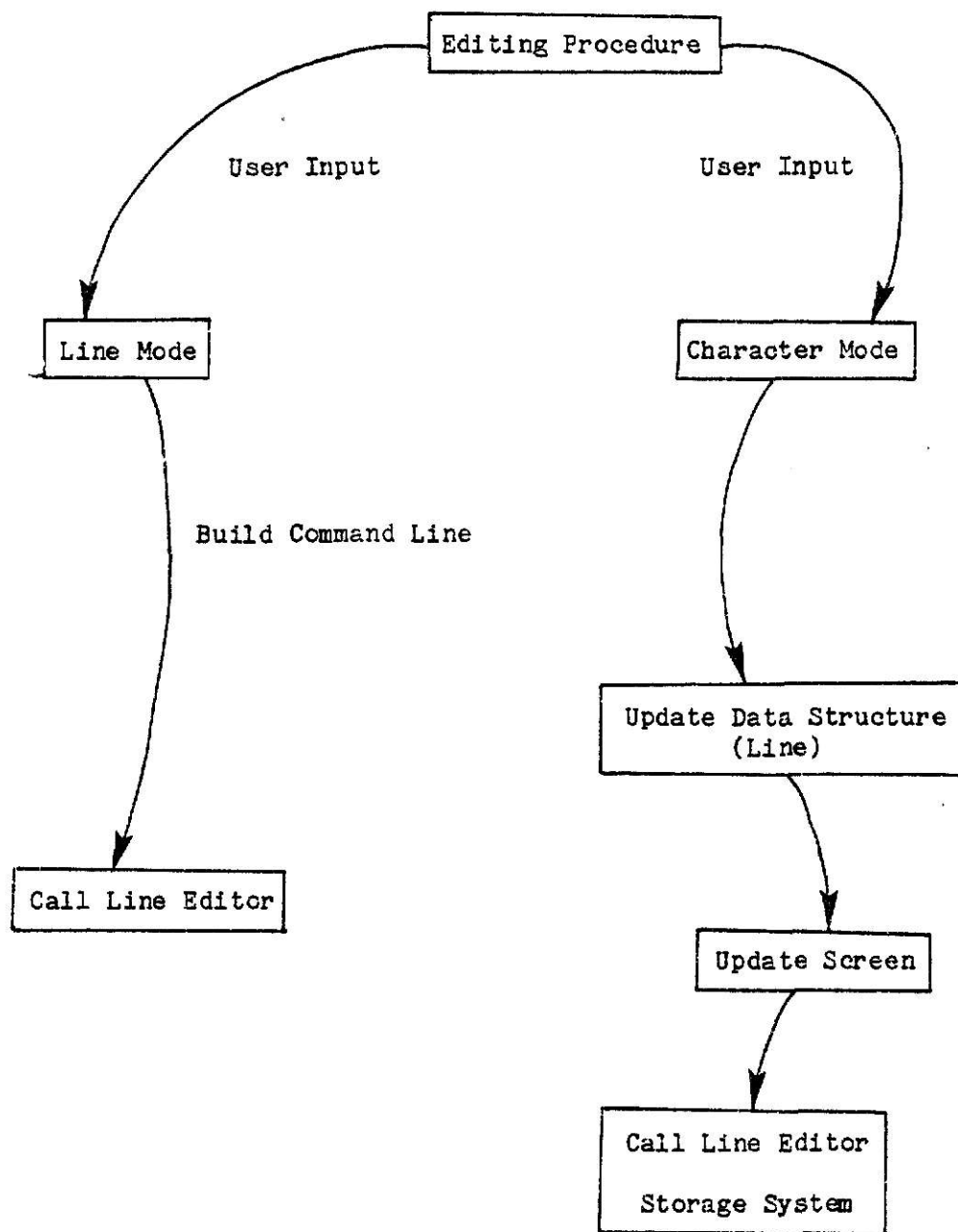


Figure 10. Editing Procedure Processes

At any time during the editing process, if the user wishes to abort the current operation, he uses the ESC key. All changes made to that point are ignored and the file, as it was before initiation of the operation, is displayed. A complete source code listing of the Screen Editor is in Appendix B.

The Command Interpreter

The Command Interpreter was implemented as the driving routine for the entire HLSEW. It uses the screen handler to display its command prompt line. The responses to the prompt line are filtered in the same manner as that which is used by Screen Editor. The additional function of the Command Interpreter is to determine which file the user wishes to utilize. This response is loaded into a string variable and issued to the appropriate module. The source code listing for the Command Interpreter is included with the Screen Editor in Appendix B.

CHAPTER IV

TESTING

The HLSEW Editing System is highly dependent on the Storage System which is currently not in executable form. This prevents the "complete" testing of the Editing System, as of the date of this report, however several methods have been utilized to verify its consistency, completeness, and correctness and examine its behavior during execution. These methods were both static and dynamic.

The program was statically checked for consistency by the UCSD Pascal compiler which insures proper syntax, compatible types, and parameter matching. Other static methods such as code inspection, desk checking and peer review are used throughout program development to determine that the code was consistent with the design.

Dynamic testing could be accomplished by instrumenting the program to calculate statement coverage. Each program point, into which control can be transferred, is identified and an array is established with as many positions as there are transfer points. At each transfer point a statement to increment the appropriate position in the array is inserted. The final values of this array are then printed at program termination. This technique provides an indication of test coverage and also shows the portions of the program with the heaviest use.

A set of test data was designed to exercise the program both structurally and functionally. This test data will be used to cause execution of each feasible statement. The test data also includes input to functionally test the domain of the program. Random values were chosen to cover domain boundaries, extremes and special cases. Test data

chosen is in Appendix D.

CHAPTER V

ENHANCEMENTS AND EXTENSIONS

As in most programs, there are several changes in the design of the HLSEW Editor System that would improve it. These enhancements were recognized during the implementation and debugging phases of the software life cycle. Primarily, there are two areas of interest; the manner in which the program addresses the specific terminal being used; and the inter-communication of the Line Editor and the Screen Editor.

Currently, the program accesses the CRT terminal through direct calls requesting operations such as erase the rest of the page, delete a line on the screen or insert a character. Although this works adequately, it reduces the portability of the program by making the transition to a new terminal more difficult. To change terminals, the program would have to be modified at each instance where it addresses a screen function of the terminal. A much nicer way to accomplish this operation would be to establish a table of terminal dependent information and assign a variable to each function. Subsequent porting of the software would then only require an update to the table.

Another approach to this problem can be found through the UCSD Operating System. A terminal that is used with the UCSD Pascal P-System must be "set up" for the System. This involves executing a "set up" program which maps terminal dependent features into the Operating System. The HLSEW Editing System could have been designed to make Operating System calls to accomplish specific terminal functions. Figure 11 shows both approaches.

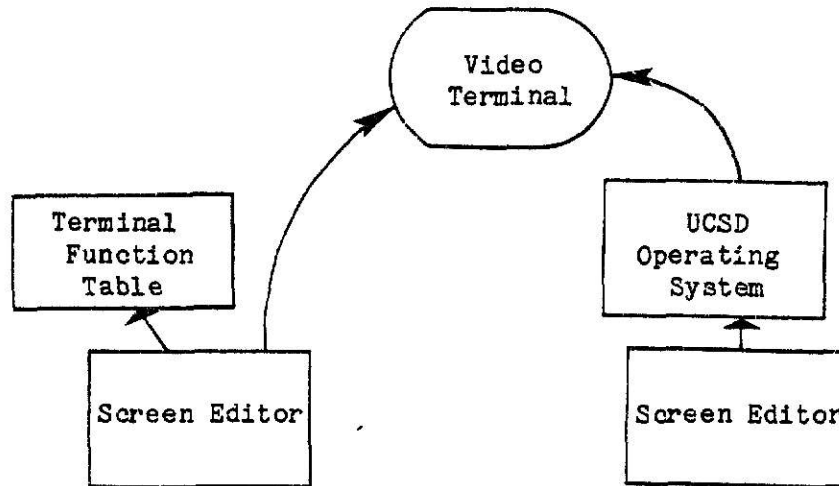


Figure 11. Enhanced Terminal Access

The second area of interest involves communication between the Line Editor and the Screen Editor. Presently, the Screen Editor makes calls directly to the Line Editor. (Figure 4) This is not necessarily a bad condition, however an enhancement could be made by requiring the Screen Editor to send its commands through the Command Interpreter. (Figure 12) If this was implemented the Command Interpreter would have to be "smarter", that is it would have to translate commands from the Screen Editor into a format recognized by the Line Editor. This would improve the portability of the System by making the Screen Editor independent of the Line Editor. If the System were moved and a new Line Editor used as the foundation, then the only changes required would be modifications to the Command Interpreter's Translator.

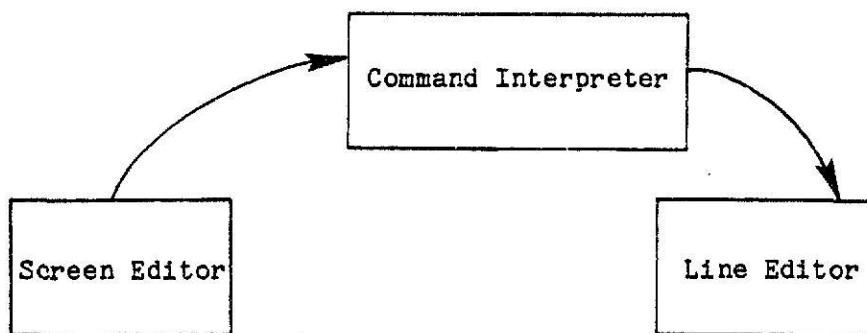


Figure 12. Enhanced Module Communication

The design of the program facilitates extensions. One which would add to the power of the Editing System, and is now considered necessary, is the addition of a "copy" command, designed to move blocks of text within a file. This was not originally included in the design, because of the restriction in dealing with text on a line by line basis. It could be implemented in the following manner.

When the copy mode is entered, a variable, `FIRST_LINE`, containing the current line number is saved. The cursor is then moved to the end of the block of text being repositioned and that line number is saved in a variable, `LAST_LINE`. The cursor is then relocated to the desired new location of the text. From that point, repeated calls to the Line Editor are made to fetch a line, store the line and delete the old line, over the range of line numbers, `FIRST_LINE` to `LAST_LINE`. This allows a line of text, referred to by its original line number, to be retrieved from the Storage System and loaded into the data structure, `TEMP_LINE`. `TEMP_LINE` is then inserted in its new location, causing it to be renumbered by the Storage System. Once it is stored then the old line is deleted.

APPENDIX A

HLSEW
USER'S MANUAL

TABLE OF CONTENTS

	<u>Page</u>
SECTION 1 INTRODUCTION	A-3
Introduction	A-3
The Command Level	A-6
SECTION 2 THE SCREEN EDITOR	A-8
Insert	A-8
Delete	A-9
Exchange	A-9
Replace	A-10
Other	A-11
Quit	A-13
SECTION 3 THE TRANSLATOR	A-14
SECTION 4 THE ANALYZER	A-15
SECTION 5 THE LINE EDITOR	A-16
Insert	A-16
Delete Lines	A-17
List	A-18
Change Text	A-18
Tab Character	A-19
Set Tabs	A-19
Verify Text	A-20
End Edit	A-20
Figure 1. HLSEW Hierarchy	A-4
Figure 2. The "Command Tree"	A-5

SECTION 1. INTRODUCTION

Introduction

The High Level Software Engineering Workstation (HLSEW) includes a Translator, which converts a program written in the Program Design Language (PDL) into a compilable COBOL code program, an Analyzer that computes McCabe's and Halstead's complexity measures for the PDL program, a Line Editor and a Screen Editor for writing and editing programs. These tools make up the HLSEW and a detailed discussion of each tool is included in the following sections of this manual. Figure 1 shows the hierarchy of the System.

Most of the time you will see a "prompt line" which shows the command options available. For example:

```
COMMAND: S)creen edit  T)ranslate  A)nalyze  L)ine edit  Q)uit
```

In response to this prompt line, you can use the Line Editor, Translate, Analyze, or use the Screen Editor just by pressing a single letter. Pressing S, for example, will invoke the Screen Editor. The Screen Editor will show another prompt line that allows you to choose command options appropriate to the Screen Editor. The "Command Tree" (Figure 2) will help you find your way around the various levels of the System.

Some entries are longer than a single character. These entries are terminated by pressing the carriage return (CR) key. If you make a mistake before pressing the CR key, you can backspace over the error by pressing the backspace key (left point arrow) or CNTRL H.

