

295

**/THE DISK STORAGE SYSTEM OF THE
HIGH LEVEL SOFTWARE ENGINEERING WORKSTATION
(HLSEW)/**

by

Russell J. Holt
of

B. S., Washburn University of Topeka, 1980

A Master's Report

submitted in partial fulfillment of the

requirements for the degree

Master of Science


Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Ks

1985

Approved by:



Major Professor

LD
2668
0.84
1985
H64
C. 2

A11202 996541

TABLE OF CONTENTS

<u>Chapter</u>	<u>Page</u>
I. INTRODUCTION	1
Structure	2
Data Storage System	3
Requirements	3
Unique Data Structure	4
Constraints	4
Memory Usage	4
Disk Usage	6
Response Time	7
Resource Usage	7
Portability	8
Summary	9
II. DESIGN	12
Editor	12
Analyzer	13
Translator	14
Design Issues	14
Design Specifications	16
File Structure	16
Logical Representation	17
Memory Conservation	18
Interfacing	19
Integrity	20
Portability	20
Storage System Design	21
Functional Overview	21
Internal Data Structure	23
Data Manipulation Considerations	24
Summary	28
III. IMPLEMENTATION	35
Reading Files	36
Retrieval of Text	38
Inserting Text	39
Deleting Text	40
Changing Text	41
Writing Files	41
Analyzer Data Transfer	42
Translator Data Transfer	43
Memory Management	43
System Enhancements	44

IV. TESTING	51
V. EXTENSIONS AND FUTURE CONSIDERATIONS	53
REFERENCES	55
APPENDIX I. Procedure Specifications	57
APPENDIX II. Parameter Specifications	66
APPENDIX III. HLSEW Declaration Source Code	75
APPENDIX IV. Storage System Source Code	78
APPENDIX V. Editing System Source Code	98
APPENDIX VI. Installation Source Code	128

Diagrams

1. Figure 1.1 HLSEW Structure	10
2. Figure 1.2 HLSEW System Communication	11
3. Figure 2.1 Linking of the HLSEW System	29
4. Figure 2.2 Generalized Linked Structure	30
5. Figure 2.3 User View of Text File	31
6. Figure 2.4 Storage System View of Data	31
7. Figure 2.5 Storage System View of Problem	32
8. Figure 2.6 Storage System View of Problem	33
9. Figure 2.7(a) User View of Text	34
10. Figure 2.7(b) System View of Text	34
11. Figure 3.1 Hierarchy Diagram	48
12. Figure 3.2 Read File Flow	49
13. Figure 3.3 System Modular Form	50

CHAPTER I

BACKGROUND

Introduction

The author has developed the Data Storage System of an interactive workstation referred to as the High Level Software Engineering Workstation (HLSEW). This interactive workstation which functions as an "intelligent terminal", is designed to aid programmers who use a pseudo-English programming language called a Program Design Language (PDL). "An intelligent terminal can do some local processing without communicating with a host computer, it offer users flexibility while freeing the host for other tasks" [CO82]. The HLSEW functions as an intelligent terminal which allows: 1) the creating and editing of a PDL file, 2) the calculation of program metrics, and 3) the translating of a PDL file into a compilable source code file. The eventual purpose of the HLSEW project is to develop an intelligent workstation that helps to identify problems as the user enters the PDL code.

PDL is a psuedo-code, ambiguous structured-programming tool with a syntax that contains statements such as REPEAT UNTIL, CASE, and DO WHILE [CA75]. The expressions and the statements of the PDL are of one language (i.e., English) and the control structure is in another (i.e., a structured

programming language). There are many variations of PDL syntax which are in the style of APL, Fortran, Pascal, and COBOL. In the case of this project, the PDL being targeted will use a COBOL PDL. With the use of the HLSEW, a complete, executable ANSI COBOL program can be translated from a PDL program file.

The HLSEW project was originally designed for a PDQ-3, which consists of a LSI-11 CPU, 128K RAM memory, dual 8 inch floppy disk drives, and runs the University of California at San Diego (UCSD) p-system. Further development of the project resulted in the system being transferred to an IBM or IBM-compatible micro computer.

Structure

The structure of the current HLSEW is divided into four sections: (1) The Editor, (2) The Data Storage System, (3) The Software Engineering Analyzer, and (4) The Translator.

The Editor serves as the input medium to the workstation system and functions as a common link by which each of the four modules can communicate. The Editor functions as a "Screen Editor" on screen oriented devices or as a "Line Editor" on line oriented devices.

The Software Engineering Analyzer supplies Halstead's and McCabe's complexity measures. Information such as complexity, implementation level, and volume levels within a program or within a block of code are calculated by the Analyzer and displayed to the user.

The Translator functions to automatically translate a PDL file into a compilable COBOL source code file. The COBOL file may then be compiled on the physical CPU at hand or on a mainframe located in another geographical area.

The Data Storage System acts as a interface medium through which the Editor, Analyzer, and the Translator make requests for: 1) data that has been stored in memory or in a disk file, and 2) data to be stored in memory or into a disk file.

DATA STORAGE SYSTEM

Requirements

The HLSEW data structure was built because the necessary data structure was not present in the UCSD p-system. This data structure must be present to preserve the logical division of the PDL. The design of the HLSEW and its data structure resulted in the design of a new system rather than using the existing UCSD p-system. Also included is the need to simultaneously handle three files within the internal memory of the machine, and the need to create a file structure which is compatible with the operating system. The three files necessary within the operation of the HLSEW are: 1) the physical code file located on the disk, 2) the software metric file, and 3) the translated PDL file.

Data Structure

The interfacing problems encountered in implementing a data structure with the present UCSD p-system could be solved by modifying the editor or by accessing the system routines of the UCSD p-system. Because neither could be done, a custom editor and storage system was built. The references supplied with the PDQ-3 and the UCSD p-system indicated that interfacing with the operating system was possible, but the necessary documentation needed was not available.

CONSTRAINTS

The initial constraints on the storage system included: 1) the amount of internal memory available in which data could be stored, and 2) the amount of free disk space needed to store an edited file. Secondary to these constraints was the need for: 3) speed of data retrieval, 4) the efficient use of the memory for the programs, and 5) portability to different machines.

Memory Usage

Reducing the amount of internal memory used by the HLSEW system was the main concern throughout the entire development period. It was evident that even though a new editing system could be constructed, the code which makes up the major body of the HLSEW added onto the UCSD

p-system could easily tax the memory of a small host machine. Although the available memory of a machine may in many cases be scarce, a file of moderate size must be maintained and edited. This memory constraint would be most evident in the majority of micros present on the market today. Even though an increasing number of companies are making larger memories standard, there will probably always exist those machines that will be classified as basic units and will contain limited resources.

In summary, an efficient method of using the available internal memory is a necessary system feature so that a moderate sized edit file could be maintained and so machines with smaller memory sizes could run the HLSEW system.

To design a system that could be used on both small and large machines, it may become necessary to design two different storage systems that would each be suited for the type of configuration that each system was being run on. This type of design would defeat the purpose of the HLSEW development project. Designing multiple systems would defeat a design goal that will be discussed further in this chapter under the topic heading of Portability. The most desirable system would be one which could run on any machine of any type of configuration, whether that configuration was large or small. Of course, concessions would no doubt have to be made so that any type of configuration could be used. One of the concessions made would be an increase in response time versus an increase in the size of an edit file.

The machines which contain sufficient memory to allow liberal use of the available resources could use a storage system that utilizes the internal memory and would display a high response time, and through careful planning, a large edit file could be maintained. Those machines that were not supplied with a large amount of memory could use a system which makes use of the available storage capacity of the floppy disks and would display a slow response time but would also be capable of maintaining a large edit file.

Disk Usage

The second constraint is the need to have a storage system that can structure data on a disk so that the amount of wasted space is limited (an efficient usage of storage). Various methods could be devised but not every method can efficiently store the data into a disk file. The USCD p-system uses a method of reducing or compacting its files by eliminating unnecessary blanks. This compacting method increases the ratio of data stored to disk space used. By compacting the amount of space needed to store a file, an increase in the amount of overhead needed to store and retrieve that data is created. Designing an efficient method that would have a balance between the space needed to store the data versus the time tradeoff needed to access the data, and would be compatible with the p-system would add to the positive aspects of the HLSEW system.

Response Time

From a user's viewpoint, the time needed to enter data should be greater than the time needed waiting for the processing of the desired data. The reading and writing speeds of floppy disk drives found on today's micro are considerably slower compared to the speeds found using a hard drive or internal memory. The amount of time needed to move any data to or from the disk increases as the number of disk accesses increases. To help eliminate the amount of time consumed by the retrieval of data, a method should be devised that could retrieve a data file with a limited number of disk accesses. By limiting the number of disk accesses, an improvement in the system response time would be observed.

Resource Usage

To this point, the concerns of major importance have involved the efficient use of the resources in respect to the file being edited by the HLSEW system. An issue that must not be overlooked is the constraints placed on the code of the HLSEW by the UCSD p-system. The mere physical size of the code can go beyond the operating capability of a machine. These size constraints become most evident when the monitoring of an executing system takes place. The UCSD p-system provides a method called segmenting by which efficiency of the program execution can be incorporated. By

carefully designing the HLSEW system code, an efficient use of the UCSD p-system and an efficient execution of the software system code can take place. A more detailed description of the use of code segmentation can be found in Chapter 4.

Portability

Portability is also desirable when considering a design which will not be limited to just one machine type. Too many software packages on the market today are structured for a particular computer. If a software package is targeted for a specific machine, the proper functioning of that software on any other machine may be impossible. Some companies have given forethought to this problem and have supplied the user with routines which tell the software what type of computer, terminal, printer (Wordstar, dBase II), and in some cases, the version of operating system that is present (Lotus 1-2-3) [MI83] [AT83] [LO83]. As a marketing issue, the less machine specific a software package is, the greater the distribution will be. The usage of features which are unique to a given machine or terminal must be avoided so that problems which do occur can be kept at a minimum and the solutions to those problems can be easily rectified. Complete portability cannot always be guaranteed from machine to machine. There is always a set of bottom line requirements that is needed to run any software package such as a minimum amount of internal memory, the particular

number of disk drives, or a specific implementation version of the operating system. The portability of a software package can be increased to a greater extent by refraining from the use of machine specific features.

Summary

The High Level Software Engineering Workstation (HLSEW) is an interactive workstation which aids the programmer who uses a PDL code. The HLSEW system is a tool that a programmer can use to edit a PDL file, to calculate program metrics, and to translate a PDL file into a compilable source code file. The Data Storage System of the HLSEW controls a unique internal data structure that maintains the attributes of the physical text file.

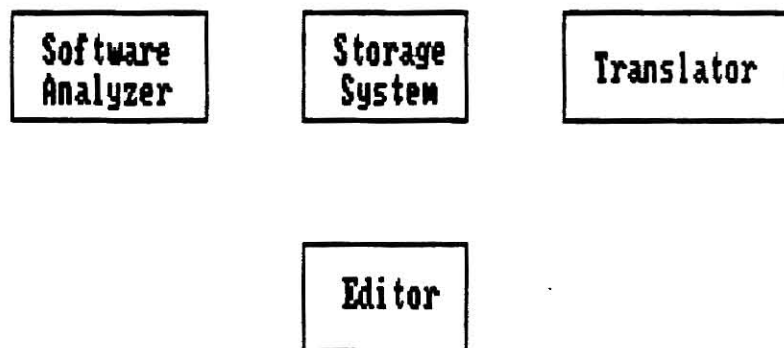


Figure 1.1
HLSEM Structure

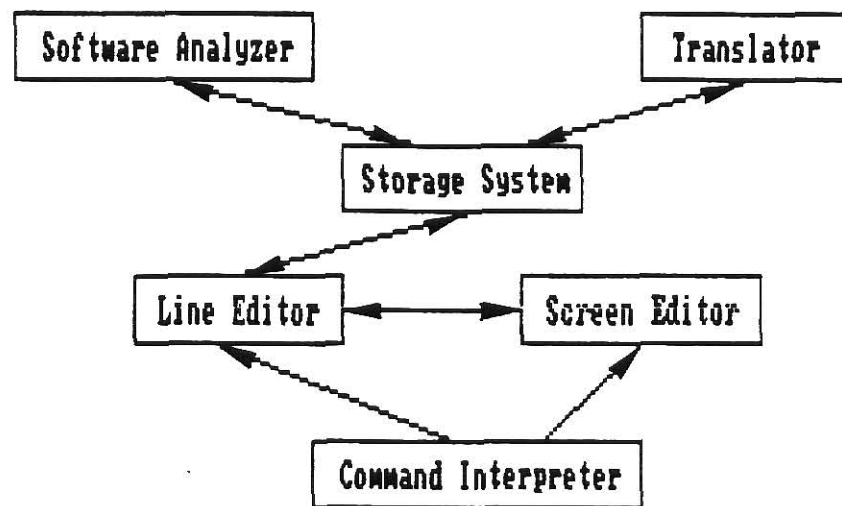


Figure 1.2
HISEW System Communication

Chapter II

Design

SYSTEM OVERVIEW

The HLSEW system is divided into four modules 1) Editor, 2) Software Analyzer, 3) Translator, and 4) Storage System. The structure of these modules is illustrated in Figure 2.1.

Editor

The majority of all data for which the storage system is responsible is either sent to or received from the editing system. Specifically, the editor will request to store, delete, transfer, or change specified lines of text. The editing system is divided into three operating modules: (1) the Command Interpreter, (2) the Line Editor, and (3) the Screen Editor. All the calls which come from the editing system originate either in the Command Interpreter or the Line Editor.

The Command Interpreter is responsible for issuing requests for files to be opened and closed by name. When these requests are issued, it is necessary for the storage system to open a named file and, if no file exists, to create a new workfile to be named later. Upon completion of an editing session, a request is issued to save and close

the file which has been edited. The storage system has the task of informing the Command Interpreter if any errors have occurred so that the proper actions can be taken to recover from the errors and to inform the user of such problems.

Within the Storage System are various sections of code which detect problems or errors that may occur when a request is made to delete, add, or change a line of text. Depending on the type of error that occurs, the storage system or the editing system will take actions to recover, and in some cases, both systems do error recovery on occurrence of the same error. Figure 2.2 represents the communication which takes place between the Command Interpreter, Line Editor, Screen Editor, and Storage System.

Analyzer

The initial command to execute the Software Analyzer originates from the Command Interpreter. Once the execution begins, the Analyzer functions independently from the editing system and communicates with the Storage System directly. The Analyzer requests a named file to be brought into memory. The Analyzer then issues calls to the Storage System requesting the software metrics and the text from a given section of the file which corresponds with the Metrics. This section of text retrieved represents either a predetermined block of text which is identified by name, or it can represent an entire file. The text is transferred to

the Analyzer one line at a time. When the processing of the Analyzer has completed, a set of software metrics is transferred back to the Storage System which then stores this set of metrics into the disk file and records the address of the metrics.

Translator

The communication between the Translator and the Storage System is similar to that with the Analyzer. Execution of the Translator originates in the Command Interpreter. The Translator is also independent of the Editing System and communicates with the Storage System directly. Requests for data is on a line by line basis. The text which is transferred to the Translator is generally a complete file, starting at the beginning of the file and continuing until the end of the file has been encountered. There exists no restrictions against requesting data on a named block basis. A request is issued to locate the name (label) of a block, the data is then transferred to the Translator until the end of the block of data is encountered.

DESIGN ISSUES

The Data Storage System was designed for the purpose of storing and retrieving data for the HLSEW system. Although the storage of data onto a floppy disk unit is itself a

relatively simple task, the continual updating which occurs in an editing session changes the physical structure of the file drastically. Various methods by which data could be stored were examined. Each method of data storage was judged by the advantages and disadvantages that it exhibited.

A linked structure representing the user supplied text proved to be the most logical and effective method of maintaining a text file. The major issue concerning the linked structure involved the structure of the information stored at each node. The Cornell Program Synthesizer implemented a tree structure which uses goto labels to indicate op-code entry and continuation points [TE81]. The tree built by this method is transversed in a preorder fashion in which no backward pointers are used resulting in the need to go all the way around the tree to achieve a backward movement.

Also considered was a method of storing at each node an entire block of data (512 bytes). This proved to be an effective way of conserving the amount of memory needed to store the text internally and decreased the number of disk accesses, but the overhead needed to maintain that type of structure was quite large.

To reduce the overhead required to parse and maintain a tree structure, a doubly linked structure which is traversed in a sequential fashion was chosen.

Design Specifications

The original design specifications stated that: 1) a data file must display a generalized physical structure so that the accessing of that file by other software packages may be achieved, 2) a data structure must be developed that will maintain a logical representation of the physical data, 3) the internal data structure specified must efficiently use the available memory resources and allow the prompt retrieval of data.

An underlying detail to the design issue was a concern that: 1) there may possibly be the need to interface the storage system with other systems or modules, 2) data integrity had to be addressed because no interaction occurs between the user and the storage system, and 3) an emphasis on the system design should be made so that there would be a high degree of portability. The present design supports all the original specifications.

File Structure

The file structure by which physical data is stored, was designed so that other software packages can easily access the stored data. The design of the storage system successfully avoids the file structuring problems the UCSD p-system has, thus allowing data to easily be accessed by other software packages.

The UCSD p-system contains one major constraint that unfortunately is inherent to this well designed operating

system. The designers of the UCSD p-system addressed the issue of managing resources and devised several methods of conserving internal and external memory. The main method the UCSD p-system uses to conserve memory resources is to replace the leading blanks with numeric codes. This method is carried from the internal storage of data to the external storage of data into a disk file. By representing blanks with numeric codes in the disk file, the access of the text by other operating systems and devices is difficult. It became necessary for the user or the programmer to know the fine details of the data structure and to construct special routines which can use the stored numeric codes to return the text to its proper structure.

Logical Representation

Designing a structure which could maintain the logical attributes of a file resulted in the ability to move quickly through the data stored within the internal memory buffer. The logical representation not only decreases the time needed to retrieve data, but it also has the ability to retrieve information about different sections of code which are needed by the software analyzer.

Memory Conservation

The amount of memory allocated to run the HLSEW on any particular machine is conserved by allowing no more than two modules (Editor, Storage System, Translator, Software Analyzer) to function at any given point in time. When the HLSEW is fully operational, only two of the system modules are executing at one time. The storage system must always be kept in memory to supply text to the other module which is resident in memory. The passing of information takes place between the storage system and 1) the editor, 2) the translator, and 3) the software analyzer. The translator and the software analyzer never interact with each other, and the editor only initiates the execution of translator and the software analyzer. Once the execution of either the translator or the software analyzer is started, the editor code is no longer needed in memory until a return to the calling procedure is indicated.

There is no particular size limit on the text file being edited. The storage system uses a dynamic memory buffer which allows the size of a file to be totally dependent upon the amount of internal memory available. This is accomplished by means of constantly monitoring the amount of memory being consumed during an editing session. A procedure is used which receives from the operating system the amount of memory remaining in the host machine.

Because of the complexity of the data structure involved, a systematic approach to designing the system was

used. A method that efficiently uses the internal memory was the first implemented. This method proved to work effectively on machines with limited internal memory. After the structure of the storage system was tested and verified to function properly, the question of improving the speed of data retrieval was addressed. To improve the performance of the system, features found in the operating system were used that can make effective use of a larger memory.

Interfacing

The need for a storage system that other modules could easily interface with was a desirable factor. Other systems or modules which need to interface with the storage system may have been constructed outside the realm of this project by others who wish to use and incorporate the code which is contained within the storage system. To help eliminate the problems which may occur when attempting to interface with the system, a general description of the functions is given in this chapter and a detailed description of the code including actions, parameters used and the expected side effects have been supplied in Appendix I.

Integrity

A concern of major importance with this project was the guarantee that the data retrieved by the Storage System corresponded with the data requested by the editor. In addition to the correct retrieval of data was the need for a system that would contain facilities which would prevent any system failures caused by operator or data error thus ensuring the continual operation of the total software package.

Portability

The question of portability of the final HLSEW software package was also of great concern. The portability of a software package can be improved by eliminating hardware dependent functions and by designing a code structure that can be run on a large range of internal memory sizes.

Fortunately, the UCSD p-system is very well equipped with functions which allow the programmer to call upon the operating system to perform tasks or supply information. At the present time, when the HLSEW is in operation, approximately 70K of memory is available for use; this does not include the memory needed by the Translator or the Software Analyzer. The information concerning these two modules is not yet known, so the actual amount memory needed for the final HLSEW system can only be estimated.

Terminal dependent features were avoided by designing

functions which do not rely on the hardware that is attached to the host machine. Not all of the terminal dependencies could be avoided. Variations in a small number of vital HLSEW functions could not be overcome. The solution to this was to design an installation routine that supplies the HLSEW system with information about the host machine. The information supplies value codes for cursor and screen control. By avoiding machine dependent functions and with the use of the installation routine, the portability of the HLSEW system is increased.

STORAGE SYSTEM DESIGN

Functional Overview

Initially, the obvious task that the Storage System has is the responsibility of handling any data which has been entered by the editing system. The Storage System is also responsible for the opening and closing of files, the reading and writing of data which has been or will be stored on a disk unit, and the monitoring of available memory. These tasks represent linking points at which the Editor, the Translator, and the Software Analyzer will send data to and receive data from the Storage System. The critical point of interest within each of these linking points is that each unit of the HLSEW system is highly dependent upon the proper parameter values to be passed so that the correct data and the correct information can be guaranteed. Figure

2.1 illustrates the linking of the storage system with the editor, translator, and the analyzer.

Along with the communication which takes place between the editor and the storage system, a sequence of steps exists which mimics or duplicates the steps that each unit takes. In reality, the editor functions not as a screen editor, but as a line editor and maintains only one line of text at a time in its own memory buffer. If the editor makes a logical movement "up" or "down" in the text file, the storage system must also repeat the same movement. The duplication of steps which takes place must occur so that editor and the storage system can maintain the same logical position in the text file. The difficulty in implementing the duplication tasks resulted because information passed between the units in the form of parameters many times represented logical values rather than physical values. Under certain conditions, the logical and physical representation of the data can become disjoint resulting in meaningless information. To avoid the editor and the storage system becoming disjoint during processing, this duplication of tasks takes place.

Internal Data Structure

The internal data structure that is built and maintained by the storage system is a linking of information found in the physical text file and the logical representation of that text file. The physical file is simply a sequential storage of the text file.

When the physical file is read into memory, the data from the file is stored into the structure detailed in Figure 2.2. This structure is comprised of two double linked lists which are in turn double linked to each other. Two types of linkings exists in the data structure. The first type of linking builds a structure which attaches each consecutive physical or logical record to the next record, i.e. record 1 is linked to record 2, record 2 is linked to 3 and so forth. The second linking connects the logical structure to the physical structure. This logical structure includes the location of a block of text, the number of lines in that block, and the relative address of the software metrics associated with that block. The need to obtain specified sections of code and information pertaining to that code quickly and to maintain a logically sequential text file without the disruption of superfluous data is the reason for this type of structure.

Data Manipulation Considerations

Throughout the development of the storage system, the expanding and the contracting of the data structure and the manipulation of the data contained in that structure, resulted in the presence of various data manipulation problems. To help understand the problems that can occur, the most difficult situations encountered are described.

In Figure 2.3 an arbitrary text file is displayed and the structure of that file is shown as it appears to the user. The lines of text shown represent three blocks of code with the labels A, B, and C. Within each of these block labels is the associated code which corresponds with each block. Each line of code is given a relative name to indicate of which block it is a member. Figure 2.4 is the internal representation of the data as it appears to the storage system. On the left side of the linked list appears the logical mapping which informs the storage system about the information pertaining to each block, and the physical representation of the text file appears on the right side. Looking at the logical side of the linking, one can determine that block "A" has four lines of text associated with it. Block "B" has three lines associated with it, and block "C" has two lines of text.

Using this description of the text file, some situations that can occur are as follows:

Case 1. When "B" is deleted, two events must take place; the text which was originally linked to block "B" must now become linked to block "A", and the number of lines of text which once belonged to block "B" must be added to the total lines belonging to block "A". The resulting structure is illustrated in Figure 2.5.

Case 2. Assume, as illustrated in Figure 2.2, that during an editing session, block label "A" should either be deleted, or it was inadvertently omitted. The storage system would not be able to associate the following text with any label nor could the Software Analyzer. To help satisfy the structural problems within the internal representation, a dummy block label is created and linked to the indicated text. This dummy label will exist only as long as there is no actual label present, and once a replacement label has been entered, the dummy label is removed. Even though a dummy label would be present in the logical linking, the unlabeled lines remain just that, unlabeled. To illustrate that the block values are unlabeled, the "?" has been placed in the representation to show that there exists an unknown value and location, as illustrated in Figure 2.6. This

will stay true until a label is inserted to replace the nonexistent label which is represented by the dummy label.

Case 3. Reverse the situation that is described in Case 1. A label is inserted into the physical linking. Using Figure 2.5 again, assume that the label "B" is inserted where it was originally located. The linking which is present between the logical and physical lists must now be rebuilt and the line numbering for labels "A" and "B" are updated. The resulting structure is illustrated in Figure 2.4.

Case 4. Cobol is a language which is structured so that the compiler can determine the function of the code by the location where the code physically appears on a line. This can be referred to as a column oriented language.

Before a line of text can be properly processed by the storage system, it must be determined whether the indicated text is a block label or some other standard type of text. This is accomplished by checking each incoming line for the presence of text in columns 8 through 11. This determining factor governs the type of processing which will take place within the storage system. If any part of the text being accessed is located within columns 8 to 11, a block label is formed. Any existing text can be changed and modified during an

editing session. If a block label becomes repositioned so that its beginning characters are no longer located between column 7 and column 12, the internal linking is rebuilt to reflect the change which has taken place. If a standard line of text which was originally located in column 12 or beyond is repositioned so that its beginning characters fall between column 7 and column 12, the linking is also rebuilt to reflect the change. A more detailed description of this action can be found in chapter 3 under Changing Text.

Case 5. Using Figure 2.7(a), assume the following sequence of events. On the left is a small text file as it appears at some point of time during an editing session. With the use of a change command, the label "B" is altered giving the structure shown in 2.7(b). Even though "B" has not been deleted, the linking is changed and the appearance of a label being deleted takes place. The only difference though is that the total number of lines does not change.

The previous sequence can now be reversed. Starting with the structure found in 2.7(b), changing the line where "B" is found can result in the structure found in 2.7(a).

Summary

The Data Storage System was designed to maintain a physical text file and the logical information associated with that file. The specifications which controlled the design of the storage system were 1) the need for a generalized data file structure, 2) a logical representation of the physical text file, and 3) an internal data structure which will use the available memory efficiently.

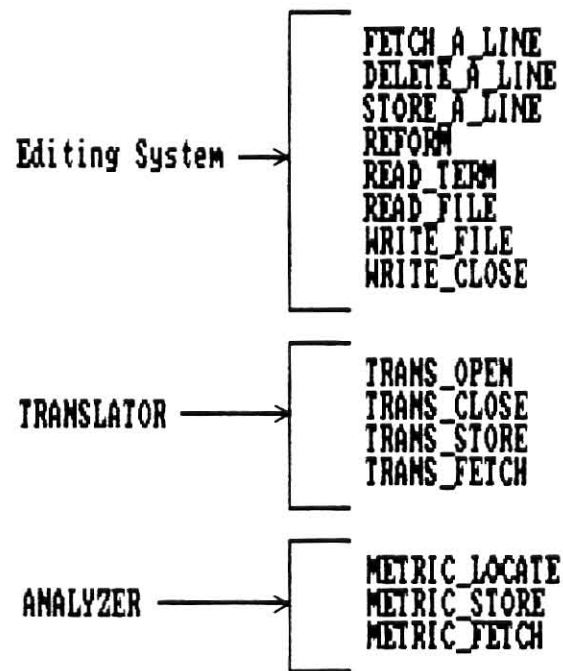


Figure 2.1
Linking of the HLSEM System

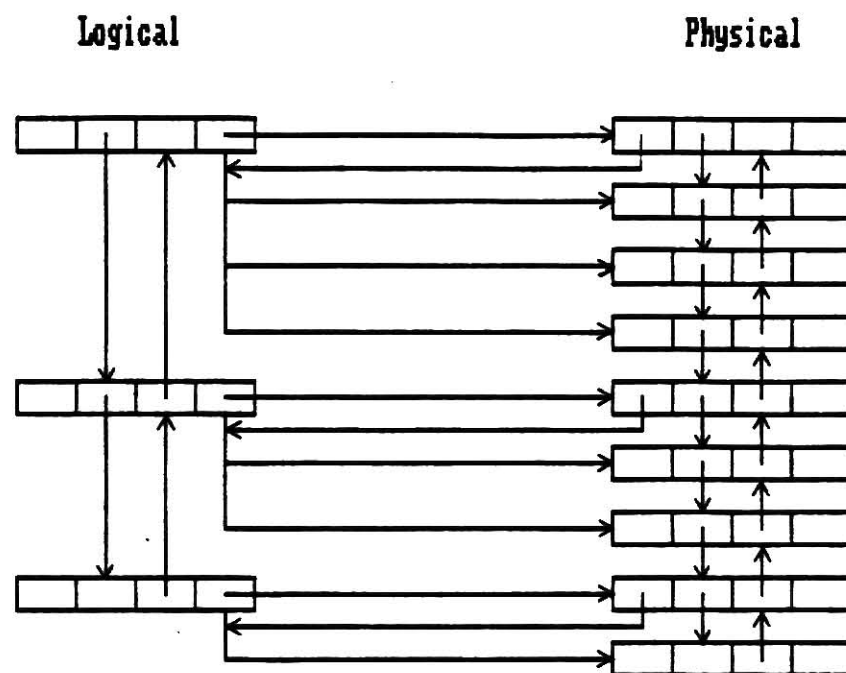


Figure 2.2
Generalized Structure

A	A1
	A2
	A3
B	B1
	B2
C	C1

Figure 2.3
User View Of Text File

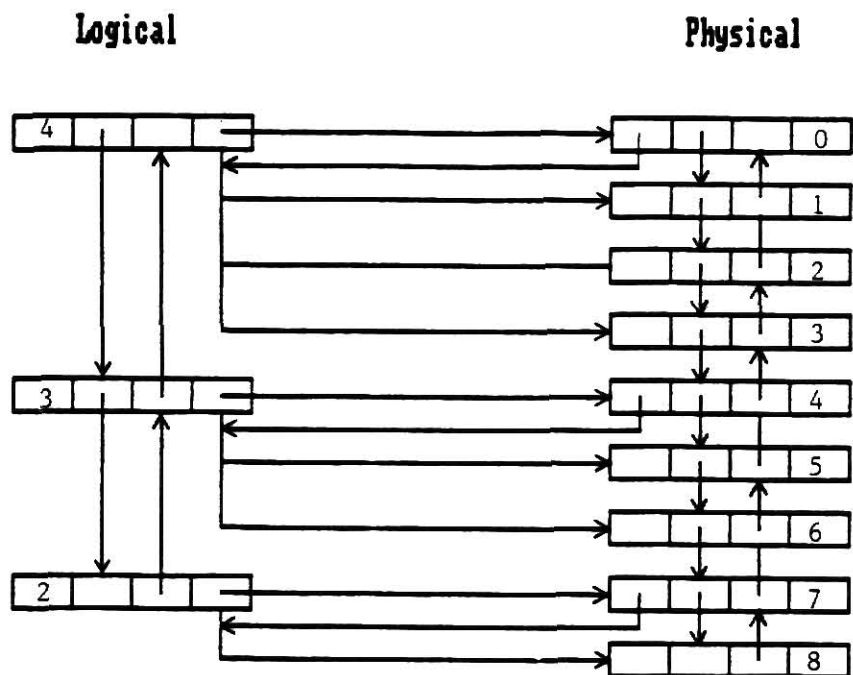


Figure 2.4
Storage System View of Data

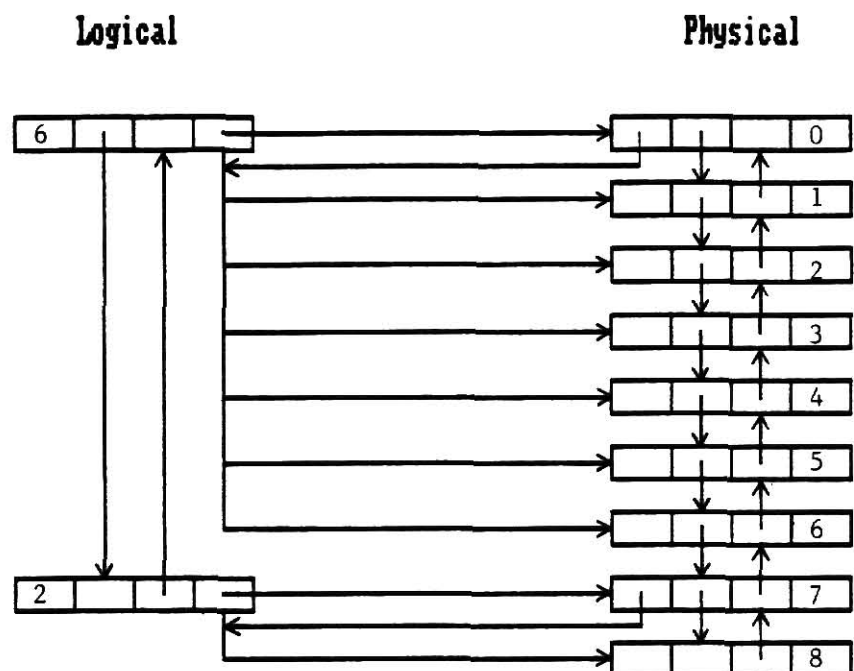


Figure 2.5
Storage System View of Problem

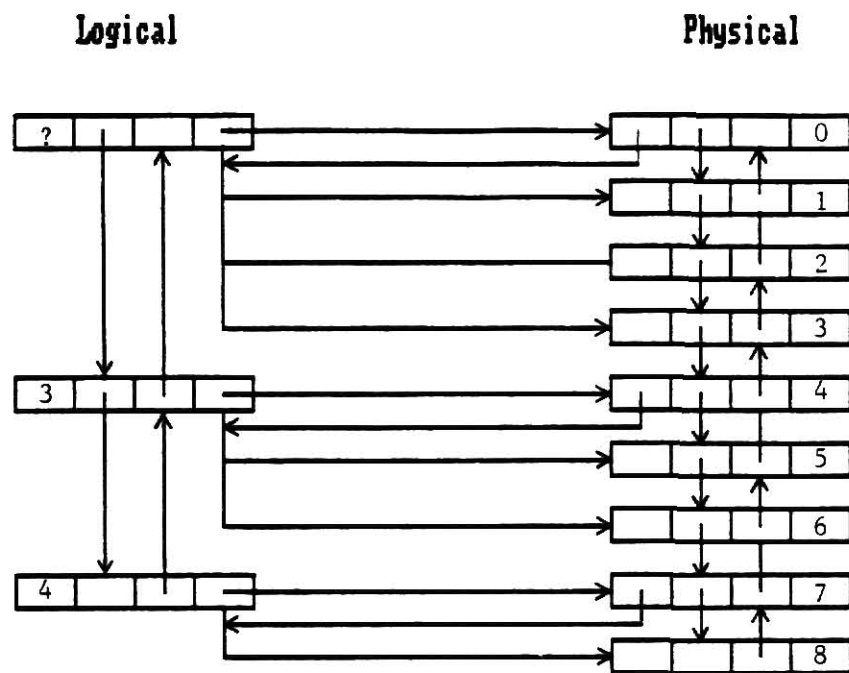


Figure 2.6
Storage System View of Problem

A
A1
A2
A3
A4
A5
A6
B
B1
B2

Figure 2.7 (a)
User View of Text

A
A1
A2
A3
A4
A5
A6
B
B1
B2

Figure 2.7 (b)
System View of Text

Chapter III

Implementation

The storage system consists of approximately 1400 lines of source code and documentation and is comprised of twenty-six procedures which perform the data manipulations. The storage system is divided into two major divisions of code. The major divisions are comprised of procedures and declarations that can be accessed by the editor, the translator, and the analyzer (global), and those that are resident to the storage system (local) and can only be accessed by the storage system code.

The global procedures consist of routines which are the points at which data is transferred. These procedures are: STORE_A_LINE, FETCH_A_LINE, DELETE_A_LINE, WRITE_FILE, WRITE_CLOSE, TRANS_OPEN, TRANS_CLOSE, TRANS_STORE, TRANS_FETCH, METRIC_FETCH, METRIC_LOCATE, METRIC_FETCH, READ_TERM, REFORM, and READ_FILE. These procedures function as communication points at which data is received from, or sent to the command interpreter, editor, translator, and the analyzer. From the procedures which are global to the HLSEW system, the procedures which are local or resident to the storage system are called.

In the next sections which describe the inserting, deleting, changing, and retrieval of text, the logic flow of

each procedure is discussed. The flow paths which take place overlap from procedure to procedure, and the series of procedure calls that are taken may differ only from that of the originating calling statement. Even though physical differences exist between inserting, changing, and deleting of text, the logical differences are quite small.

The main procedures responsible for the inserting, deleting, and changing of text are BLOCK_MOD and STAND_MOD. These procedures contain sections which modify the linked lists. BLOCK_MOD modifies the logical lists where labels are found and STAND_MOD modifies the text found in the physical list. Each of these procedures is comprised of three sections which are responsible for the majority of all the data manipulation which takes place in a file being edited.

Reading Files

At the beginning of an editing session, the Command Interpreter requests the access or the creation of a file by issuing the command to execute READ_FILE.

The initial step that READ_FILE takes is to call FILE_OPEN. FILE_OPEN appends the suffix (filetype) ".C\$\$" onto the file name which has been entered so that the name will conform to the operating system's standards. To guard against the chance that the user has entered the file name with the suffix ".TEXT", the last five characters are checked, and if these are present, the supplied suffix is

removed and the "C\$\$" suffix is appended onto the name. This eliminates the chance of a file name having a suffix or file type ".TEXT.C\$\$".

If the file name that has been entered is found, the physical, and metric files associated with the file name are opened and read. The process of reading the files is monitored by the operating system so that if any errors or system failures should occur, the files are closed and control is given back to the Command Interpreter.

READ_FILE then calls INIT_POINTERS which creates the starting of two linked lists and initializes the pointer values contained in the lists. This initialization includes creating the first and last instance nodes of each list. These nodes act as dummy nodes and any further data manipulation that may take place with a file will be within the bounds of these dummy nodes.

After the execution of OPEN_FILE and INIT_POINTERS, control returns to READ_FILE. The physical file read by OPEN_FILE is used to build the linking that exists internally to the supporting machine. When the building of the internal linking is complete, the control of execution returns to the Command Interpreter. The Command Interpreter is passed information pertaining to whether the file was found, if the named file is new, and the total number of lines contained in the file and, the occurrence of any Input/Output errors. Figure 3.2 diagrams the flow of program control for opening a file.

Retrieval of Text

The retrieval of text is a process which has its calling origins located throughout the editing system. Whether text has been inserted, deleted, or changed, the procedure `FETCH_A_LINE` is responsible for the retrieval of text and is used more extensively than any other procedure within the storage system.

Text retrieval is performed by the procedure `FETCH_A_LINE`. The editing system may process several lines of text before calling the storage system. This leads to the two systems becoming disjoint in relation to where each is logically located within the text file. `LINE_MOVE` is called from `FETCH_A_LINE` and is used to either move up or move down within the file to stay synchronized with the editing system. Synchronizing the storage and editing systems is accomplished by comparing the relational location of the editor's line index against the relational location of the storage system's line index. If the requested line cannot be found, the presence of no text is indicated. When the line of text is found, the contents of the line is retrieved and control returns to the editing system.

Inserting Text

The call to insert a line of text into a file originates in the procedure `STORE_CURRENT_LINE` which is located in the Line Editor. `STORE_A_LINE` serves three different functions; inserting, deleting, and changing text.

`STORE_A_LINE` calls `LINE_MOVE` to move the indexes, which point to the linked lists, to the proper locations. If no errors occur, a check is made to determine if the line of text is a label for a block. If the Storage System determines that a line is a label, the label is tagged, otherwise the line is tagged as a standard line of text. The result of this test will determine the proper insertion sequence. If a label is indicated the label must be inserted into the logical linking first before the text can be inserted into the physical list, this is done by calling `BLOCK_MOD`. If a standard line of text is indicated, then the insertion of the text goes directly to the physical list by calling `STAND_MOD`.