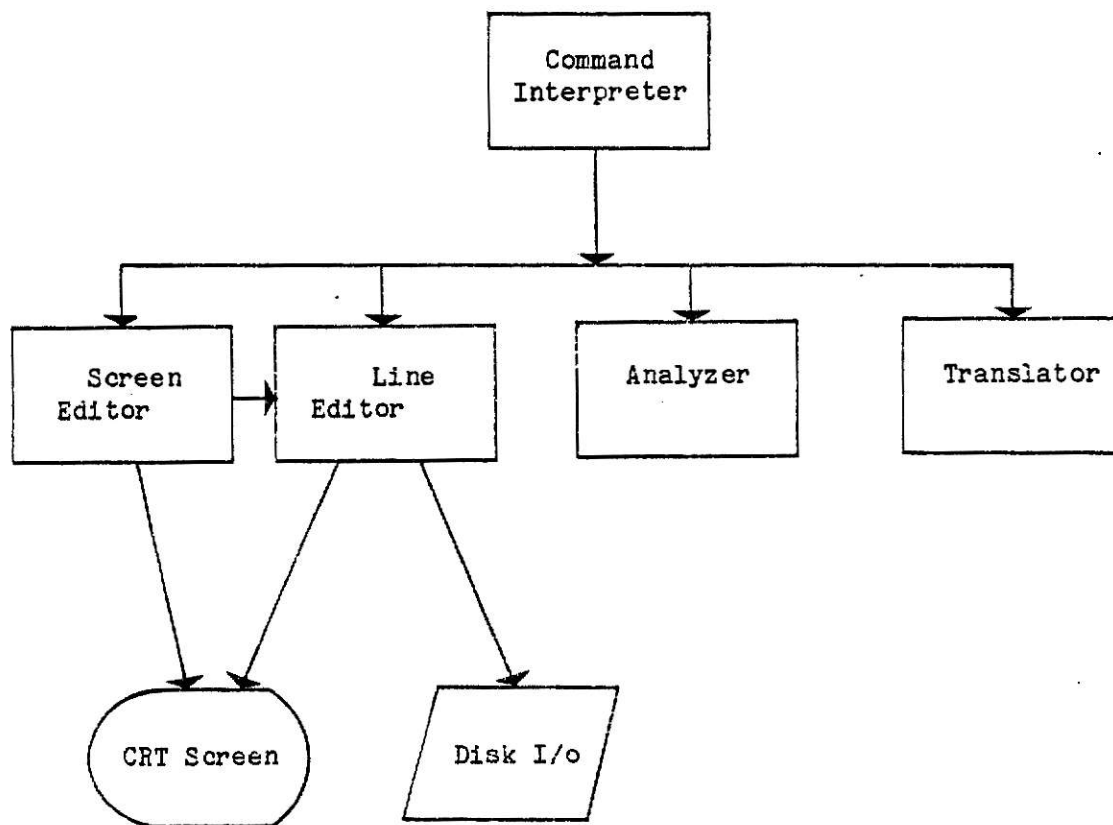


Figure 1. HLSEW Hierachy

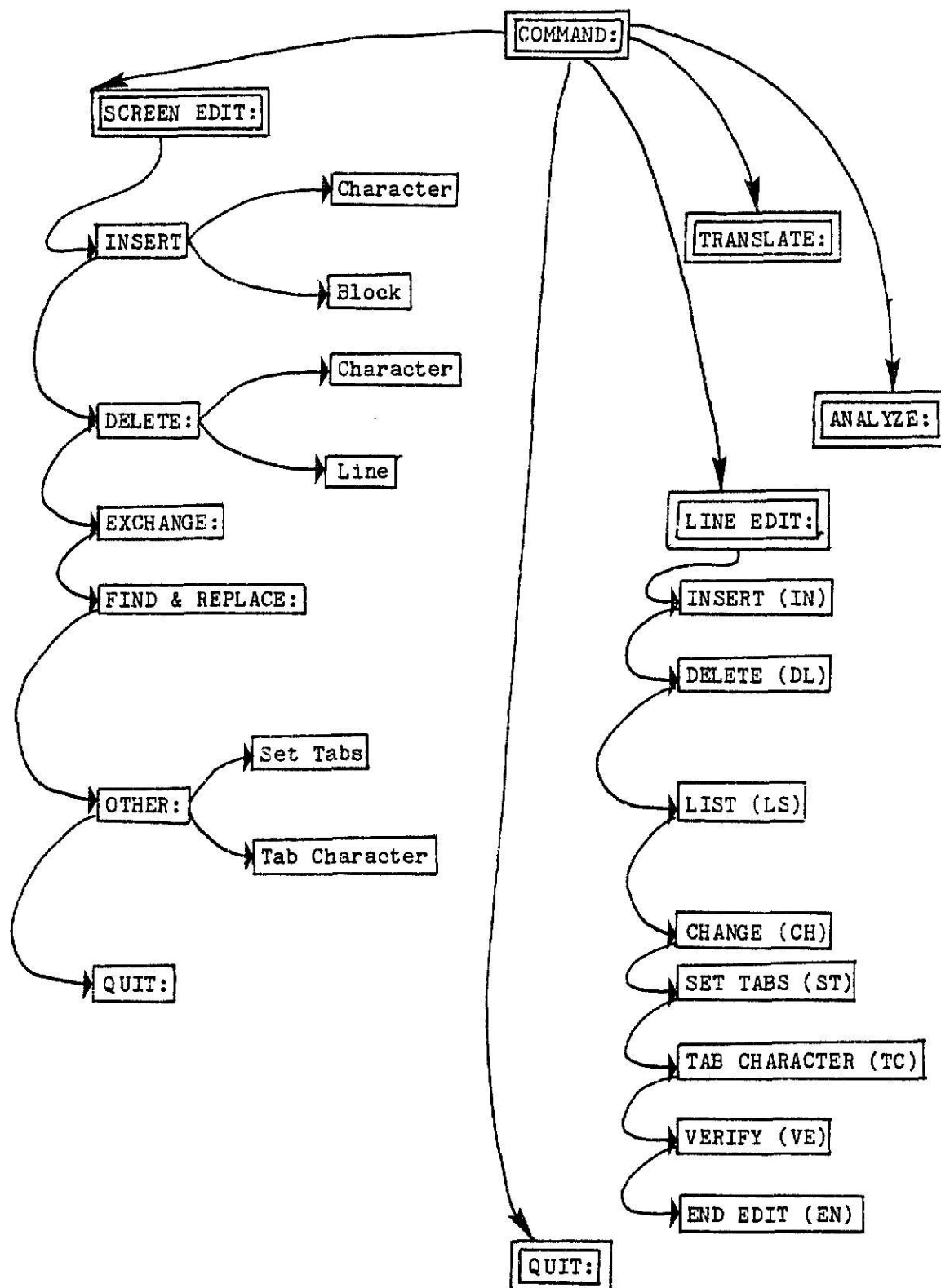


Figure 2. The Command Tree

The Command Level

The Command Level of the System acts as the "controller" for the HLSEW. It is responsible for determining which module of the System you wish to utilize.

The Command Level of the System is reached whenever you boot or reset the System (by any means), when you Q)uit either the Screen or Line Editor, and when you finish T)ranslating or A)nalyzing. The Command Level options are: S)creen edit, T)ranslate, A)nalyze, L)ine edit, or Q)uit.

The Screen Editor is specifically designed for use with displays. On entering any file, the Screen Editor displays the start of the file on the second line of the screen. If the file is too long for the screen, only the first 21 lines are displayed. This is the concept of a "window". The whole file is accessible by Screen Editor commands, but at any time only a portion of the file can be seen through the "window" of the screen. When any Screen Editor command would move to a position in the file which is not displayed, the "window" is moved to show that portion of the file.

The Screen Editor is reached by pressing S from the Command Level. You will be asked "What file?" to edit. You may either specify an existing file by typing the name of the file followed by a CR or just press CR to begin editing a new file. New files will be named at the completion of the editing session. After you press CR the Screen Editor prompt line will appear.

EDIT: I)nsert D)elete E(X)change R)eplace O)ther Q)uit

In the edit prompt lines, a word enclosed between angle brackets, <like this>, specifies that a particular key is to be pressed. For example, <CR> means that the carriage return key should be pressed, and <ESC> means to press the ESC key.

The cursor marks a position in the file and can be moved to any position on the screen. The window shows a portion of the file near the cursor. In order to edit, it is necessary to move the cursor. On the keyboard the "arrow-keys" move the cursor up, down, right, and left. You can move the cursor only when the EDIT prompt line is at the top of the screen. Vertical motion is made by using the up and down arrows. When the cursor reaches the bottom of the screen, pressing the down arrow once more will cause the next 21 lines of text to be "paged" to the screen. When the cursor reaches the top of the screen, pressing the up arrow once more will cause the prior 21 lines to be "paged" to the screen, pressing the right arrow again will cause the cursor to move to the first column of the next row. When the cursor reaches the left limits of the screen, pressing the left arrow will cause the cursor to move to the last column of the prior line.

SECTION 2. THE SCREEN EDITOR

I)nsert

The EDIT prompt line shows the command option I)nsert of which there are two submodes, C)haracter or B)lock. To insert in either mode position the cursor to the correct location and then press I. You must always move the cursor to the correct position BEFORE pressing I. After pressing I, the following prompt line will appear on the screen:

```
INSERT: C)haracter B)lock <esc> aborts
```

The C)haracter option allows you to insert information within a given text line. Pressing C will cause another prompt line to appear:

```
CHAR INSERT: <cntrl C> accepts <esc> aborts
```

The characters that you type in this mode are inserted between the character on which you placed the cursor and the character that was immediately to the cursors left. If you make a mistake while typing in I)nsert mode, just use the left arrow to delete your inserted characters. At any time during an insertion, you can cause the Screen Editor to accept the insertion as it stands and make it part of your file by pressing CNTRL C. Until you press CNTRL C, you can cause the Screen Editor to forget everything you have typed since entering I)nsert mode by pressing the ESC key.

The B)lock option allows you to insert a block of text. Pressing B will cause this prompt line to appear:

BLOCK INSERT: <cntrl C> and <CR> accepts

Upon entering the B)lock mode the cursor will be moved to column 12 of the next line and the rest of the screen erased. Column twelve is selected as a COBOL default. Text is entered normally from this point. If a mistake is made while typing in this mode, just use the left arrow to backspace over the inserted characters. At any time during an insertion you can cause the Screen Editor to accept the insertion as it stands and make it part of your file by pressing CNTRL C followed by a CR.

DELETE: <down arrow> line, <right arrow> char,
<cntrl C> accepts, <esc> aborts

BEFORE D is pressed the cursor must be in the correct position. Place the cursor directly on the first character to be deleted. Pressing the right arrow causes the character under the cursor to be deleted and all characters to the right are moved one space to the left. Notice that the cursor does not move, but appears to "eat" the text being fed from the right. Pressing the down arrow causes the entire line under the cursor to be deleted and all lines below the cursor are moved up one.

To accept the deletion at any point, press CNTRL C. To undo the entire deletion at any time before pressing CNTRL C, use ESC.

E(X)change

The E(X)change mode is reached by pressing X while at the EDIT

level. On entering the exchange mode the following prompt line appears:

EXCHANGE: <cntrl C> accepts <esc> aborts

The exchange mode is used to replace the character on which the cursor is sitting. As you type in the exchange mode, the cursor moves to the right along the line of text, replacing one character in the line each time you press a key. The left arrow key can be used to move the cursor back one character erasing any mistakes.

As with many other commands, a text exchange is made final by pressing CNTRL C. Pressing the ESC key leaves the exchange mode without making any of the changes indicated since entering the mode.

R)replace

Replace mode is reached by pressing R while at the EDIT level. On entering the replace mode the following prompt line appears:

REPLACE: delim <target> delim <substitute> delim <count>

The replace command searches through a file to find occurrences of the specified target string of characters and replaces each of those occurrences with the specified substitute string. It will do this for the number of occurrences specified by the <count>. The default is the first occurrence. An asterisk, *, replaces all occurrences of the target string. When finished, it places the cursor at the end of the last string substituted and pages the next 21 lines to the screen. An

occurrence of the target string will be found only if it appears in that portion of the text which lies between the cursor and the end of the file. If the end of the file is reached before the number of replacements specified can be carried out, this message appears:

**** STRING NOT FOUND **** press space to continue...

The replace mode has two string storage variables. The first string called <target> contains the "target string". The second string is called <substitute> and is the "substitute string". The substitute string is the sequence of characters which will replace the target string when it is found. To allow the target and substitute strings to contain any characters, each string must be typed using special rules. In particular, each string must be set off by characters called "delimiters". Both delimiters of a string must be the same character. One delimiter must precede the first character of the string and the same delimiter must follow the last character of the string. Almost any normal printing character which is not a letter or a number may be used. The most common choice is the slash (/) because it is a lower case character that is normally not found in the text and is easy to type.

O)ther

Pressing O at the EDIT level invokes the O)ther mode. On entering the O)ther mode the following prompt line appears:

OTHER: S)et tabs T)ab character R)eturn

The Other mode allows you to make the tab settings you desire, view the current tab settings, choose a tab character, or view the current tab character.

Pressing S at this point will cause this prompt to appear:

SET TABS: <cr> displays current tabs

The cursor will be positioned at the end of the prompt line. You can either press a CR, at which point the current tab settings will be displayed, or you can enter the tab settings you desire. This is accomplished by typing integers between 1 and 80 (separated by a comma or a space) and pressing the CR. The System, when initialized, defaults to standard COBOL tabs (8,12,16,24,32,36,40,56,73). To zero the current tab settings type '0' and CR. In the event that you do not enter the tab settings correctly you will get an error message like this:

** ILLEGAL COMMAND ** press space to continue...

If you see this message you will know that no changes to the tabs have been made and you must try again.

Pressing T while in the Other mode will cause this prompt line to appear:

TAB CHARACTER: <cr> displays current character

The cursor will again be positioned at the end of the prompt line. You can enter a new character followed by a CR or you can press only CR

to view the current tab character. The default tab character is '^'.

Pressing R from the Other mode returns you to the EDIT level.

Quit

Quit mode is reached by pressing Q while at the EDIT level. On entering the Quit mode your edit file will automatically be written to the disk. If it is a new file you will be asked "File name?". Enter the name followed by a CR. Should any problem occur during this operation you will be given the message:

** ERROR IN WRITING ** press space to continue...

At the completion of your edit session you will be returned to the Command Level and it's prompt line will appear.

SECTION 3. THE TRANSLATOR

To be completed at a later date.

SECTION 4. THE ANALYZER

To be completed at a later date.

SECTION 5. THE LINE EDITOR

The Line Editor is a Line Oriented Text Editor designed for use in a System having hard copy devices (e.g., teletypewriter) for terminals or for use by those who prefer Line Oriented Editors.

The Line Editor is entered from the Command Level by pressing L. You will first be asked "What file?" you wish to edit. You may either specify an existing file followed by a CR or just press the CR to begin editing a new file. After you specify the file to be edited the following message should appear:

```
HLSEW EDITOR
TYPE HELP (HE) FOR A SUMMARY OF COMMANDS
```

At any time during your edit session, typing HE will give you a summary of available commands. The Line Editor gives you the capability to perform most normal editing functions. You can insert lines of text, delete lines of text, list lines of text etc.. All Line Editor commands are two character commands followed by some set of arguments or a CR. All arguments must be separated by a comma or a space. In the following sections, arguments enclosed between angle brackets, <like this>, are mandatory. Arguments enclosed between square brackets, [like this], are optional. Commands which are not in the proper format will cause the error message:

```
** ILLEGAL COMMAND **
```

INSERT (IN)

This command allows you to insert lines into the file following the

current line. This command does not have any arguments, only a CR.

IN <CR>

When this command is issued the System is placed in the insert mode. Notice that the cursor will default to column 12 after each CR. To start before column 12 the cursor must be manually backspaced. After the desired text is entered, you can exit by typing CNTRL C followed by a CR. The cursor should then move to its normal position indicating that the insert mode has been exited.

DELETE LINES (DL)

This command deletes lines from your file. Without an argument this command deletes the current line. A line number given as an argument after DL will delete that specific line. A range of line numbers given as arguments will delete lines from argument one through argument two.

DL [LINENUMBER], [LINENUMBER] <CR>

If any line number specified is not found you will be issued the error message:

LINENUMBER ** NOT FOUND **

for each unsuccessfull attempt.

LIST (LS)

This command allows you to print lines of text to the terminal. Three arguments are valid for this command. If the argument is null (CR) then the next 23 lines of text will be printed from your file. If the argument is a valid line number, then that line becomes the current line and is printed. If the argument is a range of line numbers, then they will be printed from the first argument through the second and the last line printed will be the current line.

LS [LINENUMBER], [LINENUMBER] <CR>

Attempting to list invalid line numbers causes the error message:

LINENUMBER ** NOT FOUND **

CHANGE TEXT (CH)

This command allows an existing string to be changed to a new string, where the new string may be a null. The text file is searched from your current location to the end of the file for the target string. Once found, it is replaced by the substitute string. Three repeat options can be issued as the third argument in this command. If the third argument is omitted, then the default is one. If the third argument is an integer, then no more than that number of changes will be made between the current line and the end of the file. If the third argument is an asterisk, *, then all occurrences in the string between the current line and the end of the file are changed. A delimiter must be inserted between each argument. Any printable character not in the

string and not a letter or a number will do. The slash (/) is commonly used.

CH/<TARGET>/<SUBSTITUTE>/[REPEAT FACTOR]

Should the target string not be found you will get the error message:

** STRING NOT FOUND **

TAB CHARACTER (TC)

This command allows you to change or display the tab character. The default tab character is the up arrow, '^'. To change the tab character, simply type the new desired character as a argument to this command. A null argument (CR) displays the current tab character.

TC [TAB CHARACTER] <CR>

SET TABS (ST)

This command allows you to set or display the tab settings. If the argument is a null (CR) then the current tab settings are printed. The default tabs are the COBOL settings 8, 12, 16, 24, 32, 36, 40, 56, 73. If the argument is 0, then all tab setting positions are deleted. If the argument is a string of numbers, then these positions are added to the current tab settings. Each argument (tab setting) should be separated by a comma or a space.

ST [TABSETTING], [TABSETTING], ... [TABSETTING] <CR>

VERIFY TEXT (VE)

This command toggles a flag which controls the printing of text after a change has been made. Issuing this command with no argument will print the current state of the flag. Initial state is verify on.

VE <CR>

END EDIT (EN)

This command causes the edit session to be terminated and your edit file to be written to the disk. If this is a new file, then you will be asked to name it prior to the save. Control will be transferred to the Command Level.

APPENDIX B

SCREEN EDITOR

SOURCE CODE

APPENDIX B SCREEN EDITOR SOURCE CODE

```

(*****
*
*
*      DDDDDDD      EEEEEEE      DDDDDDD      IIIIII      TTTTTTTT
*      DDDDDDDDD      EEEEEEE      DDDDDDDDD      IIIIII      TTTTTTTT
*      DD  DD      EE      DD  DD      II      T  TT  T
*      DD  DD      EE      DD  DD      II      TT
*      DD  DD      EEEEEE      DD  DD      II      TT
*      DD  DD      EEEEEE      DD  DD      II      TT
*      DD  DD      EE      DD  DD      II      TT
*      DD  DD      EE      DD  DD      II      TT
*      DDDDDDDDD OO      EEEEEEE      DDDDDDDDD      IIIIII      TT
*      DDDDDDDDD OO      EEEEEEE      DDDDDDDDD      IIIIII      TTTT
*
*
*      INTERACTIVE SCREEN EDITOR DESIGNED BY DAVID B. AARONSON
*      IN PARTIAL FULFILLMENT OF A MASTERS OF SCIENCE IN CS.
*      FALL 1981  VERSION 1  KANSAS STATE UNIVERSITY
*
*
*****)

```

```

PROGRAM HLSEW;
(*$U EDITOR.CODE*)
USES EDITOR;      (* calls line editor unit *)

```

```

PROCEDURE SCREEN_EDIT; FORWARD;
PROCEDURE MOVE_RIGHT; FORWARD;
PROCEDURE MOVE_LEFT; FORWARD;
PROCEDURE EDITOR_MENU; FORWARD;
PROCEDURE EDITOR_COMMAND_INTERPRETER; FORWARD;
PROCEDURE COMMAND_INTERPRETER; FORWARD;

```

```

(*****
PROCEDURE CLEAR_SCREEN;
(*****
Clears entire screen of all text.
*****
)

```

```

BEGIN
  WRITE (CHR(27)); WRITELN (CHR(43));
END;  (* CLEAR SCREEN *)

```

```

(*****
PROCEDURE BELL;
(*****
Sounds bell when called.
*****
)

```

```

BEGIN
  WRITE (CHR(7));
END;  (* BELL *)

```

```

(*****)
PROCEDURE ERASE_REST OF PAGE;
(*****)
Erases remainder of page from current cursor position.
(*****)

BEGIN
    WRITE (CHR(27)); WRITE (CHR(121));
END; (* ERASE REST OF PAGE *)

```

```

(*****)
PROCEDURE CONVERT_INTEGER (INT_IN : INTEGER);
(*****)
This procedure converts integers into characters and inserts them
into an array of characters for processing by the line editor. An
integer is parsed into individual digits using the DIV and MOD
functions and these digits are converted to characters and inserted
into the array.
(*****)

```

```

VAR I, INDEX, NUMBER : INTEGER;
    TEMP : ARRAY [1..5] OF CHAR;

```

```

BEGIN
    NUMBER := INT_IN;
    INDEX := 0;
    FOR I := 1 TO 5 DO
        TEMP[I] := ' ';
    REPEAT
        INDEX := SUCC(INDEX);
        TEMP[INDEX] := CHR(NUMBER MOD 10 + ORD ('0'));
        NUMBER := NUMBER DIV 10
    UNTIL NUMBER = 0;
    FOR I := INDEX DOWNTO 1 DO
        IF TEMP[I] <> ' ' THEN
            BEGIN
                INPUT_LINE[LINE_INDEX] := TEMP[I];
                LINE_INDEX := SUCC(LINE_INDEX);
            END;
        END;
    END;
END; (* CONVERT INTEGER *)

```

```

(*****
PROCEDURE PAGE;
(*****
List 20 lines of text to the screen by calling the line editor
procedure LIST_IT.
*****)

BEGIN
  GOTOXY (0,2);
  LINE_INDEX := 3;
  INPUT_LINE [LINE_INDEX] := NL;
  LINE_EDIT(LIST);
  GOTOXY (0,2);
  ROW := 2; COLUMN := 0;
END;  (* PAGE *)

(*****
PROCEDURE PAGE_BACK;
(*****
List the prior 23 lines of text to the screen by calling the line
editor procedure LIST_IT.
*****)

BEGIN
  LINE_NUMBER := LINE_NUMBER - 20;
  IF LINE_NUMBER < 1 THEN LINE_NUMBER := 1;
  GOTOXY (0,2);
  LINE_INDEX := 3;
  INPUT_LINE [LINE_INDEX] := NL;
  LINE_EDIT(LIST);
  GOTOXY (COLUMN,22);
  LINE_NUMBER := LINE_NUMBER + 20;
END;  (* PAGE BACK *)

```

```

(*****)
PROCEDURE READ_SCREEN_COMMAND;
(*****
Interpretes commands given to the screen editor and converts them to
the format recognized by the line editor. All commands are placed in
a data stucture called input line which is then parsed by the line
editor.
(*****)

VAR C : CHAR;
    I : INTEGER;
    ALL_DONE, ESCAPE : BOOLEAN;
    VALID_SET_OF_CHAR : SET_OF_VALID;

BEGIN
    VALID_SET_OF_CHAR := [CHR(1)..CHR(9),CHR(13)..CHR(127)];
    INPUT_LINE[3] := ' ';
    LINE_INDEX := 4;
    GET_CHARACTER(VALID_SET_OF_CHAR,C,ALL_DONE.ESCAPE);
    INPUT_LINE [LINE_INDEX] := C;
    WRITE (C);
    WHILE (NOT EOLN (KEYBOARD)) AND (LINE_INDEX <> LINELENGTH)
        AND (NOT ESCAPE) DO
        BEGIN
            GET_CHARACTER(VALID_SET_OF_CHAR,C,ALL_DONE.ESCAPE);
            LINE_INDEX := SUCC(LINE_INDEX);
            IF C = CHR(8) (* backspace *)
                THEN BEGIN
                    LINE_INDEX := PRED (LINE_INDEX -1);
                    IF LINE_INDEX <= 2 THEN LINE_INDEX := 2;
                END
            ELSE BEGIN
                WRITE(C);
                INPUT_LINE[LINE_INDEX] := C;
            END
        END; (* while *)
    READLN (KEYBOARD);
    WRITELN;
    IF ESCAPE THEN BEGIN
        EDITOR_MENU;
        EXIT (EDITOR_COMMAND_INTERPRETER);
    END;
    FOR I := LINE_INDEX TO LINE_LENGTH DO
        INPUT_LINE[I] := CHR(13);
    LINE_INDEX := 3;
END; (* READ SCREEN COMMAND *)