Following the inserting of the text by `STAND_MOD`, the procedure `INSERT_RENUM` is called. If a standard line has been added, the associated block label has its line count incremented. If a label has been inserted, the situation described in Chapter 2, Case 1 of Data Manipulation, occurs. The code must be checked for the existence of a sequence change that may be present after the insertion took place.

Deleting Text

The procedure DELETE_TEXT which is found in the line editor, initiates the deleting of text by a call to DELETE_A_LINE. When examining the procedural flow to delete text, it will be found that the series of procedure executions are virtually identical to that of inserting text. In DELETE_A_LINE, LINE_MOVE is called to find the line of text. If this line is found, STORE_A_LINE is called and the procedural flow used duplicates the sequence used to insert text.

As discussed previously, the various manipulations of text which take place are handled by procedural calls that are in many cases identical. The differences that exist are internal to the procedures themselves.

Once the line of text has been deleted from the linked list, DELETE_RENUM is called. The type of line which was deleted is checked, and if its type indicates a standard line, the associated label has its line count decremented. If a label is indicated, the presence of a sequence interruption is checked for, and then the labels involved are renumbered.

Changing Text

As with inserting text, the calling origin to change a line of text is located in the procedure `STORE_CURRENT_LINE` located in the line editor. The procedural flow is the same as that used to insert a line of text. One important detail to be aware of when changing text is that when given the proper set of conditions, the changing of a line of text may result in almost the same logical conditions as when a line is inserted, or when a line is deleted. This situation is described in Case 4 of Chapter 2.

Writing Files

Once the user decides to end an editing session, control returns to the Command Interpreter in the editing system where the procedure `WRITE_FILE` is called.

The function of `WRITE_FILE` is to store the information of the physical linked list and store that information into a disk file. If at any time an error is received from the operating system, the writing of the files is aborted, the data in memory is lost, and the temporary files are deleted. This action may be objectional to the user that has done a considerable amount of editing, but it guarantees that the original file will be preserved.

Analyzer Data Transfer

From the Command Interpreter, the Software Analyzer is called. Within the Analyzer, requests to the storage system are made for data by means of the procedure METRIC_STORE, METRIC_FETCH, and METRIC_LOCATE. The procedure METRIC_FETCH supplies the Software Analyzer with the text of a specified block of code. The initial call to retrieve the text from a specified block of code begins with the procedure METRIC_LOCATE. METRIC_LOCATE performs a search for the name of a block with the use of the logical linking. If the name is found, the procedure will return to the Software Analyzer a value indicating that the block was found, the total number of lines contained within the block, and the previously calculated, if any, set of metrics. The name of the block will always be the first line of text within that block. If the name is not found, the result is passed to the analyzer. Once a block has been located, calls to METRIC_FETCH will retrieve the associated text. This process will continue until all the text within a block has been transferred. When the analyzer has completed its processing, the metric values are returned to the storage system by calling METRIC_STORE, to store the new set of metrics.

Translator Data Transfer

Execution of the Translator also originates in the Command Interpreter. Because the Translator will process the entire physical text file, a simple sequential file retrieval will take place.

In the Translator, a call to the procedure TRANS_OPEN, TRANS_CLOSE, TRANS_FETCH, TRANS_STORE which are located in the storage system. Unlike editing a file or calculating the metrics of a file, the internal linking of the text file is not formed. The named file is opened and the text is sequentially passed to the translator. TRANS_OPEN functions to open a file which contains text and the results of translated text, TRANS_CLOSE closes the PDL and COBOL files, TRANS_FETCH sends text to the translator, and TRANS_STORE stores the supplied COBOL text.

Memory Management

Checking the amount of memory which has been used is accomplished by calling the procedure MEM_CHECK. MEM_CHECK uses the UCSD Pascal operating system to monitor the host machines' internal memory. This procedure is called from STORE_A_LINE after the completion of any insertion of text.

The value returned by the memory check is compared against an optimal memory value contained within the storage system's constant declarations. If the available free memory is greater than the optimal memory value (1.5K or

15000), then no further actions will occur. When the amount of available memory is in the range of 15K to 16K (15000 to 16000 bytes), a low memory message is displayed by the storage system to the user. But if the available free memory falls below the optimal memory value, the procedure FILE_RELOAD is called and the file being edited is dumped onto the disk, the memory used is deallocated, and the file is reloaded automatically. This method helps flush the system of any unused and discarded data.

In the UCSD Pascal system, once memory is allocated, the deallocation of the memory is difficult. In most cases, but not all, deallocation of memory is an all or nothing situation because arbitrary sections of memory cannot be reclaimed. The internal memory can become consumed by the inserting and deleting of text. A memory deallocation routine is taken by the Storage System to free the memory so that further editing may take place.

System Enhancements

In addition to the storage system and its functions, various features were added to the total HLSEW package. Omitted from the original editing system were facilities to handle operating system errors, file naming procedures, terminal independent coding, and screen display constraints. Additional items added to the system included the design of a formatting front end, and specialized coding to decrease the consumption of internal memory by the system code.

As much overall control of error recovery as possible was taken away from the UCSD Pascal system. This was necessary to insure that the software package would continue the proper operation for which it was designed. The responsibility to control error recovery was shifted from the operating system to the HLSEW system. Soft error reporting is performed rather than allowing the package to be aborted by the controlling operating system.

As stated in this document and also in the Editing System Users Manual, at the beginning of an editing session, a file name is requested. If the file name was not entered, a new file is assumed. At this point, facilities were needed to recover from creating a file as apposed to using an existing one. The facilities added included the procedures needed to name a new file at the end of an editing session.

As documented in the Editing System Users Manual, problems existed with terminal dependent routines which controlled the screen display. To help reduce any terminal dependencies which may occur, a program INSTALL has been supplied. The independent INSTALL program creates a terminal code file which is read by the Command Interpreter. This allows the user to define the characteristics of the terminal that is being used.

When editing a file, text is scrolled forward and backward across the screen. The maximum number of 22 text lines can be displayed on a screen at one time. The initial

design of the editor would allow scrolling either forward or backward to non existing text. This scrolling would cause logic errors from that could not be recovered. To help prevent any logical errors from occurring, code was added to the system so that the exact number of lines in a file is always known. An attempt to access any text beyond the bounds of the file was made impossible.

Information indicates that various machines require a particular format of the file that each will use. These differences are generally related to the characters set found at the end of each line. Some facilities may require each line of text to end with a carriage return, and others may require that the lines of text end with a carriage return/line feed sequence. The Reform selection in the Command Interpreter allows the user to reformat a file into a selected structure. The Reform selection executes the procedure REFORM and reads in a named file and then creates a new file with the specified format.

As stated in the requirements of this project, the most efficient means of using the furnished resources were explored. Figure 3.3 diagrams the system as it appears in module form. Four files exist in the HLSEW system: 1) the Command Interpreter and Screen Editor file, 2) the Editor file, 3) the Storage System file, and 4) the Editing system Declaration file. When the total system is completed, the files which contain the Software Analyzer and the Translator will be included.

In the UCSD Pascal environment, large sections of code can be compiled into groups referred to as "units". These units allow large programs which normally cannot fit into the Pascal editing buffer to be compiled separately and then to be linked together into a complete executable system. An enhanced feature of the UCSD Pascal system called segmenting allows coding of a program in such a way that a segment of code will be resident in memory only during execution. Although this is an expensive process in terms of time used to load the indicated code into memory, it is quite useful for conserving memory by allowing only the necessary code to be present at execution time. This is most often used with initialization routines which may be used only once during the running of the HLSEW system.

As illustrated in Figure 3.3, the Command Interpreter functions as an interface between the user and the Editing System. From the Command Interpreter, the Screen Editor or the Line Editor may be entered. The proper execution of the Screen Editor is dependent upon the presence of the Line Editor, the reverse is not so. The use of segmentation was incorporated into the code of the Screen Editor. With the use of segmentation, the code needed to execute the Screen Editor is not resident to the operating system until the actual execution is to take place. By segmenting the Screen Editor code, the amount of memory that is available to the system is increased.

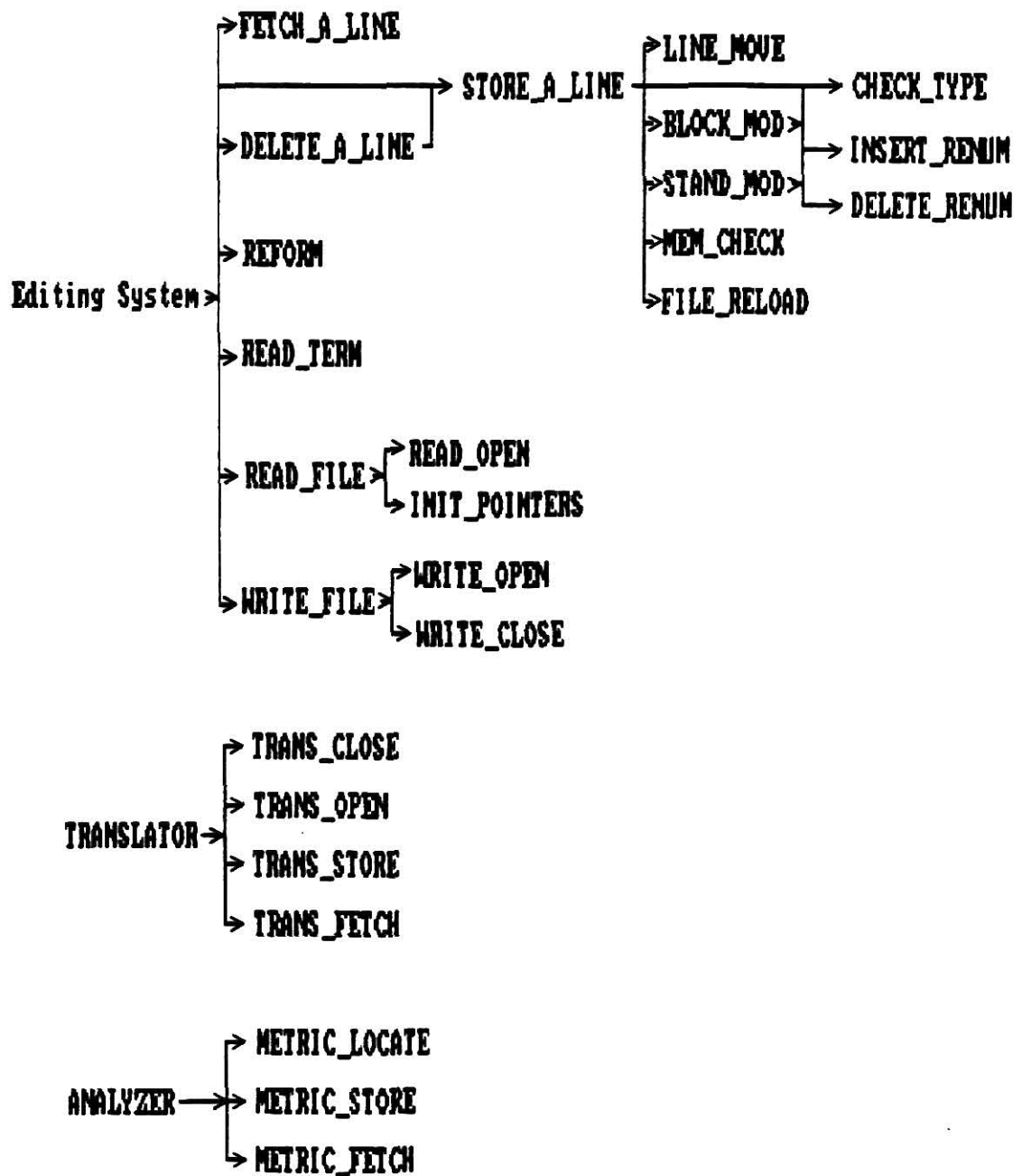


Figure 3.1
Hierarchy Diagram

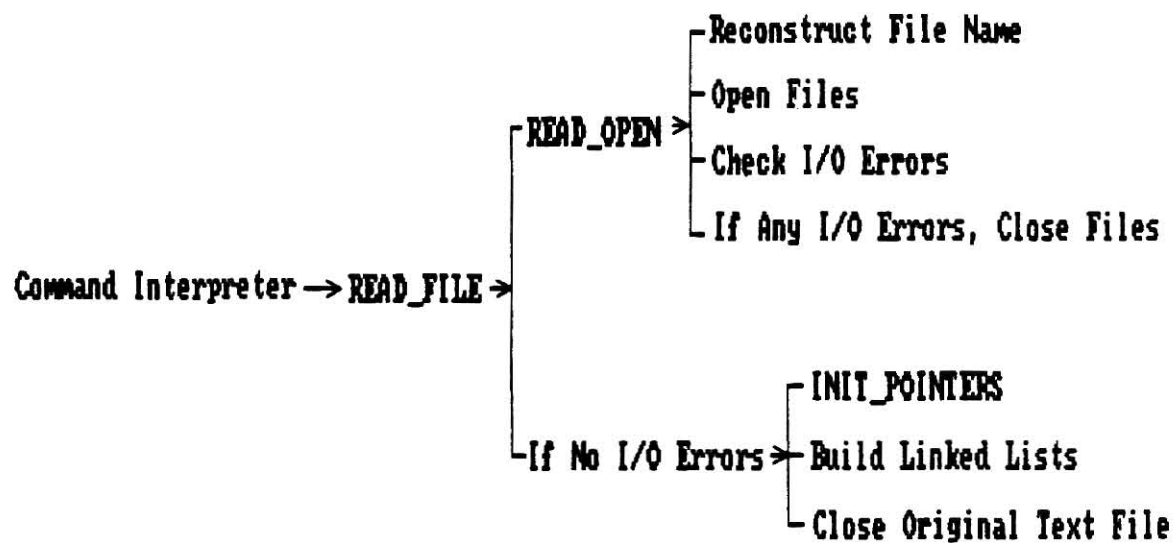
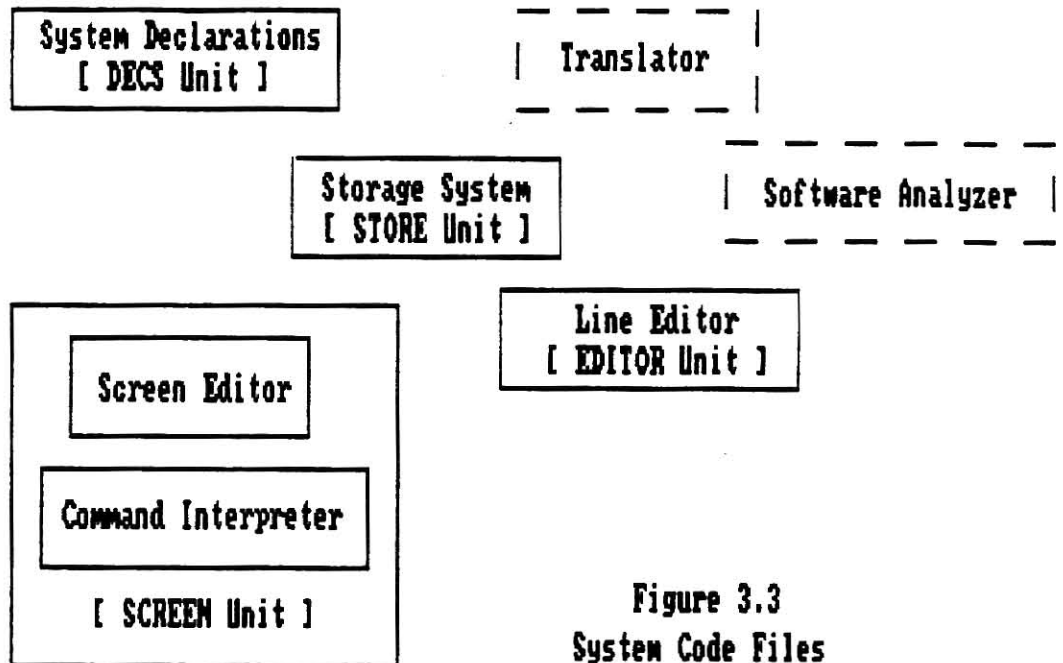


Figure 3.2
Read File Flow



Chapter IV

Testing

The HLSEW Storage System received rigorous and extensive testing. The testing of the storage included (1) the testing of the system as a singular unit, and (2) the testing of the HLSEW as it presently exists. Various aspects of the system were checked for structural and logical correctness. Both static and dynamic testing methods were used.

The system was statically checked by the UCSD compiler for proper system configuration, proper type compatibility, and correct procedural parameter passing. Throughout the development of the system, the code that was written was checked against the original specifications and design to ensure that if any deviation existed, it would be minimal.

Dynamic testing was the most conclusive method of checking for the correctness of the code. The editing system served as the most convenient and logical interface to the testing procedure. Because of the presence of the editing system, the total requirements of the HLSEW could also be inspected.

Before the completion of the storage system, the editing system could only be tested by simulation. Without the use of the storage system, unforeseen coding errors,

logic incompatibles, and data communication errors appeared. Likewise, without the use of the editing system, unforeseen problems appeared in the storage system that otherwise would have gone unnoticed by the designer when testing the code by simulation.

The most effective method of testing the system was to place it under normal operational usage. This included putting into use all the functions available within the editing system. The visual inspection of the results allowed quick and efficient verification of the editing routines. Included in this testing was the attempt to purposely create a system failure at every possible point of processing. Attempting to create a system failure was accomplished by a number of methods such as: trying to edit an old file that does not exist, entering incorrect responses to screen prompts, and attempting to edit non-existent sections of a file. These attempts to cause a system failure verified that error recovery facilities were properly addressed.

Chapter V

Extensions and Future Considerations

Presently there are two major concerns to address in the future for the HLSEW system. To help in the development of a more complete and comprehensive system, additional improvements should be made.

Even though the storage system has been designed to create the files and to store information pertaining to software metrics, the HLSEW is missing the ability to display that information to the user through the editor. The design allows the user to obtain the information about the metrics starting at the Command Interpreter level. The information is then displayed by the Software Analyzer. In the future, there may exist the need to calculate the metrics while still in the editor and then to display the information without disrupting the editing process. If necessary, those metrics are stored and can be recalled without the need for each set to be recalculated.

Minor changes to the storage system could be made so that other languages may be used on the HLSEW. In the storage system the procedure CHECK_TYPE determines the type of line by the starting location of the text. This can be modified to handle free form languages such as Pascal. The modification that must be made should be in the method a

block label is determined. Pascal for instance uses the reserved words BEGIN and END to indicated the bounds of a block. A table of possible labels could be built and a lookup would be performed.

References

- [BY83] M.L.Coffey, 'Local Intelligence For The User', Data Processing Vol.25, No.4, pp. 31-32 May 1983
- [C082] M.L.Coffey, 'Intelligent Terminals: the best of both worlds'. Datamation Vol.28, No.4, pp 104-106 April 1982
- [DJ82] D. Julian and M. Davies, 'String Searching in Text Editors', Software-Practice and Experience, Vol.12, pp. 709-717, 1982
- [MD82] D. Julian and M. Davies, 'A Note on Sparsely Filled Dynamically Allocated Memory', The Computer Journal, Vol.25, No.1, p. 159, January 1982
- [ML82] M. Levison, 'A Programmable Text-editing System', Software-Practice and Experience, Vol.12, pp. 611-621, 1982
- [TE82] T. Teitelbaum, 'On the Value of Syntax-Directed Editors', Communication of the ACM, Vol.25, No.5, pp. 351-352, May 1982
- [MA81] A.V.Magors and L.Westerfeld, 'Student Intelligent Terminal System', AEDS-81 Convention Proceedings, Assoc. Educ. Data Syst., pp 190-194, 1981
- [RS81] R.S. Scowen, 'A Survey of Some Text Editors', Software-Practice and Experience, Vol.2, pp. 883-906, 1981
- [TC81] T.A. Cargill, 'Full-Screen Editing in a Hostile Environment', Software-Practice and Experience, Vol.2, pp. 975-981, 1981
- [TE81] T. Teitelbaum and T. Reps, 'The Cornell Program Synthesizer: A Syntax-Directed Programming Environment', Communications of the ACM, Vol.24, No.9, pp. 563-573 September 1981
- [RH80] R.N. Horspool, 'Practical Fast Searching In Strings', Software-Practice and Experience, Vol.10, pp. 501-506, 1980
- [RE79] C.M. Reeves, 'Free Store Distribution Under Random Fit Allocation, Part 1', The Computer Journal, Vol.22, pp. 346-351, 1979
- [RE78] C.M. Reeves, 'Free Store Distribution Under Random Fit Allocation, Part 2', The Computer Journal, Vol.23, pp. 298-306, 1980

[VL76] P.Van Leer, 'Top-Down development using a program design language', IBM Systems Journal, Vol.15, No.2, pp. 155-172, 1976

[WI76] T.R. Wilcox, A.M. Davis, M.H. Tindall, 'The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System', Communications of the ACM, Vol.19, No.11, pp.609-619, 1976

[CA75] E.K.Gordon and S.H.Caine, 'PDL- A Tool For Software Design', Proc. AFIPS 1975 NCC, pp 271-276

[GO75] E.K. Gordon, 'Experience and Accomplishments with Structured Programming', Computer, pp.50-53, June 1975

[BA79] R.M. Balzer, 'EXDAMS-EXTendable Debugging and Monitoring System, AFIPS Proc., Vol.34, No.24, pp. 567-580, 1969

Appendix I

Procedure Specifications

Procedure: / MEM_CHECK /

Input Data Item: None

Output Data Item: MEM_OK Boolean

Description: procedure responsible for checking the status of the memory which is available to the HLSEW system.