```

```

(*****)
PROCEDURE SCREEN_INSERT;
(*****)
This procedure allows the user to insert text into his file. It
can be done on a character by character basis within a line or on
a block basis. If inserting a block the screen is erased from the
cursor position and text is entered normally utilizing the line editor.
If entering characters then each time a character is entered from
the keyboard all other characters to the right of the cursor are
moved one space and the character is entered under the cursor. Both
modes are exited by CNTL C followed by a CR.
(*****)

VAR CH : CHAR;
    ALL_DONE, ESCAPE, FOUND : BOOLEAN;
    VALID_SET_OF_CHAR : SET_OF VALID;
    I : INTEGER;

BEGIN
    ND_SCREEN_PUT('INSERT: C)haracters B)lock <esc> aborts');
    VALID_SET_OF_CHAR := ['C', 'B', CHR(27)];
    GET_CHARACTER(VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
    CASE CH OF
        'C' : BEGIN
            VALID_SET_OF_CHAR := [CHR(1)..CHR(9),CHR(13)..CHR(127)];
            ND_SCREEN_PUT('CHAR INSERT: <cntrl C> accepts <esc> aborts');
            GOTOXY(COLUMN,ROW);
            GET_CHARACTER(VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
            WHILE (NOT ALL_DONE) AND (NOT ESCAPE) DO BEGIN
                IF COLUMN = 79 THEN BELL
                ELSE
                    IF CH = CHR(8) THEN MOVE_LEFT
                    ELSE BEGIN
                        FOR I := (COLUMN + 1) TO (LINE_LENGTH - 1) DO
                            TEMP_LINE[I + 1] := TEMP_LINE[I];
                        WRITE(CHR(27)); WRITE(CHR(81)); (* insert *)
                        TEMP_LINE[COLUMN + 1] := CH;
                        WRITE (CH);
                        MOVE_RIGHT;
                    END; (* else *)
                GET_CHARACTER(VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
            END; (* while *)
            IF ALL_DONE THEN LINE_EDIT(STORE_IT);
            IF ESCAPE THEN
                BEGIN
                    LINE_EDIT(FETCH_IT);
                    GOTOXY (0,ROW); (* beginning of row *)
                    PRINT_TEMP_LINE;
                END;
            END; (* W *)
        'B' : BEGIN
            ND_SCREEN_PUT('BLOCK INSERT: <cntrl C> and <CR> accepts');
            GOTOXY (0,ROW + 1);
            ERASE_REST_OF_PAGE;
            LINE_INDEX := 3;
            INPUT_LINE[LINE_INDEX] := NL;
            LINE_EDIT(INSERT);
            LINE_NUMBER := LINE_NUMBER - 10;
            PAGE;
        END; (* B *)
    END; (* Case *)
    IF ESCAPE THEN EXIT(SCREEN_INSERT);
END; (* Screen Insert *)

```

```

(*****)
PROCEDURE SCREEN_DELETE;
(*****)
This procedure allows the user to delete either entire lines or
characters within a line by using the cursor arrows. The down arrow
deletes the current line and the right arrow deletes the character
under the cursor. Once it is determined how many lines or characters
are to be deleted, a call to the line editor is made to do the
deletion.
(*****)

VAR CH : CHAR;
    ALL_DONE, ESCAPE, FOUND : BOOLEAN;
    VALID_SET_OF_CHAR : SET_OF VALID;
    I, LAST_LINE, FIRST_LINE : INTEGER;

BEGIN
    ND_SCREEN_PUT
    ('DELETE: <down arrow> line, <right arrow> char, <Ctrl C> accepts');
    WRITE(' ', <esc> aborts');
    GOTOXY (COLUMN, ROW);
    VALID_SET_OF_CHAR := [ CHR(10), CHR(12), CHR(27), CHR(3)];
    FIRST_LINE := LINE_NUMBER;
    GET_CHARACTER(VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
    IF CH = CHR(10) THEN
        BEGIN
            WHILE (NOT ALL_DONE) AND (NOT ESCAPE) DO
                BEGIN
                    VALID_SET_OF_CHAR := [ CHR(10), CHR(27), CHR(3)];
                    LINE_NUMBER := SUCC(LINE_NUMBER);
                    COLUMN := 0;
                    WRITE(CHR(27)); WRITE(CHR(82)); (* Delete line *)
                    GET_CHARACTER(VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
                END; (* while *)
            IF ESCAPE THEN PAGE;
            END; (* if chr(10) *)
        IF CH = CHR(12) THEN
            BEGIN
                WHILE (NOT ALL_DONE) AND (NOT ESCAPE) DO
                    BEGIN
                        VALID_SET_OF_CHAR := [ CHR(12), CHR(3), CHR(27)];
                        LINE_EDIT(FETCH_IT);
                        FOR I := (COLUMN + 1) TO (LINELENGTH - 1) DO
                            TEMP_LINE [I] := TEMP_LINE [I + 1];
                        WRITE(CHR(27)); WRITE(CHR(87)); (* Delete character *)
                        GET_CHARACTER(VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
                    END; (* while *)
                IF ESCAPE THEN
                    BEGIN
                        LINE_EDIT (FETCH_IT);
                        GOTOXY (0,ROW); (* beginning of row *)
                        PRINT_TEMP_LINE
                    END; (* if escape *)
                END; (* if chr (12) *)
            END;
        END;
    END;

```

```

IF ALL DONE THEN BEGIN
  LAST_LINE := LINE_NUMBER - 1;
  IF FIRST_LINE = LAST_LINE + 1
  THEN LINE_EDIT(STORE_IT)
  ELSE BEGIN
    LINE_INDEX := 4;
    CONVERT_INTEGER(FIRST_LINE);
    LINE_INDEX := SUCC(LINE_INDEX);
    CONVERT_INTEGER(LAST_LINE);
    FOR I := (LINE_INDEX + 1) TO LINE_LENGTH DO
      INPUT_LINE[I] := NL;
    LINE_EDIT(DELETE);
    LINE_NUMBER := LINE_NUMBER - 11;
    PAGE;
  END; (* else *)
END; (* if all done *)
END; (* DELETE *)

```

```

(*****
PROCEDURE EXCHANGE;
(*****
This procedure allow the user to make literal changes within a line.
It brings into the line buffer the line associated with the cursor
position and takes input from the keyboard on a character by character
basis. If the exchange is desired it stores the line in its new format.
*****

```

```

VAR CH : CHAR;
    ALL_DONE, ESCAPE, FOUND : BOOLEAN;
    VALID_SET_OF CHAR : SET_OF VALID;

```

```

BEGIN
  LINE_EDIT(FETCH_IT);
  ND SCREEN PUT('EXCHANGE: <esc> to abort <cntrl C> to accept');
  GOTOXY (COLUMN,ROW); (* returns cursor to original position *)
  VALID_SET_OF CHAR := [CHR(3),CHR(27),CHR(8),CHR(32)..CHR(127)];
  GET_CHARACTER(VALID_SET_OF CHAR, CH, ALL_DONE, ESCAPE);
  WHILE (NOT ALL_DONE) AND (NOT ESCAPE) DO
    BEGIN
      IF COLUMN = 79 THEN BELL
      ELSE
        IF CH = CHR(8)
        THEN MOVE_LEFT
        ELSE BEGIN
          TEMP_LINE[COLUMN + 1] := CH;
          WRITE (CH);
          MOVE_RIGHT;
        END;
      GET_CHARACTER(VALID_SET_OF CHAR, CH, ALL_DONE, ESCAPE);
    END; (* while *)
  IF ALL_DONE THEN
    LINE_EDIT(STORE_IT);
  IF ESCAPE THEN
    BEGIN
      LINE_EDIT(FETCH_IT);
      GOTOXY(0,ROW); (* Beginning of row *)
      PRINT_TEMP_LINE
    END
  END; (* EXCHANGE *)

```



```

(*****)
PROCEDURE REPLACE;
(*****
This procedure expects to get a target string and a proposed change
to that string. It then call the line editor and passes the proposed
change. Upon completion it pages text to the screen starting at the
line number where the change was made.
*****)

BEGIN
  ND_SCREEN_PUT
    ('REPLACE: delim <target> delim <substitute> delim [ repeat factor]');
  GOTOXY (0,1);
  READ_SCREEN_COMMAND;
  ERASE_REST_OF_PAGE;
  LINE_EDIT (CHANGE);
  ND_SCREEN_PUT(' Press space bar to continue....');
  READ (INPUT, SPACE_BAR);
  GOTOXY (0,1);
  CLEAR TO_EOLN;
  PAGE;
END;  (* REPLACE *)

(*****)
PROCEDURE SET_THE_TABS;
(*****
Calls line editor procedure to set the tabs.
*****)

BEGIN
  ND_SCREEN_PUT ('SET TABS: <cr> displays current tabs');
  READ_SCREEN_COMMAND;
  LINE_EDIT(SETTABS);
END;  (* SET THE TABS *)

(*****)
PROCEDURE ESTABLISH_TAB_CHAR;
(*****
Displays menu and calls the line editor procedure the establish the
tab character.
*****)

BEGIN
  ND_SCREEN_PUT ('SET TAB CHARACTER: <cr> displays current character');
  READ_SCREEN_COMMAND;
  LINE_EDIT(TABCHAR);
END;  (* ESTABLISH TAB CHAR *)

```

```

(*****)
PROCEDURE OTHER;
(*****)
Interprets secondary menu of additional functions that can be
performed by the line editor and makes calls to appropriate screen
handler procedures.
(*****)

```

```
VAR CH : CHAR;
```

```

BEGIN
  ND_SCREEN.PUT('OTHER: S)et tabs T)ab character R)eturn');
  REPEAT READ (KEYBOARD, CH) UNTIL CH IN [ 'S', 'T', 'R' ];
  CASE CH OF
    'S' : SET_THE_TABS;
    'T' : ESTABLISH_TAB_CHAR;
    'R' : EXIT(OTHER);
  END; (* CASE *)
END; (* OTHER *)

```

```

(*****)
PROCEDURE MOVE_UP;
(*****)
Keeps track of the row and current line number when scrolling through
a file.
(*****)

```

```

BEGIN
  ROW := PRED(ROW);
  GOTOXY (COLUMN, ROW);
  LINE_NUMBER := PRED (LINE_NUMBER);
  IF ROW < 2 THEN
    BEGIN
      ROW := 22;
      PAGE_BACK;
    END
  END; (* MOVE UP *)

```

```

(*****)
PROCEDURE MOVE_DOWN;
(*****)
Keeps track of row number and line number when scrolling forward in
a file.
(*****)

```

```

BEGIN
  ROW := SUCC(ROW);
  LINE_NUMBER := SUCC (LINE_NUMBER);
  GOTOXY (COLUMN, ROW);
  IF ROW > 22 THEN PAGE
END; (* MOVE DOWN *)

```

```

(*****)
PROCEDURE MOVE_RIGHT;
(*****)
Keeps track of column while moving through row.
(*****)

```

```

BEGIN
    COLUMN := SUCC (COLUMN);
    IF COLUMN = 75 THEN BELL;
    IF COLUMN > 79 THEN
        BEGIN
            COLUMN := 0;
            MOVE_DOWN
        END;
    GOTOXY (COLUMN, ROW);
END; (* MOVE RIGHT *)

```

```

(*****)
PROCEDURE MOVE_LEFT;
(*****)
Keeps track of column when moving through a row.
(*****)

```

```

BEGIN
    COLUMN := PRED(COLUMN);
    IF COLUMN < 0 THEN
        BEGIN
            COLUMN := 79;
            MOVE_UP;
        END;
    GOTOXY (COLUMN, ROW);
END; (* MOVE LEFT *)

```

```

(*****)
PROCEDURE EDITOR_MENU ;
(*****)
Displays editor menu on screen.
(*****)

```

```

BEGIN
    ND_SCREEN_PUT
    ('EDIT: I)nsert D)elete E(X)change R)eplace O)ther Q)uit');
END; (* EDITOR_COMMAND_MENU *)

```

```

(*****
PROCEDURE EDITOR_COMMAND_INTERPRETER ;
(*****
Interprets commands from the keyboard within a legal set of values.
Handles movement of the cursor arrows and the screen editor menu
items.
*****)

VAR INPUT : CHAR;

BEGIN
  REPEAT READ(KEYBOARD, INPUT)
    UNTIL INPUT IN
      ['I', 'D', 'R', 'Q', 'X', 'O', CHR(10), CHR(11), CHR(12), CHR(8)];
  IF INPUT = CHR(10) THEN MOVE_DOWN;
  IF INPUT = CHR(11) THEN MOVE_UP ;
  IF INPUT = CHR(8) THEN MOVE_LEFT;
  IF INPUT = CHR(12) THEN MOVE_RIGHT;
  CASE INPUT OF
    'I' : SCREEN_INSERT;
    'D' : SCREEN_DELETE;
    'X' : EXCHANGE;
    'R' : REPLACE;
    'O' : OTHER;
    'Q' : BEGIN
      CLEAR SCREEN; EXIT (SCREEN_EDIT);
    END;
  END; (* CASE *)
  IF INPUT IN ['I', 'X', 'D', 'R', 'O']
  THEN BEGIN
    EDITOR_MENU;
    GOTOXY (COLUMN, ROW)
  END;
END; (* EDITOR_COMMAND INTERPRETER *)

(*****
PROCEDURE SET_UP_SCREEN_EDITOR;
(*****
Initializes the Screen Editor.
*****)

BEGIN
  TABS := [8, 12, 16, 24, 32, 36, 40, 56, 73];
  TAB_CHARACTER := '^';
  LINE_NUMBER := 1;
  NL := CHR(13);
  COMMAND := BADCOMMAND;
  EDITING_FROM_SCREEN := TRUE;
  CLEAR SCREEN;
  EDITOR_MENU;
  LINE_INDEX := 3;
  PAGE;
END; (* SET_UP_SCREEN_EDITOR *)

```

```

(*****)
PROCEDURE SCREEN_EDIT;
(*****
Functions as the driver procedure for the Screen Editor. It makes
repeated calls on the editor command interpreter until the user is
finished.
*****)

BEGIN
    SET_UP_SCREEN_EDITOR;
    EDITOR_COMMAND INTERPRETER;
    REPEAT
        EDITOR_COMMAND INTERPRETER;
    UNTIL FINISHED;
END; (* EDITOR *)

(*****)
PROCEDURE OUTER_COMMAND_MENU ;
(*****
Displays the outer command menu for the HLSEW Command Interpreter.
*****)

BEGIN
    ND_SCREEN_PUT
    ('COMMAND: S)screen edit T)ranslate A)nalyze Q)uit L)ine edit')
END; (* OUTER_COMMAND_MENU *)

(*****)
PROCEDURE GET_FILE;
(*****
Procedure establishes the file that the user wants and passes the
name to the appropriate procedure.
*****)

VAR FOUND : BOOLEAN;
    FILE_NAME : STRING;

BEGIN
    ND_SCREEN PUT (' What file ?');
    READLN (FILE_NAME);
    READ_FILE (FOUND, FILE_NAME);
    IF NOT FOUND THEN
        BEGIN
            ERROR(NOT_FOUND);
            EXIT(COMMAND_INTERPRETER);
        END;
END; (* GET FILE *)

```

```

(*****)
PROCEDURE COMMAND_INTERPRETER ;
(*****)
Interprets commands for the HLSEW system. Accepts only a limited
set of input from the keyboard and calls the appropriate module.
(*****)

VAR CH : CHAR;
    OK : BOOLEAN;

BEGIN
    REPEAT
        READ (KEYBOARD,CH)
        UNTIL CH IN ['A', 'T', 'S', 'Q', 'L'];
    CASE CH OF
        'S' : BEGIN
            GET_FILE ;
            SCREEN EDIT;
            WRITE_FILE (OK);
            IF NOT OK THEN
                ERROR(WRITING);
            END;
        'T' : BEGIN
            GET_FILE;
            TRANSLATOR (OK);
            IF NOT OK THEN
                ERROR(TRANS_ERROR)
            END;
        'A' : BEGIN
            GET_FILE;
            ANALYZER(OK);
            IF NOT OK THEN
                ERROR(UPDATING)
            END;
        'L' : BEGIN
            GET_FILE;
            CLEAR SCREEN;
            LINE_EDIT(EDIT_IT);
            WRITE_FILE(OK);
            IF NOT OK THEN
                ERROR(WRITING);
            END;
        'Q' : BEGIN
            CLEAR SCREEN;
            GOTOXY (25,12);
            WRITELN ('END HLSEW SESSION');
            EXIT (PROGRAM);
        END;
    END (*CASE*)
END; (* COMMAND INTERPRETER*)

```

```
(*****)
PROCEDURE INITIALIZE ;
(*****
Initializes the HLSEW system.
*****)
```

```
BEGIN
    EDITING_FROM_SCREEN := TRUE;
    CLEAR_SCREEN;
    GOTOXY(25,12);
    WRITELN ('WELCOME TO HLSEW');
    GOTOXY(22,50);
    WRITE('copywrite pending @1981');
    OUTER_COMMAND_MENU;
    NL := CHR(13);
    FINISHED := FALSE;
    VERIFY_CHANGES := TRUE;
END; (* INITIALIZE *)
```

```
(*****
                                MAIN PROGRAM
*****
Drives the outer command menu and the command interpreter of the
HLSEW system until the user is finished.
*****)
```

```
BEGIN
    INITIALIZE;
    COMMAND_INTERPRETER;
    REPEAT
        OUTER_COMMAND_MENU;
        COMMAND_INTERPRETER;
    UNTIL FINISHED
END. (* HLSEW *)
```

APPENDIX C

LINE EDITOR

SOURCE CODE

APPENDIX C LINE EDITOR SOURCE CODE

```

(*****
*
*      DDDDDDD      EEEEEEE      DDDDDDD      IIIIII      TTTTTTTT
*      DDDDDDDD      EEEEEEE      DDDDDDDD      IIIIII      TTTTTTTT
*      DD   DD      EE          DD   DD      II      T   TT   T
*      DD   DD      EE          DD   DD      II      TT
*      DD   DD      EEEEEEE      DD   DD      II      TT
*      DD   DD      EE          DD   DD      II      TT
*      DD   DD      EE          DD   DD      II      TT
*      DDDDDDDD OO      EEEEEEE      DDDDDDDD      IIIIII      TT
*      DDDDDDD  OO      EEEEEEE      DDDDDDD      IIIIII      TTTT
*
*
*      LINE EDITOR UNIT DESIGNED BY DAVID B. AARONSON IN
*      PARTIAL FULFILLMENT OF A MASTERS OF SCIENCE IN CS.
*      FALL 1981   VERSION 1   KANSAS STATE UNIVERSITY
*
*****)

```

```
UNIT EDITOR;
```

```

(*****)
INTERFACE
(*****)

```

```

CONST
  LINELENGTH = 80;
  LOG_ON_MSG = ' HLSEW EDITOR';
  HELP_MSG = 'TYPE HELP (HE) FOR A SUMMARY OF COMMANDS';

```

```

TYPE
  LINE = ARRAY [1..LINELENGTH] OF CHAR;
  TABSETTING = SET OF 1..80;
  COMMAND_TYPE = ( INSERT, DELETE, CHANGE, SETTABS, TABCHAR, VERIFY, LIST,
    FETCH_IT, STORE_IT, REPEAT_IT, HELP, ENEDIT,
    APPEND, BADCOMMAND, EDIT_IT);
  ERROR_TYPE = (COMMAND_ERROR, NOT_FOUND, STRING_NOT_FOUND, WRITING,
    LONGLINE, OTHER_ERROR, CHAR_ERROR, TRANS_ERROR, UPDATING);
  TOKEN_TYPE = (NILTOK, LINENOTOK, .OTHERTOK);
  TOKEN = RECORD
    TOKEN_KIND : TOKEN_TYPE;
    VALUE : INTEGER
  END;
  MODE_TYPE = (DELETE_MODE, CHANGE_MODE, INSERT_MODE);
  SET_OF_VALID = SET OF CHAR;

```

```

VAR
  COMMAND : COMMAND_TYPE;
  INPUT_LINE, TEMP_LINE : LINE;
  LINE_INDEX, TEMP_LENGTH, LINE_NUMBER : INTEGER;
  VERIFY_CHANGES, FINISHED, EDITING_FROM_SCREEN : BOOLEAN;
  TAB_CHARACTER, NL, SPACE_BAR : CHAR;
  TABS : TABSETTING;
  ROW, COLUMN, SAVE_ROW, SAVE_COLUMN : INTEGER;

(*****
  INTERFACE PROCEDURES
  *****)
  Those procedures that are used by both the line editor and the screen
  editor.
  *****)
PROCEDURE STORE_A_LINE (LINE_IN : LINE; LINE_NO : INTEGER; MODE_IN :
  MODE_TYPE);
PROCEDURE EXIT_STORE_A_LINE;
PROCEDURE DELETE_A_LINE (LINE_NO : INTEGER; VAR FOUND_IT : BOOLEAN);
PROCEDURE FETCH_A_LINE (INPUT_LINE_NO : INTEGER; VAR LINE_OUT : LINE;
  FOUND_IT : BOOLEAN);
PROCEDURE ANALYZER (VAR OK : BOOLEAN);
PROCEDURE TRANSLATOR (VAR OK : BOOLEAN);
PROCEDURE BUILD_A_LINE (VAR DONE : BOOLEAN);
PROCEDURE FIND_TOKEN (VAR NEXT_TOKEN : TOKEN);
PROCEDURE FIND_STRING (STRING : LINE; STRING_LENGTH : INTEGER;
  VAR START : INTEGER; VAR FOUND : BOOLEAN);
PROCEDURE FETCH_CURRENT_LINE (INPUT_LINE_NO : INTEGER;
  VAR LINE_FOUND : BOOLEAN);
PROCEDURE GET_CHARACTER (LEGAL_CHAR : SET_OF_VALID; VAR CH_OUT : CHAR;
  VAR ALL_DONE, ESCAPE : BOOLEAN);
PROCEDURE PRINT_TEMP_LINE;
PROCEDURE STORE_CURRENT_LINE (MODE : MODE_TYPE);
PROCEDURE CHANGE_TEXT;
PROCEDURE MAKE_TAB_SETTING;
PROCEDURE SET_TAB_CHAR;
PROCEDURE INSERT_TEXT;
PROCEDURE DELETE_LINES;
PROCEDURE LIST_IT;
PROCEDURE LINE_EDIT (COMMAND_IN : COMMAND_TYPE);
PROCEDURE ND_SCREEN_PUT (INPUT_STRING : STRING);
PROCEDURE CLEAR_TO_EOLN;
PROCEDURE READ_FILE (VAR FOUND : BOOLEAN; FILE_NAME : STRING);
PROCEDURE WRITE_FILE (VAR OK : BOOLEAN);
PROCEDURE ERROR (ERROR_KIND : ERROR_TYPE);

(*****
  IMPLEMENTATION
  *****)

```

```

(*****)
PROCEDURE HELP_IT;
(*****
This procedure provides the user with a summary of the commands
available to him.
*****)

BEGIN;
  WRITELN ('THE FOLLOWING IS A SUMMARY OF THE EDITOR COMMANDS');
  WRITELN;WRITELN;WRITELN
  ('"IN" PROVIDES FOR THE INSERTION OF TEXT STARTING AT THE CURRENT');
  WRITELN
  ('LINE NUMBER. TYPE CNTRL "C" AND A <CR> TO EXIT THE INSERT MODE. ');
  WRITELN;WRITELN;WRITELN
  ('"DL" DELETES LINES FROM YOUR FILE. IT MUST BE FOLLOWED BY A ');
  WRITELN
  ('LINENUMBER OR A RANGE OF LINENUMBERS. E.G. DL ARG, {ARG}');
  WRITELN;WRITELN;WRITELN
  ('"CH" CHANGES AN EXISTING STRING WITH A NEW STRING AT THE CURRENT');
  WRITELN
  ('LINENUMBER. E.G CH/OLD TEXT/NEW TEXT/ {ARG} THE THIRD ARGUMENT');
  WRITELN
  ('HAS THREE OPTIONS. IT MAY BE NULL (DEFAULT IS 1), IT MAY BE AN');
  WRITELN
  ('INTEGER SPECIFYING THE NUMBER OF OCCURANCES TO CHANGE, OR IT MAY');
  WRITELN
  ('BE AN "*" WHICH WILL MAKE THE SPECIFIED CHANGES THROUGHOUT THE');
  WRITELN
  ('THE ENTIRE FILE');
  WRITELN;
  WRITELN ('Press space bar to continue.....');
  READ (INPUT, SPACE_BAR);
  WRITELN;WRITELN;WRITELN;
  WRITELN
  ('"LS" LIST THE LINENUMBER REQUESTED. WHEN FOLLOWED BY A SECOND');
  WRITELN
  ('ARGUMENT A RANGE OF LINES ARE DISPLAYED. E.G. LS ARG, {ARG}');
  WRITELN;WRITELN;WRITELN
  ('"TC" ALLOWS YOU TO SET THE TAB CHARACTER TO ANY CHARACTER OTHER');
  WRITELN
  ('THAN A NUMBER OR LETTER. TC FOLLOWED BY A CR WILL DISPLAY CURRENT');
  WRITELN
  ('TAB CHARACTER. E.G TC {ARG}');
  WRITELN;WRITELN;WRITELN
  ('"ST" ALLOWS YOU TO SET TABS. THIS COMMAND FOLLOWED BY A CR WILL');
  WRITELN
  ('DISPLAY THE CURRENT TAB SETTINGS. OTHERWISE THIS COMMAND SHOULD');
  WRITELN
  ('FOLLOWED BY THE TAB SETTING ARGUMENTS YOU DESIRED SEPARATED BY');
  WRITELN
  ('A COMMA OR A SPACE. E.G ST {ARG} {ARG} {ARG} OR {ARG},{ARG} ETC');
  WRITELN
  ('THIS COMMAND FOLLOWED BY "C" WILL SET THE STANDARD COBOL TABS');
  WRITELN;WRITELN;WRITELN
  ('"EN" WILL EXIT THE EDIT MODE');
  WRITELN;WRITELN;WRITELN
  ('"VE" TOGGLES VERIFY ON AND OFF. UPON EXECUTION YOU WILL BE NOTIFIED');
  WRITELN
  ('OF THE CURRENT SELECTION');
END;