Comments: accesses the operating system to compare the remaining free memory against the optimal amount of free memory.

Procedure: / READ_TERM /

Input Data Item: None

Output Data Item: LEFT, DOWN, UP, RIGHT Integer
CLRL, CLRS, CLRP Key_Code
CURSOR_LEAD Integer
LEAD_IN, IO_OK Boolean

Description: reads into the HLSEW system, the terminal codes stored in the file 'TERMCODE.TXT'.

Comments: enters terminal codes which were first stored by the terminal installation routine.

Procedure: / TRANS_OPEN/

Input Data Item: FILE_NAME Name_type

Output Data Item: IO_OK Boolean

Description: procedure responsible for opening the translated PDL file which contains COBOL code.

Comments: I/O result is checked for any error in opening a file and a file pointer is placed at the beginning of linked list.

Procedure: / TRANS_CLOSE /

Input Data Item: None

Output Data Item: CLOSE_OK Boolean

Description: procedure to close the COBOL code file.

Comments: the value of the boolean CLOSE_OK will determine whether the COBOL code file will be close and saved or to be closed and deleted. If CLOSE_OK is true the code file is closed and saved, if CLOSE_OK is false the code file is close and deleted.

Procedure: / TRANS_STORE /

Input Data Item: TEXT_LINE Line
IO_OK Boolean

Output Data Item: None

Description: procedure to write the translated code to the respective COBOL code file.

Comments: I/O checks are passed back to the translator by way of IO_OK.

Procedure: / TRANS_FETCH /

Input Data Item: TEXT_LINE Line

Output Data Item: IO_OK, TRANS_DONE Boolean

Description: sequentially retrieves text from the PDL work file and sends each line to the translator.

Comments: TRANS_DONE is given the value of true when the end of the PDL file has been located. IO_OK indicates whether any I/O errors are detected.

Procedure: / METRIC_STORE /

Input Data Item: METRICS_IN Metrics

Output Data Item: IO_OK Boolean

Description: procedure stores the set of metrics calculated by the metric interpreter.

Comments: the address of the set of metrics is located in the physical linked list.

Procedure: / METRIC_FETCH /

Input Data Item: None

Output Data Item: M_LINE_OUT Copy_Rec
METRICS_FOUND, IO_OK Boolean

Description: sequentially retrieves lines of text from the temporary file for the metric analyzer.

Comments: M_LINE_OUT is assigned the value of each line of PDL text and passed back to the metric analyzer. If the line of PDL text is not found the value of METRICS_FOUND is set to false. If any io error should occur, the value of IO_OK is set to false.

Procedure: / METRIC_LOCATE /

Input Data Item: ID_NAME Copy_Rec

Output Data Item: METRICS_OUT Metrics
ID_FOUND, METRICS_FOUND Boolean
LINE_TOTAL Integer

Description: locates the position in the linked list of the block label that corresponds with the value of ID_NAME. The address of the metric values are read from the metric file and sent to the Metric Analyzer.

Comments: the total number of lines found in a block of text is also sent back to the Metric Analyzer to indicate the number of calls to the procedure METRIC_FETCH.

Procedure: / READ_OPEN /

Input Data Item: FILE_NAME Name_Type

Output Data Item: FILE_FOUND Boolean
NEW_FILE Boolean

Description: procedure is responsible for the proper opening and closing of the temporary text and metric files.

Comments: if there is any indication that an error has occurred when opening the temporary or metric files, the files are closed and an error message is returned to the calling procedure.

Procedure: / INIT_POINTERS /

Input Data Item: None

Output Data Item: None

Description: the purpose of this procedure is to form the the basic heading and trailing nodes of the linked list.

Comments: the values in the nodes are initialized so that no erroneous values will be detected.

Procedure: / READ_FILE /

Input Data Item: FILE_NAME Name_Type

Output Data Item: FILE_FOUND, NEW_FILE Boolean
TOTAL_LINES Integer

Description: the basic function of this procedure is to call the procedures READ_OPEN and INIT_POINTERS. The text found in the requested text file is read into the linked lists and the logical and physical structures of those linked lists are constructed.

Comments: the text found in a text file is stored in blocks of 512 bytes where each line has been stripped of all trailing blanks. When the text is read, each line of text is broken into physical lines of text which contain 80 characters each.

Procedure: / WRITE_OPEN /

Input Data Item: FILE_NAME Name_Type

Output Data Item: IO_OK Boolean

Description: procedure responsible for opening the original text file in preparation to save an edited file.

Comments: if any errors should occur, and error is returned to the calling procedure.

Procedure: / WRITE_CLOSE /

Input Data Item: IO_OK Boolean

Output Data Item: None

Description: procedure responsible for the closing of the saved edit file.

Comments: if the value of IO_OK that has been sent to this procedure is TRUE the edited files are saved. If the value of IO_OK is FALSE the files are close and deleted.

Procedure: / WRITE_FILE /

Input Data Item: FILE_NAME Name_Type

Output Data Item: IO_OK

Description: procedure retrieves the text found in the linked list and writes out the text in 512 bytes. Any detected I/O errors are sent back to the calling procedure by way of IO_OK.

Comments: each line of text is stripped of trailing blanks before being written to disk.

Procedure: / INSERT_RENUM /

Input Data Item: LN_TYPE Line_Type

Output Data Item: None

Description: procedure updates the total number of lines located in each logical block.

Comments: called after each line insertion has been made.

Procedure: / DELETE_RENUM /

Input Data Item: LN_TYPE Line_Type

Output Data Item: None

Description: procedure updates the total number of lines located in each logical block.

Comments: called after each line deletion has been made.

Procedure: / CHECK_TYPE /

Input Data Item: LINE_IN Copy_Rec

Output Data Item: LN_TYPE Line_Type

Description: procedure checks physical location of text found in LINE_IN. The location of the text determines whether the input text line is a standard line of text or the label of a block of text.

Comments: BLOCK_LINE indicates that a label is present, STAND_LINE indicates that a standard line of text is present.

Procedure: / STAND_MOD /

Input Data Item: LINE_IN Copy_Rec
MODE_IN Mode_Type
LN_TYPE Line_Type
DISK_OK Boolean

Output Data Item: None

Description: procedure is responsible for the actual inserting, changing, and deleting of the text in the physical linked list. Procedure also functions to monitor errors which may occur when writing to disk.

Comments: changes the contents of the physical linking either by adding a new text node, changing the contents of a text node, or by deleting a text node. Disk I/O result are assigned to DISK_OK for use by calling procedure.

Procedure: / BLOCK_MOD /

Input Data Item: LINE_IN Copy_Rec
MODE_IN Mode_Type
LN_TYPE Line_Type

Output Data Item: None

Description: procedure is responsible for the actual inserting, changing, and deleting of text in the logical linked list. Procedure also functions to monitor errors which may occur when writing to disk.

Comments: changes the contents of the logical linking either by adding a new text node, changing the contents of a text node, or by deleting a text node. Disk I/O results are assigned to DISK_OK for use by calling procedure.

Procedure: / LINE_MOVE /

Input Data Item: INPUT_LINE_NO Integer

Output Data Item: None

Description: procedure uses the supplied line number, INPUT_LINE_NUMBER, to place the storage system line pointer in sequence with the line the editor is pointing to.

Comments: if the input line number is not found, an error indication is sent back to the calling procedure.

Procedure: / FILE_RELOAD /

Input Data Item: NONE

Output Data Item: None

Description: procedure responsible for saving temporary files and reclaiming memory used by editing a text file.

Comments: if no file name exists, the file is written on disk with the name of 'HLSEW.WRK.TXT'.

Procedure: / STORE_A_LINE /

Input Data Item: LINE_IN Line
LINE_NO Integer
MODE_IN Mode_Type

Output Data Item: None

Description: procedure controls the sequence of procedural calls that is responsible for the insertion and changing of text in each of the linked lists.

Comments: this procedure is called from the editing system.

Procedure: / FETCH_A_LINE /

Input Data Item: INPUT_LINE_NO Integer

Output Data Item: LINE_OUT Line
LINE_FOUND Boolean
NEW_FILE Boolean

Description: procedure retrieves the text indicated by
INPUT_LINE_NO from the physical linking.

Comments: if the requested line number is not found,
FOUND_IT is assigned FALSE. If NEW_FILE is
TRUE, no execution takes place.

Procedure: / DELETE_A_LINE /

Input Data Item: INPUT_LINE_NO Integer

Output Data Item: LINE_FOUND

Description: procedure controls the sequence of calls
that is responsible for the deletion of text
within the linked lists.

Comments: if the requested line number is not found,
LINE_FOUND is assigned a FALSE value and
returned to the calling procedure in the
editing system.

Appendix II

Parameter Specifications

Data Type Summary

<u>Data Type</u>	<u>Description</u>
KEY_CODE	Array of 1 to 6 Characters
COPY_REC	Array of 1 to 64 Characters
NAME_TYPE	String (Array) of 1 to 15 Characters
LINE	Array of 1 to 80 Characters
LINE_TYPE	BLOCK_LINE/STAND_LINE enumeration
METRICS	Record of integer values (tobe designed at a later date)
MODE_TYPE	DELETE_MODE/CHANGE_MODE/INSERT_MODE enumeration

Procedure Parameter Description and Summary

CURSOR_LEAD

Description: KEY_CODE type, a numeric code sequence used as the lead-in to all the cursor control codes.

Produced By: READ_TERM

Used By: Editing System

CLRL

Description: KEY_CODE type, a numeric code sequence used to clear a line of text on the screen.

Produced By: READ_TERM

Used By: Editing System

CLRS

Description: KEY_CODE type, a numeric code sequence used to erase any text on the screen.

Produced By: READ_TERM

Used By: Editing System

CLRP

Description: KEY_CODE type, a numeric code sequence used to erase any text on the screen from the cursor position to the end of the screen.

Produced By: READ_TERM

Used By: Editing System

CLOSE_OK

Description: BOOLEAN type, a TRUE/FALSE indicator used to determine the type of action TRANS_CLOSE will take. A TRUE value signals the saving of the edited file and a FALSE value signals the files to be deleted.

Produced By: TRANSLATOR

Used By: TRANS_CLOSE

DISK_OK

Description: BOOLEAN type, a TRUE/FALSE indicator used to signal the storage that an error occurred when a write to the disk was made.

Produced By: STAND_MOD, BLOCK_MOD

Used By: STORE_A_LINE

DOWN

Description: KEY_CODE type, a numeric code sequence used to move the cursor one position down on the screen.

Produced By: READ_TERM

Used By: Editing System

FILE_NAME

Description: NAME_TYPE type, a character string which contains the name of the file to be opened.

Produced By: Editing System

Used By: TRANSLATOR, READ_OPEN, TRANS_OPEN,
WRITE_OPEN READ_FILE, WRITE_FILE.

FILE_FOUND

Description: BOOLEAN type, a TRUE/FALSE indicator used to signal whether the requested file name was found on a disk directory. If a TRUE value is present, the requested file was found.

Produced By: READ_OPEN

Used By: Editing System, READ_OPEN

IO_OK

Description: BOOLEAN type, a TRUE/FALSE value indicator used throughout the HLSEW System that is assigned the value of any I/O result. A TRUE value indicates that I/O error was detected.

Produced By: READ_TERM, TRANS_OPEN, TRANS_STORE,
TRANS_FETCH, METRIC_STORE, METRIC_FETCH,
WRITE_OPEN.

Used By: TRANSLATOR, ANALYZER, WRITE_FILE,
WRITE_CLOSE, Editing System.

ID_FOUND

Description: BOOLEAN type, a TRUE/FALSE indicator signaling whether an input ID Name was found in the linked list. TRUE represents that the ID was found.

Produced By: METRIC_LOCATE

Used By: ANALYZER

ID_NAME

Description: LINE type, a character string which contains the ID Name of the block of text to be searched for by the procedure METRIC_LOCATE.

Produced By: ANALYZER

Used By: METRIC_LOCATE

INPUT_LINE_NO

Description: INTEGER type, a numeric value representing the logical line number to which the procedure LINE_MOVE is to position the read pointer.

Produced By: Editing System

Used By: LINE_MOVE

LEFT

Description: INTEGER type, a numeric code used to move the cursor one position to the left on the screen.

Produced By: READ_TERM

Used By: Editing System

LEAD_IN

Description: BOOLEAN type, a TRUE/FALSE indicator used by the Editing System to signal whether a lead-in character is to be expected when sending cursor control commands. A TRUE value indicates that a lead-in code will be used.

Produced By: READ_TERM

Used By: Editing System

LINE_OUT

Description: COPY_REC type, a character string that contains the text retrieved from the temporary edit file.

Produced By: FETCH_A_LINE

Used By: Editing System

LINE_TOTAL

Description: INTEGER type, a numeric value which represents the number of lines that can be found in an indicated block of text. This value is stored in the logical linking of the Storage System and is used by the Analyzer to determine the number of text retrieval calls that should be made to the procedure METRIC_FETCH.

Produced By: METRIC_LOCATE

Used By: ANALYZER

LN_TYPE

Description: LINE_TYPE type, an indicator used to determine the type of processing which is to take place. The value of LINE_TYPE can be either BLOCK_LINE or STAND_LINE. BLOCK_LINE represents that a Block label is to be processed, and STAND_LINE represents that a standard line of text is to be processed.

Produced By: CHECK_TYPE

Used By: INSERT_RENUM, DELETE_RENUM, STAND_MOD, BLOCK_MOD.

LINE_IN

Description: COPY_REC type, contains the character string value to be stored or changed.

Produced By: Editing System

Used By: CHECK_TYPE, STAND_MOD, BLOCK_MOD, STORE_A_LINE.

LINE_FOUND

Description: BOOLEAN type, a TRUE/FALSE indicator used to signal whether an indicated line number was found in the linked list. A TRUE value indicates that the line number was found.

Produced By: LINE_MOVE

Used By: Editing System, FETCH_A_LINE.

LINE_NO

Description: INTEGER type, a numeric value containing the logical number of the line to stored or changed

Produced By: Editing System

Used By: LINE_MOVE, STORE_A_LINE.

LINE_REC

Description: COPY_REC type, a character string that contains the text retrieved from the temporary edit file.

Produced By: FETCH_A_LINE

Used By: Editing System

METRICS_IN

Description: METRICS type, a data record which contains the numeric and/or character values calculated by the ANALYZER.

Produced By: METRIC_STORE

Used By: ANALYZER

M_LINE_OUT

Description: LINE type, a character string which contains the text that has been retrieved from the temporary text file.

Produced By: METRIC_FETCH

Used By: ANALYZER

METRICS_FOUND

Description: BOOLEAN type, a TRUE/FALSE value indicator used to signal that the requested line of text was located.

Produced By: METRIC_FETCH

Used By: ANALYZER

METRICS_OUT

Description: METRICS type, a record which contains the numeric and/or character information calculated by the ANALYZER.

Produced By: METRIC_LOCATE

Used By: ANALYZER

MODE_IN

Description: MODE_TYPE type, a DELETE_MODE/CHANGE_MODE/INSERT_MODE value indicating the type of line that is to be processed by the Storage System.

Produced By: Editing System

Used By: STAND_MOD, BLOCK_MOD, STORE_A_LINE, DELETE_A_LINE.

NEW_FILE

Description: BOOLEAN type, a TRUE/FALSE value indicating the creation of a new file. TRUE indicates that a new edit file is being processed.

Produced By: READ_OPEN

Used By: FETCH_A_LINE, Editing System, READ_FILE

RIGHT

Description: INTEGER type, a numeric code used for moving the cursor one position to the right on the screen.

Produced By: READ_TERM

Used By: Editing System

TEXT_LINE

Description: LINE type, a character string which contains the text that has been retrieved from the PDL text file.

Produced By: TRANS_STORE, TRANS_FETCH

Used By: TRANSLATOR

TRANS_DONE

Description: BOOLEAN type, a TRUE/FALSE value indicator used to signal that the end of the linked list has been located and that processing can halt.

Produced By: TRANS_FETCH

Used By: TRANSLATOR

TOTAL_LINES

Description: INTEGER type, a numeric value containing the total number of text lines read into the temporary file to be edited.

Produced By: READ_FILE

Used By: Editing System

UP

Description: INTEGER type, numeric code sequence used for moving the cursor one position up on the screen.

Produced By: READ_TERM

Used By: Editing System

Appendix III

HLSEW Declaration Source Code

```

(*****
*
*      DDDDDDDD      EEEEEEEEE      CCCCCCCCC      SSSSSSSSS
*      DDDDDDDDDD      EEEEEEEEE      CCCCCCCCC      SSSSSSSSS
*      DDD   DDD      EEE           CCC           SSS
*      DDD   DDD      EEE           CCC           SSS
*      DDD   DDD      EEEEEEEEE      CCC           SSS
*      DDD   DDD      EEEEEEEEE      CCC           SSS
*      DDD   DDD      EEE           CCC           SSS
*      DDD   DDD      EEE           CCC           SSS
*      DDDDDDDDDD      EEEEEEEEE      CCCCCCCCC      SSSSSSSSS
*      DDDDDDDD      EEEEEEEEE      CCCCCCCCC      SSSSSSSS
*
*
*      DECLARATION UNIT DESIGNED BY RUSSELL J. HOLT
*
*
*
*****)

```

UNIT DECS;

```

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

```

CONST LINE_LENGTH = 80;

LOG_ON_MSG = ' HLSEW EDITOR';

HELP_MSG = 'Type Help (HE) For A Summary Of Commands';

NAME_MAX = 15;

```

SOH = 2;      ETX = 3;      BEL = 7;
HT = 9;      CR = 13;      ESC = 27;
SPACE = 32;   DEL = 127;    LF = 10;

```

TYPE LINE = ARRAY [1..LINELENGTH] OF CHAR;

KEY_CODE = PACKED ARRAY [1..6] OF INTEGER;

TABSETTING = SET OF 1..80;

REFORM_TYPE = (CR_ONLY, LF_CR);

COMMAND_TYPE = (INSERT, DELETE_IT, CHANGE, SETTABS, TABCHAR,
VERIFY, LIST, FETCH_IT, STORE_IT, REPEAT_IT,
HELP, ENDEDIT, APPEND, BADCOMMAND, EDIT_IT);

ERROR_TYPE = (COMMAND_ERROR, NOT_FOUND, STRING_NOT_FOUND,
WRITING, LONGLINE, OTHER_ERROR, CHAR_ERROR,
TRANS_ERROR, UPDATING, ARGUMENT_ERROR,
REFORM_ERROR);

TOKEN_TYPE = (NILTOK, LINENOTOK, QTHERTOK);

```

TOKEN = RECORD
    TOKEN_KIND : TOKEN_TYPE;
    VALUE : INTEGER;
END;

```

```

MODE_TYPE = ( DELETE_MODE, CHANGE_MODE, INSERT_MODE );

NAME_TYPE = STRING [ NAME_MAX ];

SET_OF_VALID = SET OF CHAR;

VAR  COMMAND : COMMAND_TYPE;
     FILE_NAME : NAME_TYPE;
     TABS : TABSETTING;
     INPUT_LINE, TEMP_LINE : LINE;
     LINE_INDEX, TEMP_LENGTH, LINE_NUMBER : INTEGER;
     TOTAL_LINES, ROW_MARK : INTEGER;
     ROW, COLUMN, SAVE_ROW, SAVE_COLUMN : INTEGER;
     LEFT, DOWN, UP, RIGHT, CURSOR_LEAD : INTEGER;
     CLRL, CLRS, CLRP : KEY_CODE;
     VERIFY_CHANGES, FINISHED, EDITING_FROM_SCREEN : BOOLEAN;
     FILE_CHANGED, NEW_FILE, LEAD_IN : BOOLEAN;
     TAB_CHARACTER, NL, SPACE_BAR : CHAR;

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

END. ( unit decs )

```

Appendix IV

Storage System Source Code

```

(*)*****
*
*      SSSSSSSS      TTTTTTTTT      0000000      RRRRRRR      EEEEEEEEE      *
*      SSSSSSSS      TTTTTTTTT      000000000      RRRRRRRRR      EEEEEEEEE      *
*      SSS      TTT      000      000      RRR      RRR      EEE      *
*      SSSS      TTT      000      000      RRR      RRRR      EEE      *
*      SSSS      TTT      000      000      RRRRRRR      EEEEEEEEE      *
*      SSSS      TTT      000      000      RRRRR      EEEEEEEEE      *
*      SSSS      TTT      000      000      RRRRRRR      EEE      *
*      SSSS      TTT      000      000      RRR      RRR      EEE      *
*      SSSSSSSS      TTT      0000000      RRR      RRR      EEE      *
*      SSSSSS      TTT      00000      RRR      RRR      EEEEEEEEE      *
*
*
*
*      DISK STORAGE SYSTEM DESIGNED BY RUSSELL J. HOLT
*
(*)*****

```

UNIT STORE;

(%U #9:DECS.CODE)

(*****)

INTERFACE

(*****)

USES DECS;

(*****)

INTERFACE PROCEDURES

(*****)

CONST ID_MAX = 30;

STORE_LENGTH = 64;

LABEL_END = 11;

LABEL_START = 8;

BLOCK_SIZE = 512;

TYPE METRICS = RECORD

()

END;

PHYS_REC = FILE;

COPY_REC = ARRAY [1..STORE_LENGTH] OF CHAR;

TRANS_REC = ARRAY [1..LINE_LENGTH] OF CHAR;

COPY_TYPE = FILE OF COPY_REC;

METRIC_FILE = FILE OF METRICS;

TERM_FILE = FILE OF KEY_CODE;

VAR P_FILE, R_FILE : PHYS_REC;

COPY_P : COPY_TYPE;

TR_FILE : TRANS_REC;

TM_FILE : TERM_FILE;

M_FILE : METRIC_FILE;

PROCEDURE REFORM

(VAR FILE_FOUND, IO_OK : BOOLEAN;

IN_FILE, OUT_FILE : NAME_TYPE;

REF_TYPE : REFORM_TYPE);

PROCEDURE READ_TERM

(VAR LEFT, DOWN, UP, RIGHT : INTEGER;

```

        VAR CLRL, CLRS, CLRP : KEY_CODE;
        VAR CURSOR_LEAD : INTEGER;
        VAR LEAD_IN, IO_OK : BOOLEAN );

PROCEDURE TRANS_OPEN      ( FILE_NAME : NAME_TYPE;
                           VAR IO_OK : BOOLEAN );

PROCEDURE TRANS_CLOSE    ( VAR CLOSE_OK : BOOLEAN );

PROCEDURE TRANS_STORE    ( VAR TEXT_IN : LINE;
                           VAR IO_OK : BOOLEAN );

PROCEDURE TRANS_FETCH    ( VAR TEXT_OUT : LINE;
                           VAR IO_OK : BOOLEAN;
                           VAR TRANS_DONE : BOOLEAN );

PROCEDURE METRIC_STORE   ( VAR IO_OK : BOOLEAN;
                           METRICS_IN : METRICS );

PROCEDURE METRIC_FETCH   ( VAR M_LINE_OUT : COPY_REC;
                           VAR METRICS_FOUND,
                           IO_OK : BOOLEAN );

PROCEDURE METRIC_LOCATE  ( VAR METRICS_OUT : METRICS;
                           VAR ID_FOUND : BOOLEAN;
                           VAR LINE_TOTAL : INTEGER;
                           VAR METRICS_FOUND : BOOLEAN;
                           ID_NAME : COPY_REC );

PROCEDURE READ_FILE      ( VAR FILE_FOUND : BOOLEAN;
                           FILE_NAME : NAME_TYPE;
                           VAR TOTAL_LINES : INTEGER;
                           VAR NEW_FILE : BOOLEAN );

PROCEDURE WRITE_FILE     ( VAR IO_OK : BOOLEAN;
                           FILE_NAME : NAME_TYPE );

PROCEDURE WRITE_CLOSE    ( VAR IO_OK : BOOLEAN );

PROCEDURE STORE_A_LINE   ( LINE_REC : LINE;
                           LINE_NO : INTEGER;
                           MODE_IN : MODE_TYPE );

PROCEDURE FETCH_A_LINE   ( INPUT_LINE_NO : INTEGER;
                           VAR LINE_REC : LINE;
                           LINE_FOUND : BOOLEAN;
                           NEW_FILE : BOOLEAN );

PROCEDURE DELETE_A_LINE  ( INPUT_LINE_NO : INTEGER;
                           VAR LINE_FOUND : BOOLEAN );

{*****}
IMPLEMENTATION
{*****}

CONST BLANK = ' ';
      PERIOD = '.';
      BUF_SIZE = 512;
      OPTIMAL = 15000;
      LOW_OPT = 1000;

```

```

TYPE LINE_TYPE = ( BLOCK_LINE, STAND_LINE );
RENUM_TYPE = ( ADDBLOCK, DELBLOCK );
L_POINTER = LOGICAL;
P_POINTER = PHYSICAL;
LOGICAL = RECORD
    BLK_LNS      : INTEGER;
    METRIC_ADRS  : INTEGER;
    NEXT_L       : L_POINTER;
    PREV_L       : L_POINTER;
    PHYS_PTR     : P_POINTER;
END;
PHYSICAL = RECORD
    PHYS_ADRS    : INTEGER;
    NEXT_P       : P_POINTER;
    PREV_P       : P_POINTER;
    LOG_PTR      : L_POINTER;
END;

VAR LOG_FIRST, L_PTR, LAST_L, METR_PTR : L_POINTER;
    PHYS_FIRST, P_PTR, LAST_P, TR_PTR  : P_POINTER;
    P_INDEX, M_INDEX, LINE_COUNT : INTEGER;
    METRIC_REC : METRICS;
    RAM_SYS : BOOLEAN;
    BACKUP_NAME : NAME_TYPE;

(*****)

PROCEDURE REFORM; ( VAR FILE_FOUND, IO_OK : BOOLEAN;
                   IN_FILE, OUTFILE : NAME_TYPE;
                   REF_TYPE : REFORM_TYPE )

VAR RESULT_IN, RESULT_OUT, OK, LINE_DONE : BOOLEAN;
    BUFF : COPY_REC;
    BUFF_INDEX, TEMP_CHAR : CHAR;
    I, BUFF_COUNT : INTEGER;

BEGIN

    BUFF_INDEX := 0;
    LINE_DONE := FALSE;
    BUFF_COUNT := 0;
    RESET ( P_FILE, IN_FILE );
    RESULT_IN := IO_RESULT = 0;
    REWRITE ( R_FILE, OUT_FILE );
    RESULT_OUT := IO_RESULT = 0;
    IO_OK := RESULT_IN AND RESULT_OUT;
    IF IO_OK
    THEN BEGIN
        FILE_FOUND := TRUE;
        TEMP_CHAR := P_FILE^;
        WHILE ( IO_RESULT = 0 ) AND ( NOT EOF ( P_FILE ) ) DO
            BEGIN
                WHILE ( TEMP_CHAR <> EOF ( P_FILE ) ) AND
                    ( IO_RESULT = 0 ) DO
                    BEGIN
                        WHILE ( TEMP_CHAR <> EOF ( P_FILE ) ) AND
                            ( NOT LINE_DONE ) AND
                            ( IO_RESULT = 0 ) DO
                        BEGIN

```



```

        IF ( TEMP_CHAR = CHR ( NL ) ) OR
            ( TEMP_CHAR = CHR ( LF ) ) OR
            ( TEMP_CHAR = CHR ( 0 ) )
        THEN LINE_DONE := TRUE
        ELSE BEGIN
            BUFF [ BUFF_INDEX ] := TEMP_CHAR;
            BUFF_INDEX := SUCC ( BUFF_INDEX );
            GET ( P_FILE );
            TEMP_CHAR := FILE^
        END;
        IF REF_TYPE = LF_CR
        THEN BEGIN
            BUFF_INDEX := SUCC ( BUFF_INDEX )
            BUFF [ BUFF_INDEX ] := CHR ( CR );
        END;
        IF ( TOTAL_CHAR + BUFF_COUNT ) >= BLOCK_SIZE
        THEN BEGIN
            FOR I := ( BUFF_COUNT + 1 ) TO BLOCK_SIZE DO
            BEGIN
                R_FILE^ := CHR ( 0 );
                PUT ( R_FILE )
            END;
            BUFF_COUNT := 0
        END;
        FOR I := 1 TO TOTAL_CHAR DO
        BEGIN
            R_FILE^ := BUFF [ I ];
            PUT ( R_FILE )
        END;
        BUFF_COUNT := BUFF_COUNT + TOTAL_CHAR;
        TOTAL_CHAR := 0;
        BUFF_INDEX := 0;
        LINE_DONE := FALSE
    END
END
END
END
END;
(*****)

PROCEDURE MEM_CHECK ( VAR MEM_OK : BOOLEAN );

    VAR AVAIL : REAL;
        I : INTEGER;

BEGIN
    IF MEMAVAIL < 0
    THEN AVAIL := 65536.0 + MEMAVAIL
    ELSE AVAIL := MEMAVAIL;
    IF AVAIL < OPTIMAL
    THEN BEGIN
        MEM_OK := TRUE;
        IF AVAIL < ( LOW_OPT + MEMAVAIL )
        THEN BEGIN
            GOTOXY ( 0, 24 );
            FOR I := 1 TO 6 DO
                WRITE ( CLRL [ I ] );
            GOTOXY ( 0, 24 );
            WRITE ( CHR ( BEL ), 'Memory Low, Please Save File' )
        END
    END
END

```

```

        END
    END
    ELSE MEM_OK := FALSE
END;

(*****)

PROCEDURE READ_TERM: ( VAR LEFT, DOWN, UP, RIGHT : INTEGER;
                      VAR CLRL, CLRS, CLRP : KEY_CODE;
                      VAR CURSOR_LEAD : INTEGER;
                      VAR LEAD_IN, IO_OK : BOOLEAN )

    VAR LEFT_CODE, DOWN_CODE, UP_CODE, RIGHT_CODE, RAM_CODE : KEY_CODE;

BEGIN
    ($I-)
    RESET ( TM_FILE, CONCAT ( '#9:', 'TERMCODE.DATA' ) );
    IO_OK := IORESULT = 0;
    IF IO_OK
    THEN BEGIN
        LEFT_CODE := TM_FILE^;
        GET ( TM_FILE ); DOWN_CODE := TM_FILE^;
        GET ( TM_FILE ); UP_CODE := TM_FILE^;
        GET ( TM_FILE ); RIGHT_CODE := TM_FILE^;
        GET ( TM_FILE ); CLRL := TM_FILE^;
        GET ( TM_FILE ); CLRS := TM_FILE^;
        GET ( TM_FILE ); CLRP := TM_FILE^;
        GET ( TM_FILE ); RAM_CODE := TM_FILE^;
        CLOSE ( TM_FILE );

        IF RAM_CODE [ 1 ] = 1 THEN RAM_SYS := TRUE;

        IF UP_CODE [ 2 ] <> 0
        THEN BEGIN
            LEAD_IN := TRUE;
            CURSOR_LEAD := UP_CODE [ 1 ];
            UP := UP_CODE [ 2 ];
            DOWN := DOWN_CODE [ 2 ];
            RIGHT := RIGHT_CODE [ 2 ];
            LEFT := LEFT_CODE [ 2 ];
        END
        ELSE BEGIN
            LEAD_IN := FALSE;
            CURSOR_LEAD := 0;
            UP := UP_CODE [ 1 ];
            DOWN := DOWN_CODE [ 1 ];
            RIGHT := RIGHT_CODE [ 1 ];
            LEFT := LEFT_CODE [ 1 ];
        END
    END
    ($I+)
END;

(*****)

PROCEDURE TRANS_OPEN: ( FILE_NAME : NAME_TYPE;
                      VAR IO_OK : BOOLEAN )

```

```

BEGIN

    ($I-)
    DELETE ( FILE_NAME, ( LENGTH ( FILE_NAME ) - 6 ), 7 );
    REWRITE ( TR_FILE, CONCAT ( '#5:', FILE_NAME, '.CBL' ) );
    IO_OK := IORESULT > 0;
    TR_PTR := LOG_FIRST^.PHYS_PTR^.NEXT_P
    ($I+)

END;

(*****)

PROCEDURE TRANS_CLOSE; ( CLOSE_OK : BOOLEAN )

BEGIN

    IF CLOSE_OK
    THEN CLOSE ( TR_FILE, LOCK )
    ELSE CLOSE ( TR_FILE )

END;

(*****)

PROCEDURE TRANS_STORE; ( VAR TEXT_IN : LINE;
                        VAR IO_OK : BOOLEAN )

BEGIN

    ($I-)
    TR_FILE^ := TEXT_IN;
    PUT ( TR_FILE );
    IO_OK := IORESULT = 0
    ($I+)

END;

(*****)

PROCEDURE TRANS_FETCH; ( VAR TEXT_OUT : LINE;
                        VAR IO_OK : BOOLEAN;
                        VAR TRANS_DONE : BOOLEAN )

    VAR TEXT_LINE : COPY_REC;
        I, O : INTEGER;

BEGIN

    ($I-)
    TRANS_DONE := FALSE;
    IO_OK := TRUE;
    IF TR_PTR <> NIL
    THEN BEGIN
        SEEK ( COPY_P, TR_PTR^.PHYS_ADRS );
        GET ( COPY_P );
        TEXT_LINE := COPY_P^;
        TR_PTR := TR_PTR^.NEXT_P;
        IO_OK := IORESULT = 0
    
```

```

        END
    ELSE
        TRANS_DONE := TRUE
        FOR O := 1 TO 7 DO
            TEXT_OUT [ O ] := BLANK;
        O := 8;
        FOR I := 1 TO 64 DO
            BEGIN
                TEXT_OUT [ O ] := TEXT_LINE [ I ];
                O := SUCC ( O )
            END;
        ($I+)
    END;

END;

(*****)

PROCEDURE METRIC_STORE; ( VAR IO_OK : BOOLEAN;
                        METRICS_IN : METRICS )

BEGIN

    ($I-)
    M_FILE^ := METRICS_IN;
    PUT ( M_FILE );
    METR_PTR^.METRIC_ADRS := M_INDEX;
    M_INDEX := SUCC ( M_INDEX );
    IO_OK := IORESULT = 0;
    ($I+)

END;

(*****)

PROCEDURE METRIC_FETCH; ( VAR M_LINE_OUT : COPY_REC;
                        VAR METRICS_FOUND,
                        IO_OK : BOOLEAN )

BEGIN

    ($I-)
    IO_OK := TRUE;
    IF METR_PTR <> NIL
    THEN BEGIN
        SEEK ( COPY_P, METR_PTR^.PHYS_PTR^.PHYS_ADRS );
        GET ( COPY_P );
        M_LINE_OUT := COPY_P^;
        METR_PTR := METR_PTR^.NEXT_L;
        METRICS_FOUND := TRUE;
        IO_OK := IORESULT = 0
    END
    ELSE
        METRICS_FOUND := FALSE
    ($I+)

END;

(*****)

PROCEDURE METRIC_LOCATE; ( VAR METRICS_OUT : METRICS;

```

```

                                VAR ID_FOUND : BOOLEAN;
                                VAR METRICS_FOUND : BOOLEAN;
                                VAR LINE_TOTAL : INTEGER;
                                ID_NAME : COPY_REC )

BEGIN

  ($I-)
  ID_FOUND := FALSE;
  METRICS_FOUND := FALSE;
  METR_PTR := LOG_FIRST^.NEXT_L;
  WHILE ( NOT ID_FOUND ) AND ( METR_PTR <> NIL ) DO
    BEGIN
      SEEK ( COPY_P, METR_PTR^.PHYS_PTR^.PHYS_ADRS );
      GET ( COPY_P );
      ID_FOUND := ( ID_NAME = COPY_P^ ) AND ( IORESULT = 0 );
      METR_PTR := METR_PTR^.NEXT_L;
    END;
  IF ID_FOUND
  THEN BEGIN
    METRICS_FOUND := METR_PTR^.METRIC_ADRS >= 0;
    IF METRICS_FOUND
    THEN BEGIN
      SEEK ( M_FILE, METR_PTR^.METRIC_ADRS );
      GET ( M_FILE );
      METRICS_OUT := M_FILE^;
    END;
    LINE_TOTAL := METR_PTR^.BLK_LNS;
  END;
  METR_PTR := METR_PTR^.PREV_L;
  ($I+)

END;

(*****)

PROCEDURE READ_OPEN ( VAR FILE_FOUND : BOOLEAN;
                      FILE_NAME : NAME_TYPE;
                      VAR NEW_FILE : BOOLEAN );

  VAR RESULT_C, RESULT_M, I : INTEGER;

BEGIN

  DELETE ( FILE_NAME, ( LENGTH ( FILE_NAME ) - 6 ), 7 );
  BACKUP_NAME := FILE_NAME;
  IF RAM_SYS
  THEN BEGIN
    REWRITE ( COPY_P, '#9:HLSEW.WRK.C$$' );
    RESULT_C := -IORESULT;
    REWRITE ( M_FILE, '#9:HLSEW.WRK.M$$' );
    RESULT_M := IORESULT;
  END
  ELSE
  BEGIN
    REWRITE ( COPY_P, '#5:HLSEW.WRK.C$$' );
    RESULT_C := IORESULT;
    REWRITE ( M_FILE, '#5:HLSEW.WRK.M$$' );
    RESULT_M := IORESULT;
  END;
END;

```

```

FILE_FOUND := TRUE;
NEW_FILE := TRUE;
IF LENGTH ( FILE_NAME ) > 0
  THEN BEGIN
    RESET ( P_FILE, CONCAT ( '#:', FILE_NAME, '.TXT' ) );
    IF ( RESULT_C > 0 ) OR ( RESULT_M > 0 ) OR ( IORESULT > 0 )
      THEN BEGIN
        CLOSE ( P_FILE );
        CLOSE ( M_FILE );
        CLOSE ( COPY_P );
        FILE_FOUND := FALSE
      END
    ELSE
      NEW_FILE := FALSE;
  END
END;

END;

(*****)

PROCEDURE INIT_POINTERS;

BEGIN

  MARK ( LOG_FIRST );
  LAST_L := LOG_FIRST;
  NEW ( L_PTR );
  WITH LAST_L^ DO
    BEGIN
      BLK_LNS := -1;
      METRIC_ADRS := -1;
      NEXT_L := L_PTR;
      PREV_L := NIL;
      PHYS_PTR := NIL
    END;
  WITH L_PTR^ DO
    BEGIN
      BLK_LNS := -1;
      METRIC_ADRS := -1;
      NEXT_L := L_PTR;
      PREV_L := NIL;
      PHYS_PTR := NIL
    END;
  MARK ( PHYS_FIRST );
  LAST_P := PHYS_FIRST;
  NEW ( P_PTR );
  WITH LAST_P^ DO
    BEGIN
      PHYS_ADRS := -1;
      NEXT_P := P_PTR;
      PREV_P := NIL;
      LOG_PTR := NIL
    END;
  WITH P_PTR^ DO
    BEGIN
      PHYS_ADRS := -1;
      NEXT_P := P_PTR;
      PREV_P := NIL;
      LOG_PTR := NIL
    END;
END;