```

```

( ***** )
PROCEDURE ND_SCREEN_PUT (INPUT_STRING : STRING);
( ***** )
Non destructive write routine which places lines on the top line
of the screen.
( ***** )

BEGIN
    GOTOXY (0,0);
    CLEAR TO_EOLN;
    GOTOXY (0,0);
    WRITE (INPUT_STRING);
END;    (* ND SCREEN PUT *)

( ***** )
PROCEDURE CLEAR TO_EOLN;
( ***** )
Erases screen to end of current line.
( ***** )

BEGIN
    WRITE(CHR(27)); WRITE(CHR(84));
END;    (* CLEAR TO EOLN *)

( ***** )
PROCEDURE GET_CHARACTER (LEGAL_CHAR : SET_OF_VALID; VAR CHAR OUT :
                        CHAR; VAR ALL_DONE. ESCAPE : BOOLEAN);
( ***** )
I/O procedure which receives as input a set of valid characters which
are acceptable. It then reads until one of those characters is found
and returns it to the calling routine. Special cases such as the ESC
key and CNTRL C set flags which are returned also.
( ***** )

VAR CH : CHAR;

BEGIN
    ESCAPE := FALSE;
    ALL_DONE := FALSE;
    REPEAT READ (KEYBOARD, CH) UNTIL CH IN LEGAL_CHAR;
    IF CH = CHR(8) THEN BEGIN
        WRITE(CHR(8)); WRITE(' '); WRITE(CHR(8)); END; (* destroys char *)
    IF CH = CHR(3) THEN ALL_DONE := TRUE;
    IF CH = CHR(27) THEN ESCAPE := TRUE;
    CH_OUT := CH
END;    (* GET CHARACTER *)

```

```

(*****
PROCEDURE ERROR (ERROR_KIND : ERROR_TYPE);
(*****
Procedure Error is a centralized means of providing error messages.
If the user is in screen edit mode the the error messages are written
to the top of the screen so as not to destroy any text on the screen.
If the user is in the line edit mode, then the messages are written
normally to the screen.
*****)

BEGIN
  IF EDITING_FROM_SCREEN THEN BEGIN
    CASE ERROR_KIND OF
      LONGLINE : ND_SCREEN_PUT ('** LONG LINE **');
      COMMAND_ERROR : ND_SCREEN_PUT ('** ILLEGAL COMMAND **');
      NOT_FOUND : ND_SCREEN_PUT ('** NOT FOUND **');
      OTHER_ERROR : ND_SCREEN_PUT ('** ERROR **');
      STRING_NOT_FOUND : ND_SCREEN_PUT ('** STRING NOT FOUND **');
      WRITING : ND_SCREEN_PUT ('** ERROR IN WRITING **');
      TRANS_ERROR : ND_SCREEN_PUT ('** ERROR IN TRANSLATING **');
      UPDATING : ND_SCREEN_PUT ('** ERROR IN UPDATING **');
    END; (* CASE *)
    WRITE(' Press space bar to continue.....');
    READ (INPUT, SPACE_BAR);
  END (* THEN *)
  ELSE
    CASE ERROR_KIND OF
      COMMAND_ERROR : WRITELN ('** ILLEGAL COMMAND **');
      NOT_FOUND : WRITELN ('** NOT FOUND **');
      OTHER_ERROR : WRITELN ('** ERROR **');
      WRITING : WRITELN ('** ERROR IN WRITING **');
      TRANS_ERROR : WRITELN ('** ERROR IN TRANSLATING **');
      UPDATING : WRITELN ('** ERROR IN UPDATING **');
      STRING_NOT_FOUND : WRITELN ('** STRING NOT FOUND **');
      LONGLINE : WRITELN ('** LONG LINE **');
    END (* CASE *) ;
  END; (* ERROR *)

(*****
DUMMY ROUTINES
*****)

PROCEDURE STORE_A_LINE ; BEGIN END;
PROCEDURE EXIT_STORE_A_LINE; BEGIN END;
PROCEDURE DELETE_A_LINE ; BEGIN END;
PROCEDURE FETCH_A_LINE ; BEGIN END;
PROCEDURE ANALYZER; BEGIN END;
PROCEDURE TRANSLATOR; BEGIN END;
PROCEDURE WRITE_FILE; BEGIN END;
PROCEDURE READ_FILE; BEGIN END;

```

```

(*****
PROCEDURE BUILD_A_LINE (VAR DONE : BOOLEAN);
*****
This procedure is the primary procedure for creating text. The cursor
is positioned to column 12 and characters are repeatedly retrieved from
the key board until an EOLN condition or the maximum line length is
encountered. The data structure utilized is TEMP_LINE, which is an
array of characters. The pointer to the present position in the array
is LINE_INDEX. When a tab character is encountered the procedure fills
the data structure with the appropriate number of blanks.
*****
)

```

```

VAR
  C : CHAR;
  I : INTEGER;
  ALL_DONE, ESCAPE : BOOLEAN;
  VALID_SET_OF_CHAR : SET OF CHAR;

BEGIN
  VALID_SET_OF_CHAR := [CHR(32)..CHR(127),CHR(13),CHR(8),CHR(27),
                        CHR(3), TAB_CHARACTER ];
  FOR I := 1 TO 11 DO BEGIN    (* position cursor to column 12 *)
    C := ' ';
    WRITE (OUTPUT,C)
    END;
  LINE_INDEX := 12;
  GET_CHARACTER(VALID_SET_OF_CHAR,C,ALL_DONE,ESCAPE);
  WHILE (NOT EOLN (KEYBOARD)) AND (LINE_INDEX <> LINELENGTH)
    AND (NOT ALL_DONE) DO BEGIN
    IF C = CHR (8)  (* backspace *)
    THEN BEGIN
      LINE_INDEX := PRED (LINE_INDEX);
      IF LINE_INDEX <= 0 THEN LINE_INDEX := 1
      END
    ELSE IF C = TAB_CHARACTER (* tabsetting *)
    THEN
      REPEAT
        C := ' ';
        TEMP_LINE [LINE_INDEX] := C;
        LINE_INDEX := SUCC (LINE_INDEX);
      UNTIL (LINE_INDEX IN TABS) OR (LINE_INDEX > 79)
    ELSE BEGIN
      TEMP_LINE [LINE_INDEX] := C;
      LINE_INDEX := SUCC (LINE_INDEX);
      WRITE (OUTPUT,C);
    END;
    GET_CHARACTER(VALID_SET_OF_CHAR,C,ALL_DONE,ESCAPE);
  END; (* while *)
  READLN (KEYBOARD);
  Writeln;
  LINE_INDEX := PRED (LINE_INDEX);
  FOR I := LINE_INDEX TO LINE LENGTH DO
    TEMP_LINE [I] := NL;
  DONE := ALL_DONE;
END; (* BUILD A LINE *)

```

```

(*****)
PROCEDURE FIND_TOKEN (VAR NEXT_TOKEN : TOKEN);
(*****)
This procedure is designed to parse tokens from the global data
structure called INPUT_LINE. It successively scans the data stucture
until it finds a character which is not a blank and then determines
what type of token it is. It sets the fields in the record NEXT_TOKEN
according to the type of token, its location in the data structure,
and its value if it is a number.
(*****)

VAR
  C : CHAR;
  I : INTEGER;

BEGIN
  WITH NEXT_TOKEN DO
    BEGIN
      REPEAT
        C := INPUT_LINE [LINE_INDEX];
        LINE_INDEX := SUCC (LINE_INDEX)
      UNTIL C <> ' ';
      VALUE := PRED (LINE_INDEX); (* sets position where token found*)
      IF C = CHR(13)
      THEN TOKEN_KIND := NILTOK
      ELSE IF (C < '0') OR (C > '9') (* not a number *)
      THEN TOKEN_KIND := OTHERTOK
      ELSE BEGIN
        TOKEN_KIND := LINENOTOK;
        VALUE := 0;
        REPEAT
          (* change character to integer *)
          VALUE := 10 * VALUE + ORD(C) - ORD('0');
          C := INPUT_LINE [LINE_INDEX];
          LINE_INDEX := SUCC (LINE_INDEX)
        UNTIL (C < '0') OR (C > '9');
        END;
      IF TOKEN_KIND = OTHERTOK
      THEN LINE_INDEX := PRED (LINE_INDEX)
      END
    END
  END; (* FIND_TOKEN *)

(*****)
PROCEDURE QUIT;
(*****)
Writes signoff message to the user.
(*****)

BEGIN
  WRITE(CHR(27)); WRITE(CHR(43)); (* clear screen *)
  GOTOXY (15,12);
  WRITELN (' END OF HLSEW SESSION');
END; (* QUIT *)

```

```

(*****)
PROCEDURE FIND_STRING (STRING : LINE; STRING_LENGTH : INTEGER;
                      VAR START : INTEGER; VAR FOUND : BOOLEAN);
(*****)
This procedure is used for string searches. It receives as input the
string to be found and its length. It returns the position within the
array and a boolean FOUND. It proceeds through the data structure
TEMP_LINE looking for the first occurrence of the string. If it is
found, then another loop is initiated to count each match.
(*****)

```

```

VAR
    MATCH : CHAR;
    SUB_STRING, I : INTEGER;
    DONE : BOOLEAN;

BEGIN
    DONE := FALSE;
    FOUND := FALSE;
    MATCH := STRING [1];
    I := STRING_LENGTH - 1;
    IF START + I <= TEMP_LENGTH
    THEN REPEAT
        IF TEMP_LINE [START] = MATCH
        THEN BEGIN
            FOUND := TRUE;
            SUB_STRING := 0;
            WHILE (SUB_STRING <= I) AND FOUND
            DO BEGIN
                FOUND := FOUND AND
                    (TEMP_LINE [START + SUB_STRING]
                     = STRING [SUB_STRING + 1]);
                SUB_STRING := SUCC (SUB_STRING)
            END (* while *)
            END;
        IF NOT FOUND
        THEN START := SUCC (START);
        DONE := FOUND OR (START + I > TEMP_LENGTH);
    UNTIL DONE;
END; (* FIND_STRING *)

```

```

(*****)
PROCEDURE VERIFY_IT;
(*****)
Toggles the flag VERIFY_CHANGES off and on.
(*****)

```

```

BEGIN
    VERIFY_CHANGES := NOT VERIFY_CHANGES;
    WRITE ('VERIFY ');
    IF VERIFY_CHANGES
    THEN WRITELN ('ON')
    ELSE WRITELN ('OFF')
END; (* VERIFY IT *)

```



```

(*****
PROCEDURE FETCH_CURRENT_LINE (INPUT_LINE_NO : INTEGER;
                             VAR LINE_FOUND : BOOLEAN);
(*****
This procedure calls the storage system with a request for a certain
line number. It receives back the line of text and a boolean. The line
of text received is then transferred into a data structure called
TEMP_LINE for editing.
*****

```

```

VAR
  I : INTEGER;
  CURRENT_LINE : LINE;
  FOUND : BOOLEAN;

BEGIN
  FETCH_A_LINE(INPUT_LINE_NO,CURRENT_LINE,FOUND);
  LINE_FOUND := FOUND;
  IF FOUND
  THEN BEGIN
    LINE_NUMBER := INPUT_LINE_NO;
    TEMP_LENGTH := LINE_LENGTH;
    I := 0;
    REPEAT
      I := SUCC (I);
      TEMP_LINE [I] := CURRENT_LINE [I];
    UNTIL (CURRENT_LINE [I] = NL) OR (I > LINELENGTH);
    TEMP_LENGTH := I
  END
  ELSE BEGIN
    WRITE ('LINE ',INPUT_LINE_NO,' ');
    ERROR (NOT_FOUND)
  END
END; (* FETCH_CURRENT_LINE *)

```

```

(*****
PROCEDURE PRINT_TEMP_LINE;
(*****
Writes the data structure TEMP_LINE to the screen.
*****