```

```

L_PTR := LAST_L;
P_PTR := LAST_P;

END;

(*****)

PROCEDURE READ_FILE; ( VAR FILE_FOUND : BOOLEAN;
                      FILE_NAME : NAME_TYPE;
                      VAR TOTAL_LINES : INTEGER;
                      VAR NEW_FILE : BOOLEAN )

VAR TEMP_INDEX, BUF_INDEX : INTEGER;
    OK : BOOLEAN;
    TEMP_LINE : COPY_REC;
    BUFF : PACKED ARRAY [ 1..BLOCK_SIZE ] OF CHAR;
    LINE_MODE : LINE_TYPE;

BEGIN

    ($I-)
    P_INDEX := 0;
    M_INDEX := 0;
    LINE_COUNT := 0;
    TOTAL_LINES := 0;
    READ_OPEN ( FILE_FOUND, FILE_NAME, NEW_FILE );
    IF FILE_FOUND
    THEN BEGIN
        INIT_POINTERS;
        OK := BLOCKREAD ( P_FILE, BUFF, 1 ) = 1;
        WHILE OK DO
            BEGIN
                BUF_INDEX := 1;
                WHILE ( BUFF [ BUF_INDEX ] <> CHR ( 0 ) )
                    AND ( BUF_INDEX <= BLOCK_SIZE ) DO
                    BEGIN
                        FOR TEMP_INDEX := 1 TO STORE_LENGTH DO
                            TEMP_LINE [ TEMP_INDEX ] := BLANK;
                        TEMP_INDEX := 1;
                        WHILE ( BUFF [ BUF_INDEX ] <> NL ) AND
                            ( BUFF [ BUF_INDEX ] <> LF ) AND
                            ( TEMP_INDEX <= STORE_LENGTH ) AND
                            ( BUF_INDEX <= BLOCK_SIZE ) DO
                            BEGIN
                                TEMP_LINE [ TEMP_INDEX ] := BUFF [ BUF_INDEX ];
                                LINE_MODE := STAND_LINE;
                                IF ( TEMP_LINE [ TEMP_INDEX ] <> BLANK ) AND
                                    ( TEMP_INDEX <= LABEL_END )
                                THEN LINE_MODE := BLOCK_LINE;
                                TEMP_INDEX := SUCC ( TEMP_INDEX );
                                BUF_INDEX := SUCC ( BUF_INDEX );
                            END; ( while )
                        COPY_P^ := TEMP_LINE;
                        PUT ( COPY_P );
                        NEW ( P_PTR );
                        IF LINE_MODE = BLOCK_LINE
                        THEN BEGIN
                            NEW ( L_PTR );
                            L_PTR^.BLK_LNS := 0;
                            L_PTR^.METRIC_ADRS := -1;

```

```

        L_PTR^.NEXT_L := LAST_L^.NEXT_L;
        L_PTR^.PREV_L := LAST_L;
        L_PTR^.NEXT_L^.PREV_L := L_PTR;
        L_PTR^.PHYS_PTR := P_PTR;
        LAST_L := L_PTR
    END; ( if )
    P_PTR^.LOG_PTR := L_PTR;
    P_PTR^.NEXT_P := LAST_P^.NEXT_P;
    P_PTR^.PREV_P := LAST_P;
    P_PTR^.PHYS_ADRS := P_INDEX;
    P_PTR^.NEXT_P^.PREV_P := P_PTR;
    LAST_P^.NEXT_P := P_PTR;
    LAST_P := P_PTR;
    L_PTR^.BLK_LNS := SUCC ( L_PTR^.BLK_LNS );
    P_INDEX := SUCC ( P_INDEX )
END; ( while )
OK := BLOCKREAD ( P_FILE, BUFF, 1 ) = 1
END; ( ok )
L_PTR := LOG_FIRST;
LAST_L := L_PTR;
P_PTR := PHYS_FIRST;
LAST_P := P_PTR;
CLOSE ( P_FILE )
END ( ok )
($I+)
END;

(*****)
PROCEDURE WRITE_OPEN ( FILE_NAME : NAME_TYPE;
                      VAR IO_OK : BOOLEAN );

BEGIN
    DELETE ( FILE_NAME, ( LENGTH ( FILE_NAME ) - 6 ), 7 );
    IF RAM_SYS
    THEN REWRITE ( P_FILE, CONCAT ( '#9:', FILE_NAME, '.TXT' ) );
    ELSE REWRITE ( P_FILE, CONCAT ( '#5:', FILE_NAME, '.TXT' ) );
    IO_OK := IORESULT = 0
END;

(*****)
PROCEDURE WRITE_CLOSE; ( IO_OK : BOOLEAN )

BEGIN
    IF IO_OK
    THEN CLOSE ( P_FILE, LOCK )
    ELSE CLOSE ( P_FILE );
    CLOSE ( M_FILE, PURGE );
    CLOSE ( COPY_P, PURGE );
END;

(*****)
PROCEDURE WRITE_FILE; ( VAR IO_OK : BOOLEAN;
                      FILE_NAME : NAME_TYPE )

```



```

VAR CHAR_INDEX, BUF_INDEX, CHAR_COUNT : INTEGER;
    BUFF : PACKED ARRAY [ 1..BLOCK_SIZE ] OF CHAR;
    WRITE_OK : BOOLEAN;

BEGIN

    ($I-)
    WRITE_OK := TRUE;
    IO_OK := TRUE;
    BUF_INDEX := 0;
    WRITE_OPEN ( FILE_NAME, IO_OK );
    P_PTR := PHYS_FIRST^.NEXT_P;
    WRITE ( 'WRITING: PHYSICAL FILE' );
    WHILE ( P_PTR^.NEXT_P <> NIL ) AND ( IO_OK ) AND ( WRITE_OK ) DO
        BEGIN
            SEEK ( COPY_P, P_PTR^.PHYS_ADRS );
            GET ( COPY_P );
            CHAR_INDEX := 1;
            WHILE COPY_P^[ CHAR_INDEX ] <> NL DO
                CHAR_INDEX := SUCC ( CHAR_INDEX );
            CHAR_COUNT := CHAR_INDEX;
            WRITE ( PERIOD );
            IF CHAR_COUNT >= ( BLOCK_SIZE - BUF_INDEX )
                THEN BEGIN
                    FOR CHAR_INDEX := ( BUF_INDEX + 1 ) TO BLOCK_SIZE DO
                        BUFF [ CHAR_INDEX ] := CHR ( 0 );
                    WRITE_OK := BLOCKWRITE ( P_FILE, BUFF, 1 ) = 1;
                    IO_OK := IORESULT = 0;
                    FOR BUF_INDEX := 1 TO BLOCK_SIZE DO
                        BUFF [ BUF_INDEX ] := BLANK;
                    BUF_INDEX := 0;
                END; ( if )
            FOR CHAR_INDEX := 1 TO CHAR_COUNT DO
                BEGIN
                    BUF_INDEX := SUCC ( BUF_INDEX );
                    BUFF [ BUF_INDEX ] := COPY_P^[ CHAR_INDEX ]
                END; ( for )
            P_PTR := P_PTR^.NEXT_P;
        END; ( while )
    P_PTR := P_PTR^.NEXT_P;
    IF WRITE_OK
        THEN BEGIN
            WRITE_OK := BLOCKWRITE ( P_FILE, BUFF, 1 ) = 1;
            IO_OK := IORESULT = 0;
        END; ( ok )
    WRITE_CLOSE ( WRITE_OK )
    ($I+)

END;

(*****)

PROCEDURE INSERT_RENUM ( LN_TYPE : LINE_TYPE );

    VAR SAVE_P : ^PHYSICAL;
        COUNT : INTEGER;

BEGIN

```

```

CASE LN_TYPE OF
  STAND_LINE : L_PTR^.BLK_LNS := SUCC ( L_PTR^.BLK_LNS );
  BLOCK_LINE : BEGIN
    IF P_PTR^.NEXT_P^.LOG_PTR = P_PTR^.PREV_P^.LOG_PTR
    THEN BEGIN
      COUNT := 0;
      SAVE_P := P_PTR;
      REPEAT
        SAVE_P^.LOG_PTR := L_PTR;
        SAVE_P := SAVE_P^.NEXT_P;
        COUNT := SUCC ( COUNT )
      UNTIL ( SAVE_P^.LOG_PTR <> P_PTR^.PREV_P^.LOG_PTR )
            OR ( SAVE_P^.LOG_PTR = NIL );
      L_PTR^.BLK_LNS := COUNT;
      IF L_PTR^.PREV_L^.PHYS_PTR <> NIL
      THEN
        L_PTR^.PREV_L^.BLK_LNS :=
          L_PTR^.PREV_L^.BLK_LNS - PRED ( COUNT )
      END
    ELSE
      L_PTR^.BLK_LNS := SUCC ( L_PTR^.BLK_LNS )
    END
  END
END
END;

(*****

PROCEDURE DELETE_RENUM ( LN_TYPE : LINE_TYPE );

  VAR SAVE_P : ^PHYSICAL;
      COUNT : INTEGER;

BEGIN
  CASE LN_TYPE OF
    STAND_LINE : L_PTR^.BLK_LNS := PRED ( L_PTR^.BLK_LNS );
    BLOCK_LINE : BEGIN
      IF P_PTR^.LOG_PTR <> L_PTR
      THEN BEGIN
        COUNT := 0;
        SAVE_P := P_PTR;
        REPEAT
          SAVE_P^.LOG_PTR := L_PTR^.PREV_L;
          SAVE_P := SAVE_P^.NEXT_P;
          COUNT := SUCC ( COUNT );
        UNTIL ( SAVE_P^.NEXT_P = NIL ) OR
              ( SAVE_P^.LOG_PTR = L_PTR );
        L_PTR^.PREV_L^.BLK_LNS :=
          L_PTR^.PREV_L^.BLK_LNS + COUNT;
      END
    ELSE
      L_PTR^.BLK_LNS := PRED ( L_PTR^.BLK_LNS );
    END;
  END;
END;

(*****

```

```

PROCEDURE CHECK_TYPE ( LINE_IN : COPY_REC;
                      VAR LN_TYPE : LINE_TYPE );

CONST LABEL_END = 4;
      LABEL_START = 1;

VAR I : INTEGER;

BEGIN

  LN_TYPE := STAND_LINE;
  I := LABEL_START;
  REPEAT
    IF LINE_IN [ I ] < BLANK
      THEN
        LN_TYPE := BLOCK_LINE;
        I := SUCC ( I )
    UNTIL ( I > LABEL_END ) OR
          ( LN_TYPE = BLOCK_LINE ) OR
          ( LINE_IN [ I ] = NL )

END;

(*****)

PROCEDURE STAND_MOD ( LINE_IN : COPY_REC;
                     MODE_IN : MODE_TYPE;
                     LN_TYPE : LINE_TYPE;
                     DISK_OK : BOOLEAN );

VAR I : INTEGER;
    TYPE_CHANGE : BOOLEAN;
    TEMP_TYPE : LINE_TYPE;

BEGIN
  CASE MODE_IN OF
    INSERT_MODE : BEGIN
      NEW ( P_PTR );
      P_PTR^.NEXT_P := LAST_P^.NEXT_P;
      P_PTR^.PREV_P := LAST_P;
      P_PTR^.NEXT_P^.PREV_P := P_PTR;
      LAST_P^.NEXT_P := P_PTR;
      P_PTR^.PHYS_ADRS := P_INDEX;
      P_PTR^.LOG_PTR := L_PTR;
      SEEK ( COPY_P, P_PTR^.PHYS_ADRS );
      COPY_P^ := LINE_IN;
      PUT ( COPY_P );
      LAST_P := P_PTR;
      IF LN_TYPE = BLOCK_LINE
        THEN L_PTR^.PHYS_PTR := P_PTR;
      -INSERT_RENUM ( LN_TYPE );
      P_INDEX := SUCC ( P_INDEX );
      LINE_COUNT := SUCC ( LINE_COUNT );
      DISK_OK := IORESULT = 0
    END;
    CHANGE_MODE : BEGIN
      SEEK ( COPY_P, P_PTR^.NEXT_P^.PHYS_ADRS );
      COPY_P^ := LINE_IN;
      CHECK_TYPE ( COPY_P^, TEMP_TYPE );
      TYPE_CHANGE := TEMP_TYPE <> LN_TYPE;
      IF TYPE_CHANGE

```

```

        THEN BEGIN
            P_PTR^.PREV_P^.NEXT_P := P_PTR^.NEXT_P;
            P_PTR^.NEXT_P^.PREV_P := P_PTR^.PREV_P;
            P_PTR := P_PTR^.NEXT_P;
            LAST_P := P_PTR;
            DELETE_RENUM ( TEMP_TYPE )
        END;
        SEEK ( COPY_P, P_PTR^.NEXT_P^.PHYS_ADRS );
        COPY_P := LINE_IN;
        PUT ( COPY_P );
        DISK_OK := IORESULT = 0
    END;
DELETE_MODE : BEGIN
    P_PTR^.PREV_P^.NEXT_P := P_PTR^.NEXT_P;
    P_PTR^.NEXT_P^.PREV_P := P_PTR^.PREV_P;
    P_PTR := P_PTR^.NEXT_P;
    LAST_P := P_PTR;
    DELETE_RENUM ( LN_TYPE )
END;

END;

END;

(*****)

PROCEDURE BLOCK_MOD ( LINE_IN : COPY_REC;
                      MODE_IN : MODE_TYPE;
                      LN_TYPE : LINE_TYPE;
                      DISK_OK : BOOLEAN );

VAR I : INTEGER;
    TYPE_CHANGE : BOOLEAN;
    TEMP_TYPE : LINE_TYPE;

BEGIN
    CASE MODE_IN OF
        INSERT_MODE : BEGIN
            TYPE_CHANGE := FALSE;
            IF ( L_PTR = LOG_FIRST ) AND ( LINE_COUNT > 0 )
            THEN BEGIN
                L_PTR := L_PTR^.NEXT_L;
                SEEK ( COPY_P, L_PTR^.PHYS_PTR^.PHYS_ADRS );
                GET ( COPY_P );
                CHECK_TYPE ( COPY_P, TEMP_TYPE );
                TYPE_CHANGE := TEMP_TYPE <> LN_TYPE;
                LAST_P := P_PTR^.PREV_P;
                DISK_OK := IORESULT = 0
            END;
            IF NOT TYPE_CHANGE
            THEN BEGIN
                NEW ( L_PTR );
                L_PTR^.NEXT_L := LAST_L^.NEXT_L;
                L_PTR^.PREV_L := LAST_L;
                L_PTR^.NEXT_L^.PREV_L := L_PTR;
                LAST_L^.NEXT_L := L_PTR;
                L_PTR^.BLK_LNS := 0;
                L_PTR^.METRIC_ADRS := -1;
                LAST_L := L_PTR
            END
        END
    END

```

```

END;
CHANGE_MODE : BEGIN
    SEEK ( COPY_P, L_PTR^.PHYS_PTR^.PHYS_ADRS );
    GET ( COPY_P );
    CHECK_TYPE ( COPY_P^, TEMP_TYPE );
    TYPE_CHANGE := TEMP_TYPE <> LN_TYPE;
    IF TYPE_CHANGE
    THEN BEGIN
        IF ( L_PTR^.PREV_L <> LOG_FIRST )
        THEN BEGIN
            NEW ( L_PTR );
            L_PTR^.NEXT_L := LAST_L^.NEXT_L;
            L_PTR^.PREV_L := LAST_L;
            L_PTR^.NEXT_L^.PREV_L := L_PTR;
            LAST_L^.NEXT_L := L_PTR;
            L_PTR^.PHYS_PTR := P_PTR;
            P_PTR^.LOG_PTR := L_PTR;
            L_PTR^.BLK_LNS := 0;
            L_PTR^.METRIC_ADRS := -1;
            LAST_L^.METRIC_ADRS := -1;
            LAST_L := L_PTR;
            INSERT_RENUM ( TEMP_TYPE )
        END;
        SEEK ( COPY_P, L_PTR^.PHYS_PTR^.PHYS_ADRS );
        COPY_P^ := LINE_IN;
        PUT ( COPY_P );
        DISK_OK := IORESULT = 0
    END
END;
DELETE_MODE : BEGIN
    IF ( L_PTR^.PREV_L = LOG_FIRST ) AND
        ( L_PTR^.BLK_LNS > 1 )
    THEN
        L_PTR^.PHYS_PTR := L_PTR^.PHYS_PTR^.NEXT_P
    ELSE BEGIN
        L_PTR^.PREV_L^.NEXT_L := L_PTR^.NEXT_L;
        L_PTR^.NEXT_L^.PREV_L := L_PTR^.PREV_L;
        L_PTR := L_PTR^.NEXT_L;
        LAST_L := L_PTR
    END
END
END

END;

{*****}
PROCEDURE LINE_MOVE ( INPUT_LINE_NO : INTEGER;
                     VAR LINE_FOUND : BOOLEAN );

BEGIN
    LINE_FOUND := TRUE;
    WHILE ( LINE_COUNT <> INPUT_LINE_NO ) AND ( LINE_FOUND ) DO
        BEGIN
            IF LINE_COUNT < INPUT_LINE_NO
            THEN BEGIN
                IF P_PTR^.NEXT_P^.LOG_PTR = NIL
                THEN
                    LINE_FOUND := FALSE
            END
        END
    END

```

```

        ELSE BEGIN
            P_PTR := P_PTR^.NEXT_P;
            LINE_COUNT := SUCC ( LINE_COUNT )
        END
    END
ELSE
    IF LINE_COUNT > INPUT_LINE_NO
    THEN BEGIN
        IF P_PTR^.PREV_P = NIL
        THEN
            LINE_FOUND := FALSE
        ELSE BEGIN
            P_PTR := P_PTR^.PREV_P;
            LINE_COUNT := PRED ( LINE_COUNT )
        END
    END
END;
IF P_PTR^.LOG_PTR = NIL
THEN L_PTR := LOG_FIRST
ELSE L_PTR := P_PTR^.LOG_PTR;
LAST_P := P_PTR;
LAST_L := L_PTR;

END;

(*****)

PROCEDURE FILE_RELOAD;

VAR IO_OK, NAME_OK : BOOLEAN;

BEGIN

    IF LENGTH ( FILE_NAME ) = 0
    THEN WRITE_FILE ( IO_OK, 'HLSEW.WRK' )
    ELSE WRITE_FILE ( IO_OK, BACKUP_NAME );
    RELEASE ( LOG_FIRST );
    RELEASE ( PHY_FIRST );
    IF IO_OK
    THEN READ_FILE ( NAME_OK, BACKUP_NAME, TOTAL_LINES, NEW_FILE );

END;

(*****)

PROCEDURE STORE_A_LINE; ( LINE_REC : LINE;
                        LINE_NO : INTEGER;
                        MODE_IN : MODE_TYPE )

VAR LN_TYPE : LINE_TYPE;
    FOUND_IT, MEM_OK, DISK_OK : BOOLEAN;
    R, I : INTEGER;
    LINE_OUT : COPY_REC;

BEGIN

    ($I-)
    R := START_LABEL;
    FOR I := 1 TO STORE_LENGTH DO
        BEGIN

```

```

        LINE_IN [ I ] := LINE_REC [ R ];
        R := SUCC ( R );
    END;
    IF ( MODE_IN <> DELETE_MODE )
    THEN LINE_MOVE ( LINE_NO, FOUND_IT );
    CHECK_TYPE ( LINE_IN, LN_TYPE );
    CASE LN_TYPE OF
        BLOCK_LINE : BEGIN
            BLOCK_MOD ( LINE_IN, MODE_IN,
                        BLOCK_LINE, DISK_OK );
            IF ( MODE_IN <> CHANGE_MODE ) AND DISK_OK
            THEN STAND_MOD ( LINE_IN, MODE_IN,
                            BLOCK_LINE, DISK_OK )
            END;
        STAND_LINE : BEGIN
            IF LINE_COUNT <= 0
            THEN BLOCK_MOD ( LINE_IN, MODE_IN,
                            STAND_LINE, DISK_OK );
            IF DISK_OK
            THEN STAND_MOD ( LINE_IN, MODE_IN,
                            STAND_LINE, DISK_OK )
            END
        END;
    END;
    MEM_CHECK ( MEM_OK );
    IF ( NOT MEM_OK ) OR ( NOT DISK_OK )
    THEN FILE_RELOAD;

    ($I+)

END;

(*****)

PROCEDURE FETCH_A_LINE; ( INPUT_LINE_NO : INTEGER;
                          VAR LINE_REC : LINE;
                          VAR LINE_FOUND : BOOLEAN;
                          NEW_FILE : BOOLEAN )

    VAR R, I : INTEGER;
        LINE_OUT : COPY_REC;

BEGIN

    ($I-)
    IF NEW_FILE
    THEN
        LINE_FOUND := FALSE
    ELSE BEGIN
        LINE_MOVE ( INPUT_LINE_NO, LINE_FOUND );
        IF LINE_FOUND
        THEN BEGIN
            SEEK ( COPY_P, P_PTR^.PHYS_ADRS );
            GET ( COPY_P );
            LINE_OUT := COPY_P^
        END
        END;
    FOR R := 1 TO ( START_LABEL - 1 ) DO
        LINE_REC [ R ] := BLANK;
    R := START_LABEL;
    FOR I := 1 TO STORE_LABEL DO

```

```

    BEGIN
        LINE_REC [ R ] := LINE_OUT [ I ];
        R := SUCC ( R );
    END;

    { $I+ }

END;

(*****)

PROCEDURE DELETE_A_LINE; ( INPUT_LINE_NO : INTEGER;
                           VAR LINE_FOUND : BOOLEAN )

BEGIN

    { $I- }
    IF LINE_COUNT > 0
    THEN BEGIN
        LINE_MOVE ( INPUT_LINE_NO, LINE_FOUND );
        IF LINE_FOUND
        THEN BEGIN
            SEEK ( COPY_P, P_PTR^.PHYS_ADRS );
            GET ( COPY_P );
            STORE_A_LINE ( COPY_P^, INPUT_LINE_NO, DELETE_MODE )
        END
    END
    ELSE
        LINE_FOUND := FALSE
    { $I+ }

END;

(*****)

END. ( unit store )

```


Appendix V

Editing System Source Code

Software Notice

The source code listing of the Line and Screen Editing System which are found on the following pages, represents the combined effort of this author and a previously published work. Although the original design is not the total work of this author, a number of changes and additions have been made to circumvent system incompatibilities, and to enhance system features and performance.

```

(*****
*
*   EEEEEEEEEEE DDDDDDDDD IIIIIIII TTTTTTTTTT 0000000 RRRRRRRR
*   EEEEEEEEEEE DDDDDDDDD IIIIIIII TTTTTTTTTT 000000000 RRRRRRRR
*   EEE          DDD   DDD   III          TTT          000   000 RRR   RRR
*   EEE          DDD   DDD   III          TTT          000   000 RRR   RRR
*   EEEEEEEEEEE DDD   DDD   III          TTT          000   000 RRR   RRR
*   EEEEEEEEEEE DDD   DDD   III          TTT          000   000 RRRRRRRR
*   EEE          DDD   DDD   III          TTT          000   000 RRR   RRR
*   EEEEEEEEEEE DDDDDDDDD IIIIIIII TTT          000000000 RRR   RRR
*   EEEEEEEEEEE DDDDDDDDD IIIIIIII TTT          0000000   RRR   RRR
*
*****

```

UNIT EDITOR;

```

(*****
INTERFACE
(*****

```

```

USES (<$U #9:DECS.CODE>
DECS,

```

```

<$U #9:STORE.CODE>
STORE;

```

```

(*****
INTERFACE PROCEDURES
*****

```

```

PROCEDURE ANALYZER ( VAR OK : BOOLEAN );
PROCEDURE TRANSLATOR ( VAR OK : BOOLEAN );
PROCEDURE BUILD_A_LINE ( VAR DONE, ESCAPE : 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 ERROR ( ERROR_KIND : ERROR_TYPE );

```

```

(*****
IMPLEMENTATION
(*****

```

(*****)

PROCEDURE HELP_IT;

BEGIN

```
WRITELN ('The following is a summary of the Editor Commands');
WRITELN; WRITELN; WRITELN;
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
('line number or a range of line numbers, e.g. DL arg, {arg}');
```

WRITELN; WRITELN; WRITELN

```
('"CH" changes an existing string with a new string at the current');
WRITELN
('line number, e.g. CH/old text/new text/ {arg}. The third argument');
```

WRITELN

```
('has three options. It may be null (default is 1), or it may be an');
WRITELN
('integer specifying the number of occurrences to change, of 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 the');
```

WRITELN

```
('current tab character, e.g. TC {arg}');
```

WRITELN; WRITELN; WRITELN

```
('"ST" allows you to set the tabs. This command followed by a <CR>');
```

WRITELN

```
('will display the current tab settings. Otherwise this command');
```

WRITELN

```
('should be followed by the tab setting arguments tou desired, separated');
```

WRITELN

```
('by a comma or a space, e.g. ST {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;

```

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

PROCEDURE ANALYZER; BEGIN END;
PROCEDURE TRANSLATOR; BEGIN END;

(*****)

PROCEDURE ERROR; ( ERROR_KIND : ERROR_TYPE )

BEGIN

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

END;

(*****
      -
*****)

PROCEDURE ND_SCREEN_PUT; ( INPUT_STRING : STRING )

BEGIN

  GOTOXY (0,0);
  CLEAR_TO_EOLN;
  GOTOXY (0,0);
  WRITELN (INPUT_STRING);

END;

```

```

(*****)

PROCEDURE CLEAR_TO_EOLN;

  VAR I : INTEGER;

BEGIN
  FOR I := 1 TO 6 DO
    WRITE ( CHR(CLRL[I]));
  END;

(*****)

PROCEDURE GET_CHARACTER; (LEGAL_CHAR : SET_OF_VALID;
                          STRING_LENGTH : INTEGER;
                          VAR START : INTEGER;
                          VAR FOUND : BOOLEAN)

  VAR CH : CHAR;

BEGIN
  ESCAPE := FALSE;
  ALL_DONE := FALSE;
  REPEAT READ ( KEYBOARD,CH) UNTIL CH IN LEGAL_CHAR;
  IF CH = CHR (CURSOR_LEAD)
    THEN READ ( KEYBOARD, CH );
  IF CH = CHR(LEFT)
    THEN BEGIN
      WRITE (CHR(CURSOR_LEAD), CHR(LEFT));
      WRITE(CHR(SPACE));
      WRITE (CHR(CURSOR_LEAD), CHR(LEFT));
    END;
  IF CH = CHR(ETX) THEN ALL_DONE := TRUE;
  IF CH = CHR(ESC)
    THEN BEGIN
      ESCAPE := TRUE;
      ALL_DONE := TRUE;
    END;
  CH_OUT := CH;
END;

(*****)

PROCEDURE BUILD_A_LINE; ( VAR DONE, ESCAPE : BOOLEAN )

  VAR C : CHAR;
  I : INTEGER;
  ALL_DONE : BOOLEAN;
  VALID_SET_CHAR : SET OF CHAR;

BEGIN
  VALID_SET_OF_CHAR := [ CHR(CURSOR_LEAD), CHR(SPACE)..CHR(DEL), CHR(CR),
                        CHR(LEFT), CHR(ESC), CHR(ETX), TAB_CHARACTER ];
  FOR I := 1 TO 7 DO BEGIN
    C := CHR(SPACE);

```

```

    WRITE (OUTPUT,C);
    TEMP_LINE [I] := C;
END;
LINE_INDEX := 8;
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 (LEFT)
    THEN BEGIN
        LINE_INDEX := PRED (LINE_INDEX);
        IF LINE_INDEX <= 0 THEN LINE_INDEX := 1;
    END
    ELSE IF C = TAB_CHARACTER
    THEN REPEAT
        C:= CHR (SPACE);
        WRITE (OUTPUT, 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;
RESET (KEYBOARD);
WRITELN;
FOR I := LINE_INDEX TO LINE_LENGTH DO
    TEMP_LINE [I] := NL;
DONE := ALL_DONE;

END;

(*****

PROCEDURE FIND_TOKEN; ( VAR NEXT_TOKEN : TOKEN )

    VAR C : CHAR;
        I : INTEGER;

BEGIN

    WITH NEXT_TOKEN DO
        BEGIN REPEAT
            C := INPUT_LINE [LINE_INDEX];
            LINE_INDEX := SUCC (LINE_INDEX);
            UNTIL C <> CHR (SPACE);
            VALUE := PRED (LINE_INDEX);
            IF C = CHR (13)
            THEN TOKEN_KIND := NILTOK
            ELSE IF (C < '0') OR (C > '9')
            THEN TOKEN_KIND := OTHERTOK
            ELSE BEGIN
                TOKEN_KIND := LINENOTOK;
                VALUE := 0;
                REPEAT
                    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;

(*****)

PROCEDURE FIND_STRING; ( STRING : LINE;
                        STRING_LENGTH : INTEGER;
                        VAR START : INTEGER;
                        VAR FOUND : BOOLEAN )

    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;
            END;
            IF NOT FOUND
            THEN START := SUCC (START);
            DONE := FOUND OR (START + I > TEMP_LENGTH);
        UNTIL DONE;

    END;

(*****)

PROCEDURE VERIFY_IT;

BEGIN

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

END;

```

(*****)

```
PROCEDURE FETCH_CURRENT_LINE; ( INPUT_LINE_NO : INTEGER;  
                               VAR LINE_FOUND : BOOLEAN )
```

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

```
BEGIN
```

```
    INPUT_LINE_NO := SUCC (INPUT_LINE_NO);  
    FETCH_A_LINE (INPUT_LINE_NO, CURRENT_LINE, FOUND, NEW_FILE);  
    LINE_FOUND := FOUND;  
    IF FOUND  
    THEN BEGIN  
        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  
        TEMP_LINE := CURRENT_LINE;  
        WRITE ('LINE ', INPUT_LINE_NO, ' ');  
        WRITELN (' NOT FOUND');  
        (ERROR (NOT_FOUND))  
    END;  
END;
```

```
END;
```

(*****)

```
PROCEDURE PRINT_TEMP_LINE;
```

```
VAR I : INTEGER;
```

```
BEGIN
```

```
    IF TEMP_LENGTH = 1 THEN WRITE ('. ');  
    FOR I := 1 TO TEMP_LENGTH DO  
        WRITE (TEMP_LINE [I]);
```

```
END;
```

(*****)

```
PROCEDURE STORE_CURRENT_LINE; ( MODE : MODE_TYPE )
```

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

```
BEGIN
```

```
    FOR I := 1 TO LINE_LENGTH DO  
        CURRENT_LINE [I] := TEMP_LINE [I];  
    MODE_OUT := MODE;
```



```

STORE_A_LINE (CURRENT_LINE, LINE_NUMBER, MODE_OUT);
IF MODE_OUT <> CHANGE_MODE
  THEN BEGIN
    LINE_NUMBER := SUCC (LINE_NUMBER);
    TOTAL_LINES := SUCC (TOTAL_LINES)
  END;
FILE_CHANGES := TRUE;
NEW_FILE := FALSE;

END;

(*****

PROCEDURE CHANGE_TEXT;

  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
      DELIM := INPUT_LINE [VALUE];
      OLD_LENGTH := 0;
      INDEX := SUCC (VALUE);
      WHILE (INPUT_LINE [INDEX] <> DELIM) AND
        (INPUT_LINE [INDEX] <> NL)
      DO BEGIN
        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
          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
          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 :=
                (TOTAL_LINES - 1)
            ELSE CHANGE_COUNT := 0;
    END;
END;
SINGLE_CHANGE := CHANGE_COUNT = 1;
FETCH_CURRENT_LINE (LINE_NUMBER, LINE_FOUND);
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
            LINE_NUMBER := SUCC (LINE_NUMBER);
            FETCH_CURRENT_LINE (LINE_NUMBER, LINE_FOUND);
            STRING_START := 1;
        END
    ELSE BEGIN
        CHANGE_COUNT := PRED (CHANGE_COUNT);
        LINE_CHANGED := TRUE;
        IF TEMP_LENGTH - OLD_LENGTH + NEW_LENGTH > 132
        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;
    END;
    IF LINE_CHANGED
    THEN BEGIN
        STORE_CURRENT_LINE (CHANGE_MODE);
        IF VERIFY_CHANGES
        THEN PRINT_TEMP_LINE;
        LINE_CHANGED := FALSE;
    END;
END;
END;
END;
END;

```

(*****)

```

PROCEDURE MAKE_TAB_SETTING;

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

BEGIN

  FIND_TOKEN (NEXT);
  CASE NEXT.TOKEN_KIND OF
    NILTOK : BEGIN
      IF EDITING_FROM_SCREEN
      THEN BEGIN
        GOTOXY (0,0);
        CLEAR_TO_EOLN;
        GOTOXY (0,0);
      END;
      WRITE ('TAB SETTINGS : ');
      FOR I := 1 TO 80 DO
        IF I IN TABS THEN WRITE (I, ' ');
      WRITELN;
      IF EDITING_FROM_SCREEN
      THEN BEGIN
        WRITE (' Press space bar to continue..... ');
        READ (INPUT, SPACE_BAR);
        WRITELN;
        GOTOXY (0,1);
        CLEAR_TO_EOLN;
      END;
    END;
    OTHERTOK : BEGIN
      C := INPUT_LINE [4];
      IF C = 'C'
      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]
        ELSE ERROR (OTHER_ERROR);
      FIND_TOKEN (NEXT);
      UNTIL NEXT.TOKEN_KIND <> LINENOTOK;
    END;
  END;

END;

(*****

PROCEDURE SET_TAB_CHAR;

  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;
END;

END;

(*****)

PROCEDURE INSERT_TEXT;

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

BEGIN
  DONE := FALSE;
  FIND_TOKEN (NEXT);
  IF NEXT.TOKEN_KIND <> NILTOK
  THEN ERROR (COMMAND_ERROR)
  ELSE BEGIN
    MODE_OUT := INSERT_MODE;
    WHILE NOT DONE DO
      BEGIN
        BUILD_A_LINE (DONE, ESCAPE);
        IF (NOT ESCAPE) AND (NOT DONE)
        THEN STORE_CURRENT_LINE (MODE_OUT);
      END;
    END;
    NEW_FILE := FALSE;
  END;
END;

(*****)

PROCEDURE DELETE_LINES;

  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;
        OTHERTOK : OK := FALSE;
        LINENOTOK : BEGIN
          LAST_LINE := VALUE;
          FIND_TOKEN (NEXT);
          IF NEXT.TOKEN_KIND <> NILTOK
            THEN OK := FALSE;
          END;
        END
      END
    END
  END;
END;
IF NOT OK
THEN ERROR (COMMAND_ERROR)
ELSE BEGIN
  FOR I := FIRST_LINE TO LAST_LINE DO
    BEGIN
      DELETE_A_LINE ((FIRST_LINE + 1), FOUND);
      IF NOT FOUND
      THEN BEGIN
        WRITE (I, ' ');
        ERROR (NOT_FOUND);
      END
      ELSE TOTAL_LINES := PRED (TOTAL_LINES);
    END;
  {LINE_NUMBER := LAST_LINE;}
  FILE_CHANGED := TRUE;
END;
END;

(*****)

PROCEDURE LIST_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
        FIRST_LINE := LINE_NUMBER;
        LAST_LINE := FIRST_LINE + 20;
      END;
    END;
  END;

```

```

OTHERTOK : OK := FALSE;
LINENOTOK : BEGIN
    FIRST_LINE := VALUE - 1;
    FIND_TOKEN (NEXT);
    WITH NEXT DO
        CASE TOKEN_KIND OF
            NILTOK : BEGIN
                LAST_LINE := FIRST_LINE;
                LINE_NUMBER := LAST_LINE;
            END;
            OTHERTOK : OK := FALSE;
            LINENOTOK : BEGIN
                LAST_LINE := VALUE - 1;
                LINE_NUMBER := LAST_LINE;
                FIND_TOKEN (NEXT);
                IF NEXT.TOKEN_KIND <> NILTOK
                THEN OK := FALSE
                ELSE;
            END;
        END;
    END;
END;
IF NOT OK
THEN ERROR (COMMAND_ERROR)
ELSE FOR I := FIRST_LINE TO LAST_LINE DO
    BEGIN
        FETCH_CURRENT_LINE (I, LINE_FOUND);
        IF LINE_FOUND
        THEN PRINT_TEMP_LINE;
    END;
END;

END;

(*****

PROCEDURE QUIT;

    VAR I : INTEGER;

BEGIN

    FOR I := 1 TO 6 DO
        WRITE (CHR (CLRS [I]));
    GOTOXY (15, 12);
    WRITELN (' END OF HLSEW SESSION ');

END;

(*****

PROCEDURE READ_COMMAND (VAR COMMAND : COMMAND_TYPE);

    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(SOH)..CHR(HT), CHR(CR)..CHR(DEL) ];
LINE_INDEX := 1;
REPEAT
  GET_CHARACTER (VALID_SET_OF_CHAR, C, ALL_DONE, ESCAPE)
UNTIL C <> CHR(SPACE);
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(LEFT)
      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;
  END;
  READLN (KEYBOARD);
  WRITELN;
  FOR I := LINE_INDEX TO LINELENGTH DO
    INPUT_LINE [I] := CHR (CR);
  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_IT;
    '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;
END;

(*****)

PROCEDURE EXECUTE_COMMAND (COMMAND_IN : COMMAND_TYPE);

  VAR LINE_FOUND : BOOLEAN;

BEGIN
  CASE COMMAND_IN OF
    CHANGE : CHANGE_TEXT;
    DELETE_IT : DELETE_LINES;
    ENEDIT : 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;

END;

(*****

PROCEDURE INIT_LINE;

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(CR);
    COMMAND := BADCOMMAND;
    EDITING_FROM_SCREEN := FALSE;

END;

(*****

PROCEDURE LINE_EDIT; ( COMMAND_IN : COMMAND_TYPE )

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;

(*****

END. (* UNIT EDITOR *)

```



```

(*****
*
*   SSSSSSSS  CCCCCCCC  RRRRRRRR  EEEEEEEEE  EEEEEEEEE  NNN  NNN  *
*   SSSSSSSSS CCCCCCCCC RRRRRRRRR EEEEEEEEE EEEEEEEEE NNNN  NNN  *
*   SSS      CCCC      RRRR  RRR  EEE      EEE      NNNNN  NNN  *
*   SSS      CCCC      RRRR  RRRR  EEE      EEE      NNNNNN  NNN  *
*   SSSSSSSS  CCCC      RRRRRRRR  EEEEEEEEE EEEEEEEEE NNNNNNNNNN  *
*   SSSSSSSSS CCCC      RRRRRRRR  EEEEEEEEE EEEEEEEEE NNN  NNNNNN  *
*   SSS CCCC      RRRRRRRR  EEE      EEE      NNN  NNNNN  *
*   SSS CCCC      RRR  RRRR  EEE      EEE      NNN  NNNNN  *
*   SSSSSSSSS CCCCCCCCC RRR  RRRR  EEEEEEEEE EEEEEEEEE NNN  NNNN  *
*   SSSSSSSS  CCCCCCCC  RRR  RRRR  EEEEEEEEE EEEEEEEEE NNN  NNN  *
*
*****)

```

```
PROGRAM HLSEW;
```

```
USES (*$U #9:DECS.CODE*)
DECS,
```

```
(*$U #9:STORE.CODE*)
STORE,
```

```
(*$U #9:EDITOR.CODE*)
EDITOR;
```

```
(*$R SCREEN_EDIT *)
```

```
PROCEDURE COMMAND_INTERPRETER; FORWARD;
```

```
(*****)
```

```
PROCEDURE CLEAR_SCREEN;
```

```
VAR I : INTEGER;
```

```
BEGIN
```

```
FOR I := 1 TO 6 DO
  WRITE (CHR (CLRS [I]));
```

```
END;
```

```
(*****)
```

```
SEGMENT PROCEDURE SCREEN_EDIT;
```

```
PROCEDURE MOVE_RIGHT; FORWARD;
PROCEDURE MOVE_LEFT; FORWARD;
PROCEDURE EDITOR_MENU; FORWARD;
PROCEDURE EDITOR_COMMAND_INTERPRETER; FORWARD;
```

```
(*****)
```

```
PROCEDURE BELL;
```

```
BEGIN
  WRITE (CHR (BEL));
```

```

END;

(*****)

PROCEDURE ERASE_REST_OF_PAGE;

  VAR I : INTEGER;

BEGIN
  FOR I := 1 TO 6 DO
    WRITE (CHR (CLRP [I]));
  END;

(*****)

PROCEDURE CONVERT_INTEGER (INT_IN : INTEGER);

  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] := CHR (SPACE);
  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] <> CHR (SPACE)
    THEN BEGIN
      INPUT_LINE [LINE_INDEX] := TEMP [I];
      LINE_INDEX := SUCC (LINE_INDEX);
    END;
  END;

END;

(*****)

PROCEDURE PAGE;