```

```

VAR
  I : INTEGER;

BEGIN
  FOR I := 1 TO TEMP_LENGTH DO
    WRITE ( TEMP_LINE [I]);
  END; (* PRINT_TEMP_LINE *)

```

```

(*****)
PROCEDURE STORE_CURRENT LINE (MODE : MODE_TYPE);
(*****)
Transfers the working data structure into a permanent one and passes
it to the storage system.
(*****)

VAR
  I : INTEGER;
  MODE_OUT : MODE_TYPE;
  CURRENT_LINE : LINE;

BEGIN
  FOR I := 1 TO LINE_INDEX DO
    CURRENT_LINE [I] := TEMP_LINE [I];
  MODE_OUT := MODE;
  LINE_NUMBER := SUCC (LINE_NUMBER);
  STORE_A_LINE (CURRENT_LINE, LINE_NUMBER, MODE_OUT)
END; (* STORE_CURRENT_LINE *)

(*****)
PROCEDURE CHANGE_TEXT;
(*****)
This procedure is responsible for exchanging a target string of
text with a substitute string. It first scans the data structure
INPUT_LINE, looking for the proper command structure. As each portion
of the command is found it is stored in its own variable. If the
command is in the correct format then the procedure calls FIND_STRING
to locate an occurrence of the target string. When target string is
found then a swap is made with the substitute string.
(*****)

VAR
  DELIM : CHAR;
  NEXT : TOKEN;
  NEW_STRING, OLD_STRING : LINE;
  NEW_LENGTH, OLD_LENGTH, CHANGE_COUNT, OLD_START : INTEGER;
  STRING_START, I, J, INDEX : INTEGER;
  FOUND, LINE_CHANGED, SINGLE_CHANGE, LINE_FOUND : BOOLEAN;

```

```

BEGIN
  LINE_CHANGED := FALSE;
  STRING_START := 1;
  FIND_TOKEN (NEXT);
  WITH NEXT DO
    IF TOKEN_KIND <> OTHERTOK
    THEN ERROR (COMMAND_ERROR)
    ELSE BEGIN (* look for a delimiter *)
      DELIM := INPUT_LINE [VALUE];
      OLD_LENGTH := 0;
      INDEX := SUCC (VALUE);
      WHILE (INPUT_LINE [INDEX] <> DELIM) AND
        (INPUT_LINE [INDEX] <> NL)
      DO BEGIN (* read old string *)
        OLD_LENGTH := SUCC (OLD_LENGTH);
        OLD_STRING [OLD_LENGTH] := INPUT_LINE [INDEX];
        INDEX := SUCC (INDEX)
      END;
      IF INPUT_LINE [INDEX] <> DELIM
      THEN ERROR (COMMAND_ERROR)
      ELSE BEGIN
        INDEX := SUCC (INDEX);
        NEW_LENGTH := 0;
        WHILE (INPUT_LINE [INDEX] <> DELIM) AND
          (INPUT_LINE [INDEX] <> NL)
        DO BEGIN (* read new string *)
          NEW_LENGTH := SUCC (NEW_LENGTH);
          NEW_STRING [NEW_LENGTH] := INPUT_LINE [INDEX];
          INDEX := SUCC (INDEX)
        END;
        IF INPUT_LINE [INDEX] = NL
        THEN CHANGE_COUNT := 1
        ELSE BEGIN (* find out how many changes *)
          LINE_INDEX := SUCC (INDEX);
          FIND_TOKEN (NEXT);
          CASE TOKEN_KIND OF
            NILTOK : CHANGE_COUNT := 1;
            LINENOTOK : CHANGE_COUNT := VALUE;
            OTHERTOK :
              IF INPUT_LINE [VALUE] = '*'
              THEN CHANGE_COUNT := -1
              ELSE CHANGE_COUNT := 0
          END; (*case*)
        END; (*else*)
        SINGLE_CHANGE := CHANGE_COUNT = 1;
        FETCH_CURRENT_LINE (LINE_NUMBER, LINE_FOUND);
      END;
    END;
  END;

```

```

WHILE (CHANGE_COUNT <> 0) AND (LINE_FOUND)
DO BEGIN
  FIND_STRING (OLD_STRING, OLD_LENGTH,
              STRING_START, FOUND);
  IF NOT FOUND
  THEN IF SINGLE_CHANGE
  THEN BEGIN
    ERROR (STRING_NOT_FOUND);
    CHANGE_COUNT := 0
  END
  ELSE BEGIN (* move up and keep looking *)
    LINE_NUMBER := SUCC (LINE_NUMBER);
    FETCH_CURRENT LINE (LINE_NUMBER, LINE_FOUND)
  END
  ELSE BEGIN (* found a line to change *)
    CHANGE_COUNT := PRED (CHANGE_COUNT);
    LINE_CHANGED := TRUE;
    IF TEMP_LENGTH - OLD_LENGTH + NEW_LENGTH > 80
    THEN ERROR (LONGLINE)
    ELSE BEGIN
      IF OLD_LENGTH > NEW_LENGTH
      THEN FOR I := STRING_START + OLD_LENGTH
        TO TEMP_LENGTH
        DO TEMP_LINE [I - OLD_LENGTH + NEW_LENGTH]
          := TEMP_LINE [I]
      ELSE FOR I := TEMP_LENGTH DOWNTO STRING_START
        + OLD_LENGTH
        DO TEMP_LINE [I + NEW_LENGTH - OLD_LENGTH]
          := TEMP_LINE [I];
      TEMP_LENGTH := TEMP_LENGTH - OLD_LENGTH
        + NEW_LENGTH;
      FOR I := 1 TO NEW_LENGTH DO
        TEMP_LINE [STRING_START + I - 1] :=
          NEW_STRING [I];
      STRING_START := STRING_START + NEW_LENGTH;
    END; (*else*)
  END;
  IF LINE_CHANGED
  THEN BEGIN
    START_STRING := 1;
    STORE_CURRENT LINE (CHANGE_MODE);
    IF VERIFY_CHANGES
    THEN PRINT_TEMP_LINE;
    LINE_CHANGED := FALSE
  END
END (*while*)
END
END
END; (* CHANGE_TEXT *)

```

```

(*****
PROCEDURE MAKE_TAB_SETTING;
(*****
This procedure parses the data structure INPUT_LINE in order to
establish the desired tab setting. If no arguments are found then
then current tab settings are displayed. Otherwise, the values found
in the array are added to the current set of tab settings.
*****)

VAR
  I, N : INTEGER;
  C : CHAR;
  NEXT : TOKEN;
  OUT_STRING : STRING;

BEGIN
  FIND_TOKEN (NEXT);
  CASE NEXT.TOKEN_KIND OF
    NILTOK :
      BEGIN
        N := 0;
        IF EDITING_FROM_SCREEN THEN
          BEGIN
            FOR I := 1 TO 80 DO
              IF I IN TABS
                THEN BEGIN
                  REPEAT
                    N := SUCC(N);
                    OUT_STRING[N] := CHR(I MOD 10 + ORD ('0'));
                    I := I DIV 10;
                  UNTIL I = 0;
                  N := SUCC(N);
                  OUT_STRING[N] := ' ';
                END;
            ND_SCREEN PUT (OUT_STRING);
          END
        ELSE BEGIN
          FOR I := 1 TO 80 DO
            IF I IN TABS THEN
              WRITE(I, ' ');
            WRITELN;
          END;
        END;
      OTHERTOK :
        BEGIN
          C := INPUT_LINE [4];
          IF C = 'C' (* Cobol tab option *)
            THEN TABS := [8,12,16,20,24,32,36,40,56,73]
            ELSE ERROR (COMMAND_ERROR)
          END;
        LINENOTOK :
          IF NEXT.VALUE = 0
            THEN TABS := [ ]
            ELSE REPEAT
              I := NEXT.VALUE;
              IF (I > 0) AND (I < 79)
                THEN TABS := TABS + [I] (* set union *)
                ELSE ERROR (OTHER_ERROR);
              FIND_TOKEN (NEXT);
            UNTIL NEXT.TOKEN_KIND <> LINENOTOK
          END; (*case*)
        END; (*MAKE_TAB_SETTING*)

```

```

(*****)
PROCEDURE SET_TAB_CHAR;
(*****)
This procedure parses the INPUT_LINE to determine if the user wishes
to view the current tab character or establish a new one. If the
argument is a null then the current tab character is displayed.
(*****)

VAR
    NEW_TAB_CHAR : TOKEN;
    MESSAGE_OUT, TAB_STRING : STRING;

BEGIN
    FIND_TOKEN (NEW_TAB_CHAR);
    WITH NEW_TAB_CHAR DO
        CASE TOKEN_KIND OF
            NILTOK : BEGIN
                IF EDITING_FROM_SCREEN THEN
                    BEGIN
                        MESSAGE_OUT := ('The Tab Character is ');
                        ND_SCREEN PUT(MESSAGE_OUT); WRITE(TAB_CHARACTER);
                        WRITE(' Press space bar to continue....');
                        READ (INPUT, SPACE_BAR);
                    END
                ELSE
                    WRITELN('THE TAB CHARACTER IS ',TAB_CHARACTER);
                END;
            OTHERTOK : TAB_CHARACTER := INPUT_LINE [VALUE];
            LINENOTOK : ERROR (COMMAND_ERROR)
        END; (* case *)
    END; (* SET_TAB_CHAR *)

(*****)
PROCEDURE INSERT_TEXT;
(*****)
Parses the command found in INPUT_LINE and repeatedly builds a line
and stores it until a flag indicating done is recieved from the user.
(*****)

VAR
    NEXT : TOKEN;
    MODE_OUT : MODE_TYPE;
    DONE : BOOLEAN;

BEGIN
    FIND_TOKEN (NEXT);
    IF NEXT.TOKEN_KIND <> NILTOK
    THEN ERROR (COMMAND_ERROR)
    ELSE BEGIN
        MODE_OUT := INSERT_MODE;
        WHILE NOT DONE DO
            BUILD_A_LINE (DONE);
            STORE_CURRENT_LINE (MODE_OUT);
        END; (* while *)
        EXIT_STORE_A_LINE (* wakes up file handler *)
    END; (* INSERT_TEXT *)

```

```

(*****)
PROCEDURE DELETE_LINES;
(*****)
This procedure parses the command line INPUT_LINE, to determine
which lines of text to delete. Receipt of a null token will cause
the current line to be deleted.
(*****)

VAR
  NEXT : TOKEN;
  OK, FOUND : BOOLEAN;
  FIRST_LINE, LAST_LINE, I : INTEGER;

BEGIN
  OK := TRUE;
  FIND_TOKEN (NEXT);
  WITH NEXT DO
    CASE TOKEN_KIND OF
      NILTOK : BEGIN
        FIRST_LINE := LINE_NUMBER;
        LAST_LINE := LINE_NUMBER;
      END;
      OTHERTOK : OK := FALSE;
      LINENOTOK : BEGIN
        FIRST_LINE := VALUE;
        FIND_TOKEN (NEXT);
        WITH NEXT DO
          CASE TOKEN_KIND OF
            NILTOK : LAST_LINE := FIRST_LINE; (* one line delete *)
            OTHERTOK : OK := FALSE;
            LINENOTOK : BEGIN
              LAST_LINE := VALUE;
              FIND_TOKEN (NEXT);
              IF NEXT.TOKEN_KIND <> NILTOK
                THEN OK := FALSE
                ELSE (* everything's ok *)
              END
            END (* second case *)
          END
        END (* first case *)
      END;
    IF NOT OK
    THEN ERROR (COMMAND_ERROR)
    ELSE
      FOR I := FIRST_LINE TO LAST_LINE DO
        BEGIN
          DELETE_A_LINE (I, FOUND);
          IF NOT FOUND
          THEN BEGIN
            WRITE (I, ' ');
            ERROR (NOT_FOUND)
          END
        END
      END
    END; (*DELETE_LINES*)

```

```

(*****)
PROCEDURE LIST_IT;
(*****)
This procedure parses the command line, INPUT_LINE to determine
which lines to list. It then makes calls to fetch each line and
print it.
(*****)

VAR
  FIRST_LINE, LAST_LINE, I : INTEGER;
  OK, LINE_FOUND : BOOLEAN;
  NEXT : TOKEN;

BEGIN
  OK := TRUE;
  FIND_TOKEN (NEXT);
  WITH NEXT DO
    CASE TOKEN_KIND OF
      NILTOK : BEGIN (* default to a screenfull *)
        FIRST_LINE := LINE_NUMBER;
        LAST_LINE := FIRST_LINE + 20;
        END;
      OTHERTOK : OK := FALSE;
      LINENOTOK : BEGIN
        FIRST_LINE := VALUE;
        FIND_TOKEN (NEXT);
        WITH NEXT DO
          CASE TOKEN_KIND OF
            NILTOK : LAST_LINE := FIRST_LINE; (* one line only *)
            OTHERTOK : OK := FALSE;
            LINENOTOK : BEGIN
              LAST_LINE := VALUE;
              FIND_TOKEN (NEXT);
              IF NEXT.TOKEN_KIND <> NILTOK
                THEN OK := FALSE
                ELSE (* everything's ok *)
              END
            END
          END (* second case *)
        END
      END; (* first case *)
  IF NOT OK
    THEN ERROR (COMMAND_ERROR)
    ELSE
      FOR I := FIRST_LINE TO LAST_LINE DO
        BEGIN
          FETCH_CURRENT_LINE (I, LINE_FOUND);
          PRINT_TEMP_LINE
        END
      END
    END; (* LIST_IT *)

```



```

(*****)
PROCEDURE READ_COMMAND (VAR COMMAND : COMMAND_TYPE);
(*****)
This procedure determines what the command being issued is. It first
gets characters from the keyboard and places them in the data structure
INPUT_LINE. Then the first two letters of the command are placed into
the variable COMMAND_ID. The case statement then determines which
type of command is being issued.
(*****)

VAR
  C : CHAR;
  I : INTEGER;
  COMMAND_ID : ARRAY [1..2] OF CHAR;
  ALL_DONE, ESCAPE : BOOLEAN;
  VALID_SET_OF_CHAR : SET_OF_VALID;

BEGIN
  VALID_SET_OF_CHAR := [CHR(1)..CHR(9),CHR(13)..CHR(127)];
  LINE_INDEX := 1;
  REPEAT
    GET_CHARACTER(VALID_SET_OF_CHAR,C,ALL_DONE,ESCAPE)
  UNTIL C <> ' ';
  INPUT_LINE [LINE_INDEX] := C;
  WRITE(C);
  WHILE (NOT EOLN (KEYBOARD)) AND (LINE_INDEX <> LINE_LENGTH) DO
    BEGIN
      GET_CHARACTER(VALID_SET_OF_CHARACTER,C,ALL_DONE,ESCAPE);
      LINE_INDEX := SUCC (LINE_INDEX);
      IF C = CHR(8) (*backspace*)
      THEN BEGIN
        LINE_INDEX := PRED (LINE_INDEX - 1);
        IF LINE_INDEX <= 0 THEN LINE_INDEX := 0
        END
      ELSE BEGIN
        WRITE(C);
        INPUT_LINE [LINE_INDEX] := C;
      END
    END; (* while *)
  READLN(KEYBOARD);
  WRITELN;
  FOR I := LINE_INDEX TO LINELENGTH DO
    INPUT_LINE [I] := CHR(13);
  FOR I := 1 TO 2 DO
    COMMAND_ID [I] := INPUT_LINE [I];
  LINE_INDEX := 3;
  COMMAND := BADCOMMAND;
  IF COMMAND_ID [1] = NL THEN COMMAND := REPEAT_IT;
  CASE COMMAND_ID [1] OF
    'C' : IF COMMAND_ID [2] = 'H' THEN COMMAND := CHANGE;
    'D' : IF COMMAND_ID [2] = 'L' THEN COMMAND := DELETE;
    'E' : IF COMMAND_ID [2] = 'N' THEN COMMAND := ENEDIT;
    'H' : IF COMMAND_ID [2] = 'E' THEN COMMAND := HELP;
    'I' : IF COMMAND_ID [2] = 'N' THEN COMMAND := INSERT;
    'L' : IF COMMAND_ID [2] = 'S' THEN COMMAND := LIST;
    'S' : IF COMMAND_ID [2] = 'T' THEN COMMAND := SETTABS;
    'T' : IF COMMAND_ID [2] = 'C' THEN COMMAND := TABCHAR;
    'V' : IF COMMAND_ID [2] = 'E' THEN COMMAND := VERIFY;

  END; (*case*)
END; (*READ_COMMAND*)

```

```

(*****)
PROCEDURE EXECUTE_COMMAND (COMMAND_IN : COMMAND_TYPE);
(*****)
This procedure makes calls to the appropriate module to execute the
command. It is required in addition to READ_COMMAND because
the screen editor makes calls to the line editor with commands that
do not need to be parsed, but only executed.
(*****)

VAR LINE_FOUND : BOOLEAN;

BEGIN
  CASE COMMAND_IN OF
    CHANGE : CHANGE_TEXT;
    DELETE : DELETE_LINES;
    ENDEDIT : QUIT;
    HELP : HELP_IT;
    INSERT : INSERT_TEXT;
    LIST : LIST_IT;
    SETTABS : MAKE_TAB_SETTING;
    TABCHAR : SET_TAB_CHAR;
    VERIFY : VERIFY_IT;
    FETCH_IT : FETCH_CURRENT LINE (LINE_NUMBER, LINE_FOUND);
    STORE_IT : STORE_CURRENT LINE (CHANGE_MODE);
    BADCOMMAND : ERROR (COMMAND_ERROR);
    REPEAT_IT : BEGIN
      FETCH_CURRENT LINE (LINE_NUMBER, LINE_FOUND);
      PRINT_TEMP_LINE;
      LINE_NUMBER := SUCC (LINE_NUMBER)
    END
  END (* CASE *)
END; (* EXECUTE_COMMAND *)

(*****)
PROCEDURE INIT_LINE ;
(*****)
Initializes the line editor.
(*****)

BEGIN
  WRITELN (LOG_ON_MSG);
  WRITELN (HELP_MSG);
  TABS := [8,12,16,20,32,36,40,56,73];
  TAB_CHARACTER := '^';
  VERIFY_CHANGES := TRUE;
  LINE_NUMBER := 0;
  NL := CHR(13);
  COMMAND := BADCOMMAND;
  EDITING_FROM_SCREEN := FALSE;
END; (* INIT LINE *)

```

```

(*****
PROCEDURE LINE_EDIT (COMMAND_IN : COMMAND_TYPE);
(*****
Called by the command interpreter and acts as the driver for the line
editor.
*****)

BEGIN
  IF COMMAND_IN = EDIT_IT THEN
    BEGIN
      INIT_LINE;
      REPEAT
        READ_COMMAND(COMMAND);
        EXECUTE_COMMAND(COMMAND);
      UNTIL COMMAND = ENDEDIT
    END
  ELSE
    EXECUTE_COMMAND(COMMAND_IN)
  END;  (* LINE EDIT *)

END. (* UNIT EDITOR *)

```

APPENDIX D

PROGRAM

TEST DATA

APPENDIX D. PROGRAM TEST DATA

The following set of data is designed to exercise the program both structurally and functionally. Data points should be executed sequentially in both Line Editor and Screen Editor modes.

Line Editor Mode

L

<CR>

CH/AAA/B <CR>

AB <BS> <BS> HE <CR>

AB <BS> <BS> <BS> HE <CR>

CHAA/B <CR>

CH//B <CR>

CH///* <CR>

CH/AA/A/Z <CR>

CH/A/AAAAAA <CR>

CH/A,B <CR>

CX/AA/B <CR>

CH/AA/B/37000 <CR>

DL <CR>

DL* <CR>

DL 100,80 <CR>

DL 80,100 <CR>

DL -80,100 <CR>

DL 80,100/ <CR>

DL 37000,38000 <CR>

HO <CR>
HE <CR>
IN/ <CR>
IN 3 <CR>
IN A <CR>
IN, <CR>
^ ABCDEF <CR>
ABCDEF <BS> FG <CR>
<ESC> <CR>
IN <CR>
<CNTRL C> <CR>
LS <CR>
LS, <CR>
LS 15,16 <CR>
LS -30,40 <CR>
LS 40, 30 <CR>
LS 40,, <CR>
LS40,50 <CR>
LS 40 50,, <CR>
LS 37000,38000 <CR>
LX <CR>
ST <CR>
SX <CR>
STC <CR>
ST-C <CR>
ST,C <CR>
ST1,2,3 <CR>

ST 1,3,5,-6 <CR>
 ST 10 20 30 * <CR>
 TC % <CR>
 TC <CR>
 TC& <CR>
 TC -A <CR>
 TC 2 <CR>
 VE <CR>
 VE* <CR>
 VE2 <CR>

Screen Editor Mode

S
 A
 Z
 *
 2
 UP ARROW (* move through *)
 DOWN ARROW (* entire *)
 LEFT ARROW (* screen *)
 RIGHT ARROW
 I
 X
 2
 *

/
 <ESC>
 I
 C
 ABCDE <CR>
 <CR>
 <ESC>
 ABCDE <CNTRL C>
 I
 <CNTRL C>
 D
 A
 X
 2
 /
 DOWN ARROW <ESC>
 DOWN ARROW <CNTRL C>
 RIGHT ARROW <BS> RIGHT ARROW
 <ESC>
 RIGHT ARROW <CNTRL C>
 RIGHT ARROW TO THE END OF LINE
 X
 ABCDE*2/ <CR>
 <ESC>
 X
 ABCEDEF <BS> <BS>
 <CNTRL C>

X

AB <BS> <BS> <BS>

<CNTRL C>

R

/AA/E

R

<CR>

O

<ESC>

O

S

<CR>

O

T

<CR>

O

T

8 <CR>

O

S

10,20 <CR>

Q

REFERENCES

REFERENCES

- [Bow180] Bowels, K.W. Beginners Manual for the UCSD Pascal Software System, Byte Publishing Inc., 1980.
- [Grap80] Grappel, R.D. and Hemenway, J. "The Credit Goes to Intel." Mini-Micro Systems, Vol 13, June 1980 pp. 119-22.
- [Grue81] Gruenberg, F. "Making Friend With User-Friendly." Datamation, Vol 27, Jan 1981, pp. 108-112.
- [Huds80] Hudson, R. "Text Editor for the 6800." Interface Age, Vol 5, Sep 1980, pp 94-95.
- [Hurt76] Hurt, J.J. MEDIT: A Program to Edit Computer Source Programs, NTIS, Springfield VA, [1976].
- [Kans80] Kansas State University "OSMANUAL A Guide in Using MTM at Kansas State University", Manhattan, KS, 1980.
- [Kell77] Kelly, J. Guide to NED: A New On Line Computer Editor, Santa Montica, Rand Corp., [1977].
- [Ledg80] Ledgard, H.F. "An Experiment on Human Engineering of Interactive Software." IEEE Transactions, Software Engineering, Vol 6, Nov 1980 p. 602.
- [Mill80] Miller, A.R. "Work-Master Micropro's Video Editor for CP/M.", Interface Age Vol 5, Jan 1980, pp. 118,120,122-123.
- [Nagy79] Nagy, G. and Embley, D.W. Cardinal and Venial Sins in the Design of Programming Editors for Beginning Student Use, University of Nebraska, [1979] in "1979 ACM Computer Science Conference Proceedings." New York, 1979.
- [RaynNA] Rayner, D. Designing User Interfaces For Friendliness, Amsterdam, Netherlands, no date.
- [Shne80] Shneiderman, B. Software Psychology New York, Winthrop Publishers Inc., 1980.

- [Snee78] Sneeringer, J. "User-Interface Design for Text Editing: A Case Study." Software Practices and Experiences, Vol 8 Sep-Oct 1978, pp. 543-558.

HLSEW
EDITING SYSTEM

by

DAVID BARTLETT AARONSON

B.S., Troy State University, 1977

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

The Computer Science Department of Kansas State University is currently in the process of developing a High level Software Engineering Workstation (HLSEW). This interactive workstation will be the software implementation of an "intelligent terminal", designed to aid in program development. It will consist of five modules: (1) The Command Interpreter, which serves as the main driver for the workstation; (2) The Software Engineering Analyzer that provides the programmer with Halstead's and McCabe's software metrics throughout the development of his program; (3) The Translator, which converts a program written in a program design language (PDL) into compilable COBOL source code; (4) The Storage System for mapping and accessing information on secondary storage; and (5) The Editing System which serves as the tool the programmer utilized to create and edit text. This project describes the design and implementation of the HLSEW Editing System.

Intitial constraints required that the design and implementation be compact, portable, powerful, extensible, flexible and present a friendly user interface. This imposed conflicts among those constraints which seemed to advocate a small powerful line oriented Editor, and those which appeared to call for a big powerful screen oriented Editor. After some investigation into other approaches, a novel course was chosen which satisfied the majority of conflicts.

A Line Editor with a Screen Editor front end was designed. The Screen Editor front end manages the information on the screen, interprets the commands from the user and passes them, in appropriate form, to the Line Editor to do the actual editing. This approach offers

portability of the System because the terminal specific code (the screen front end) is kept to a minimum.

The Editing System was implemented on a PDQ/3 Microcomputer in UCSD Pascal and consists of approximately 1800 lines of source code.