BEGIN
  GOTOXY (0,1);
  ERASE_REST_OF_PAGE;
  GOTOXY (0,2);
  LINE_INDEX := 3;
  INPUT_LINE [LINE_INDEX] := NL;
  LINE_EDIT (LIST);
  GOTOXY (0,2);
  ROW := 2;
  COLUMN := 0;
  ROW_MARK := 2;

END;

```

(*****)

PROCEDURE PAGE_BACK;

VAR TEMP_NUMBER : INTEGER;

BEGIN

TEMP_NUMBER := LINE_NUMBER;
IF (LINE_NUMBER - 20) < 0
THEN LINE_NUMBER := 0
ELSE LINE_NUMBER := LINE_NUMBER - 20;
GOTOXY (0,1);
ERASE_REST_OF_PAGE;
GOTOXY (0,2);
LINE_INDEX := 3;
INPUT_LINE [LINE_INDEX] := NL;
LINE_EDIT (LIST);
LINE_NUMBER := TEMP_NUMBER;
ROW := LINE_NUMBER + 2;
ROW_MARK := ROW;
GOTOXY (COLUMN, ROW);

END;

(*****)

PROCEDURE READ_SCREEN_COMMAND;

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

BEGIN

VALID_SET_OF_CHAR := [CHR(SOH)..CHR(HT), CHR(CR)..CHR(DEL)];
INPUT_LINE [3] := CHR(SPACE);
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 (LEFT)
THEN BEGIN
LINE_INDEX := PRED (LINE_INDEX);
IF LINE_INDEX <= 2
THEN LINE_INDEX := 2;
END
ELSE BEGIN
WRITE (C);
INPUT_LINE [LINE_INDEX] := C
END;
END;
READLN (KEYBOARD);
WRITELN;

```

IF ESCAPE
  THEN BEGIN
    EDITOR_MENU;
    EXIT (EDITOR_COMMAND_INTEPRETER);
  END;
FOR I := LINE_INDEX TO LINE_LENGTH DO
  INPUT_LINE [I] := CHR(CR);
LINE_INDEX := 3;
END;

(*****)

PROCEDURE SCREEN_INSERT;

  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 (ESC) ];
  GET_CHARACTER (VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
  CASE CH OF
    'C' : BEGIN
      IF (TOTAL_LINES <= 0) OR (LINE_NUMBER >= TOTAL_LINES)
        THEN EXIT (SCREEN_INSERT);
      VALID_SET_OF_CHAR := [ CHR(SOH)..CHR(HT), CHR(CR)..CHR(DEL) ];
      ND_SCREEN_PUT ('CHAR INSERT: <cntrl C> accepts <esc> aborts');
      LINE_EDIT (FETCH_IT);
      GOTOXY (COLUMN, ROW);
      CLEAR_TO_EOLN;
      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(LEFT)
              THEN BEGIN
                MOVE_LEFT;
                FOR I := (COLUMN + 1) TO (LINE_LENGTH - 1) DO
                  TEMP_LINE [I] := TEMP_LINE [I + 1];
                END
              ELSE BEGIN
                FOR I := (LINE_LENGTH - 1) DOWNTO (COLUMN + 1) DO
                  TEMP_LINE [I + 1] := TEMP_LINE [I];
                TEMP_LINE [COLUMN + 1] := CH;
                WRITE (CH);
                MOVE_RIGHT;
              END;
            GET_CHARACTER (VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
          END;
          IF (ALL_DONE) AND (NOT ESCAPE) THEN LINE_EDIT (STORE_IT);
          LINE_EDIT (FETCH_IT);
          GOTOXY (0, ROW);
          PRINT_TEMP_LINE;
        END;
      END;
    'B' : BEGIN
      ND_SCREEN_PUT ('BLOCK INSERT: <cntrl C> and <CR> accepts');

```

```

        GOTOXY (0,ROW);
        ERASE_REST_OF_PAGE;
        LINE_INDEX := 3;
        INPUT_LINE [LINE_INDEX] := NL;
        LINE_EDIT (INSERT);
        GOTOXY (0,2);
        ERASE_REST_OF_PAGE;
        IF (LINE_NUMBER + 10) <= TOTAL_LINES
            THEN LINE_NUMBER := LINE_NUMBER - 10
            ELSE LINE_NUMBER := TOTAL_LINES - 20;
        IF LINE_NUMBER < 0
            THEN LINE_NUMBER := 0;
        PAGE;
    END;
END;
IF ESCAPE THEN EXIT (SCREEN_INSERT);

END;

(*****

PROCEDURE SCREEN_DELETE;

    VAR CH : CHAR;
        ALL_DONE, ESCAPE, FOUND : BOOLEAN;
        VALID_SET_OF_CHAR : SET_OF_VALID;
        I, LAST_LINE, FIRST_LINE, TEMP_COLUMN, TEMP_TOTAL : INTEGER;

BEGIN

    ND_SCREEN_PUT
        ('DELETE: <down arrow> line, <right arrow> char, <cntrl C> accepts');
    WRITE ('', <esc> aborts');
    VALID_SET_OF_CHAR := [ CHR(CURSOR_LEAD), CHR(DOWN), CHR(RIGHT),
                           CHR(ESC), CHR(ETX) ];
    GOTOXY (COLUMN, ROW);
    GET_CHARACTER (VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
    IF CH = CHR(RIGHT)
        THEN BEGIN
            IF (TOTAL_LINES <= 0) OR (LINE_NUMBER >= TOTAL_LINES)
                THEN EXIT (SCREEN_DELETE);
            VALID_SET_OF_CHAR := [ CHR(CURSOR_LEAD), CHR(RIGHT), CHR(ESC), CHR(ETX) ];
            LINE_EDIT (FETCH_IT);
            GOTOXY (COLUMN, ROW);
            TEMP_COLUMN := COLUMN;
            WHILE (NOT ALL_DONE) AND (NOT ESCAPE) DO
                BEGIN
                    IF TEMP_COLUMN >= 79 THEN BELL
                        ELSE BEGIN
                            FOR I := (COLUMN + 1) TO (TEMP_LENGTH - 1) DO
                                TEMP_LINE [I] := TEMP_LINE [I + 1];
                            WRITE (CHR(SPACE));
                            TEMP_LENGTH := PRED (TEMP_LENGTH);
                            TEMP_COLUMN := SUCC (TEMP_COLUMN);
                        END;
                    GET_CHARACTER (VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
                END;
            IF (ALL_DONE) AND (NOT ESCAPE) THEN LINE_EDIT (STORE_IT);
            LINE_EDIT (FETCH_IT);
            CLEAR_TO_EOLN;
        END
    END

```

```

        GOTOXY (0,ROW);
        PRINT_TEMP_LINE;
    END;
    IF CH = CHR(DOWN)
    THEN BEGIN
        IF (TOTAL_LINES <= 0) OR (LINE_NUMBER >= TOTAL_LINES)
        THEN EXIT (SCREEN_DELETE);
        TEMP_TOTAL := TOTAL_LINES;
        FIRST_LINE := LINE_NUMBER;
        VALID_SET_CHAR := [ CHR(CURSOR_LEAD), CHR(DOWN), CHR(ETX), CHR(ESC) ];
        WHILE (NOT ALL_DONE) AND (NOT ESCAPE) DO
            BEGIN
                GOTOXY (0,ROW);
                CLEAR_TO_EOLN;
                IF ROW < (TEMP_TOTAL + 2)
                THEN BEGIN
                    ROW := SUCC (ROW);
                    GOTOXY (0,ROW);
                    LINE_NUMBER := SUCC (LINE_NUMBER);
                    TEMP_TOTAL := PRED (TEMP_TOTAL);
                END;
                GET_CHARACTER (VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
            END;
            LINE_NUMBER := PRED (LINE_NUMBER);
            IF NOT ESCAPE
            THEN BEGIN
                LAST_LINE := LINE_NUMBER;
                FOR I := 1 TO LINE_LENGTH DO
                    INPUT_LINE [I] := CHR(SPACE);
                LINE_INDEX := 3;
                CONVERT_INTEGER (FIRST_LINE);
                LINE_INDEX := SUCC (LINE_INDEX);
                CONVERT_INTEGER (LAST_LINE);
                FOR I := LINE_INDEX TO LINE_LENGTH DO
                    INPUT_LINE [I] := NL;
                LINE_INDEX := 3;
                LINE_EDIT (DELETE_IT);
            END;
            GOTOXY (0,2);
            LINE_NUMBER := PRED (LINE_NUMBER);
            IF (LINE_NUMBER + 10) <= TOTAL_LINES
            THEN LINE_NUMBER := LINE_NUMBER - 10
            ELSE LINE_NUMBER := TOTAL_LINES - 20;
            IF LINE_NUMBER < 0
            THEN LINE_NUMBER := 0;
            PAGE;
        END;
        IF ESCAPE THEN EXIT (SCREEN_DELETE);
    END;

    (*****)

    PROCEDURE EXCHANGE;

        VAR CH : CHAR;
            ALL_DONE, ESCAPE, FOUND : BOOLEAN;
            VALID_SET_OF_CHAR : SET_OF_VALID;

    BEGIN

```

```

ND_SCREEN_PUT ('EXCHANGE: <esc> to abort <ctrl C> to escape');
GOTOXY (COLUMN, ROW);
VALID_SET_OF_CHAR := [ CHR(ETX), CHR(ESC), CHR(LEFT),
                        CHR(CURSOR_LEAD), CHR(SPACE)..CHR(DEL) ];
GET_CHARACTER (VALID_SET_OF_CHAR, CH, ALL_DONE, ESCAPE);
IF ROW > TOTAL_LINES THEN
  EXIT (EXCHANGE);
WHILE (NOT ALL_DONE) AND (NOT ESCAPE) DO
  BEGIN
    LINE_EDIT (FETCH_IT);
    IF COLUMN = 79 THEN BELL
    ELSE
      IF CH = CHR(LEFT)
      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;
IF (ALL_DONE) AND (NOT ESCAPE)
  THEN LINE_EDIT (STORE_IT);
IF ESCAPE
  THEN BEGIN
    LINE_EDIT (FETCH_IT);
    GOTOXY (0, ROW);
    PRINT_TEMP_LINE;
  END;
END;

(*****

PROCEDURE REPLACE;

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 (' Pres space bar to continue.....');
  READ (INPUT, SPACE_BAR);
  GOTOXY (0,1);
  CLEAR_TO_EOLN;
  IF (LINE_NUMBER + 10) <= TOTAL_LINES
    THEN LINE_NUMBER := LINE_NUMBER - 10
    ELSE LINE_NUMBER := TOTAL_LINES - 20;
  IF LINE_NUMBER < 0
    THEN LINE_NUMBER := 0;
  PAGE;

END;

(*****

```

```

PROCEDURE SET_THE_TABS;

BEGIN
    ND_SCREEN_PUT ('SET TABS: <cr> display current tabs ');
    READ_SCREEN_COMMAND;
    LINE_EDIT (SETTABS);

END;

(*****)

PROCEDURE ESTABLISH_TAB_CHAR;

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

END;

(*****)

PROCEDURE OTHER;

    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_CHARS;
        'R' : EXIT (OTHER);
    END;

END;

(*****)

PROCEDURE MOVE_UP;

BEGIN
    IF LINE_NUMBER < 1 THEN EXIT (MOVE_UP);
    IF ROW <= ROW_MARK
    THEN BEGIN
        LINE_NUMBER -= PRED (LINE_NUMBER);
        ROW_MARK := PRED (ROW_MARK);
    END;
    ROW := PRED (ROW);
    GOTOXY (COLUMN, ROW);
    IF ROW < 2
    THEN PAGE_BACK;

END;

(*****)

```



```

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

(*****)

PROCEDURE EDITOR_COMMAND_INTERPRETER;

    VAR INPUT : CHAR;

BEGIN
    REPEAT
        READ (KEYBOARD, INPUT)
    UNTIL INPUT IN [ 'I', 'D', 'R', 'Q', 'X', 'O', CHR(CURSOR_LEAD) ];

    IF (INPUT = CHR (CURSOR_LEAD)) AND (LEAD_IN)
    THEN READ (KEYBOARD, INPUT);
    IF INPUT = CHR(DOWN) THEN MOVE_DOWN;
    IF INPUT = CHR(UP) THEN MOVE_UP;
    IF INPUT = CHR(LEFT) THEN MOVE_LEFT;
    IF INPUT = CHR(RIGHT) 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;
    IF INPUT IN [ 'I', 'X', 'D', 'R', 'O' ]
    THEN BEGIN
        EDITOR_MENU;
        GOTOXY (COLUMN, ROW);
    END;
END;

(*****)

PROCEDURE SET_UP_SCREEN_EDITOR;

BEGIN
    TABS := [ 8, 12, 16, 20, 32, 36, 40, 56, 73 ];
    TAB_CHARACTER := ' ';
    LINE_NUMBER := 0;
    NL := CHR (CR);
    COMMAND := BADCOMMAND;
    EDITING_FROM_SCREEN := TRUE;
    CLEAR_SCREEN;
    EDITOR_MENU;

```

PROCEDURE MOVE_DOWN;

BEGIN

```
IF LINE_NUMBER > TOTAL_LINES
  THEN BEGIN
    LINE_NUMBER := SUCC (LINE_NUMBER);
    ROW := SUCC (ROW);
    ROW_MARK := SUCC (ROW_MARK);
  END
ELSE
  ROW := ROW_MARK;
GOTOXY (COLUMN, ROW);
IF ROW > 22
  THEN BEGIN
    IF (LINE_NUMBER + 10) >= TOTAL_LINES
      THEN LINE_NUMBER := LINE_NUMBER - 10
      ELSE LINE_NUMBER := TOTAL_LINE - 20;
    PAGE;
  END;
END;
```

END;

(*****)

PROCEDURE MOVE_RIGHT;

BEGIN

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

END;

(*****)

PROCEDURE MOVE_LEFT;

BEGIN

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

END;

(*****)

PROCEDURE EDITOR_MENU;

```

LINE_INDEX := 1;
PAGE;

END;

(*****
      MAIN BODY OF SEGMENT SCREEN_EDIT
*****)

BEGIN
    SET_UP_SCREEN_EDITOR;
    EDITOR_COMMAND_INTERPRETER;
    REPEAT
        EDITOR_COMMAND_INTERPRETER;
    UNTIL FINISHED;

END;

(*****

PROCEDURE OUTER_COMMAND_MENU;

BEGIN
    ND_SCREEN_PUT
        ('COMMAND: S)creen edit  T)ranslate  A)nalyze  Q)uit  L)ine edit');

END;

(*****

PROCEDURE GET_FILE (VAR FOUND : BOOLEAN);

BEGIN
    FILE_CHANGED := FALSE;
    ND_SCREEN_PUT
        ('FILE NAME: <cr> new file <file name> existing file: ');
    READLN (FILE_NAME);
    READ_FILE (FOUND, FILE_NAME, TOTAL_LINES, NEW_FILE);
    IF NOT FOUND
    THEN BEGIN
        ERROR (NOT_FOUND);
        EXIT (COMMAND_INTERPRETER);
    END;

END;

(*****

PROCEDURE GET_NAME (VAR OK : BOOLEAN);

BEGIN
    OK := FALSE;
    ND_SCREEN_PUT
        ('FILE NAME: <cr> deletes edited file <file name> names new file: ');
    READLN (FILENAME);
    IF LENGTH (FILE_NAME) > 0

```

```

        THEN OK := TRUE;
        CLEAR_SCREEN;

    END;

    (*****)

    PROCEDURE FILE_REFORM ( VAR OK : BOOLEAN );

        VAR IN_NAME, OUT_NAME : NAME_TYPE;
            SELECT_TYPE : REFORM_TYPE;
            RESPONSE : CHAR;

    BEGIN

        ND_SCREEN_PUT ('File Reform Utility');
        GOTOXY(0,2);
        WRITE ('Name of File to Reform: ');
        READLN (IN_NAME);
        WRITE ('Name of Reformed File: ');
        READLN (OUT_NAME);
        WRITELN;
        WRITE ('Reform Type: <C> Carriage Return, <L> Linefeed/Carriage Return: ');
        REPEAT
            READ (KEYBOARD,RESPONSE)
        UNTIL RESPONSE IN [ 'L', 'C' ];
        IF RESPONSE = 'L'
            THEN SELECT_TYPE := LF_CR
            ELSE SELECT_TYPE := CR_ONLY;
        REFORM (FOUND, OK, IN_NAME, OUT_NAME, SELECT_TYPE);
        IF NOT FOUND
            THEN BEGIN
                ERROR (NOT_FOUND);
                OK := TRUE;
                EXIT (COMMAND_INTERPRETER)
            END
        END;

    (*****)

    PROCEDURE COMMAND_INTERPRETER;

        VAR CH : CH-1;
            OK : BOOLEAN;

    BEGIN

        REPEAT
            READ (KEYBOARD, CH)
        UNTIL CH IN [ 'A', 'T', 'S', 'Q', 'L', 'R' ];
        CASE CH OF
            'S' : BEGIN
                GET_FILE (OK);
                SCREEN_EDIT;
                IF NEW_FILE
                    THEN OK := FALSE
                    ELSE IF NOT FILE_CHANGED
                        THEN OK := FALSE
                        ELSE IF LENGTH (FILE_NAME) <= 0

```

```

        THEN GET_NAME (OK);
        ELSE IF LENGTH (FILE_NAME) > 0
            THEN OK := TRUE;
    .IF OK
        THEN BEGIN
            WRITE_FILE (OK, FILE_NAME);
            CLEAR_SCREEN;
            IF NOT OK
                THEN ERROR (WRITING);
            WRITE_CLOSE (OK);
        END
        ELSE WRITE_CLOSE (OK);
    END;
'T' : BEGIN
    GET_FILE (OK);
    TRANSLATOR (OK);
    IF NOT OK
        THEN ERROR (TRANS_ERROR);
    END;
'A' : BEGIN
    GET_FILE (OK);
    ANALYZER (OK);
    IF NOT OK
        THEN ERROR (UPDATING);
    END;
'L' : BEGIN
    GET_FILE (OK);
    CLEAR_SCREEN;
    LINE_EDIT (EDIT_IT);
    IF NEW_FILE
        THEN OK := FALSE
        ELSE IF NOT FILE_CHANGED
            THEN OK := FALSE
            ELSE IF LENGTH (FILE_NAME) <= 0
                THEN GET_NAME (OK)
                ELSE IF LENGTH (FILE_NAME) > 0
                    THEN OK := TRUE;
    IF OK
        THEN BEGIN
            WRITE_FILE (OK, FILE_NAME);
            CLEAR_SCREEN;
            IF NOT OK
                THEN ERROR (WRITING);
            WRITE_CLOSE (OK);
        END
        ELSE WRITE_CLOSE (OK);
    END;
'R' : BEGIN
    FILE_REFORM (OK);
    IF NOT OK
        THEN ERROR (REFORM_ERROR)
    END;
'Q' : BEGIN
    CLEAR_SCREEN;
    GOTOXY (25,12);
    WRITELN ('END HLSEW SESSION');
    EXIT (PROGRAM);
END;
END;

```

```

END;

(*****
PROCEDURE INITIALIZE;

    VAR OK : BOOLEAN;

BEGIN

    READ_TERM (LEFT, DOWN, UP, RIGHT, CLRL, CLRS,
               CLRP, CURSOR_LEAD, LEAD_IN, OK);
    IF NOT OK
    THEN BEGIN
        WRITE ('Error: Unable to Open TERMCODE File');
        EXIT ( PROGRAM );
    END;
    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(CR);
    FINISHED := FALSE;
    VERIFY_CHANGES := TRUE;

END;

(*****
                                MAIN PROGRAM
*****

BEGIN

    INITIALIZE;
    COMMAND_INTERPRETER;
    REPEAT
        OUTER_COMMAND_MENU;
        COMMAND_INTERPRETER;
    UNTIL FINISHED;

END.

```

Appendix VI

Installation Source Code

```

*****
*
*
*          .      TERMINAL & SYSTEM INSTALLATION
*
*
*          DESIGNED BY
*
*
*          Russell J. Holt
*
*
*****

```

PROGRAM INSTALL;

TYPE KEYCODE = PACKED ARRAY [1..6] OF INTEGER;

VAR T_FILE : FILE OF KEYCODE;
LEFT, DOWN, UP, RIGHT, CLRL, CLRS, CLRP, RAM_SYS : KEYCODE;
CURSOR_IN : STRING;
I : INTEGER;
OK, LEAD_IN : BOOLEAN;
SYS_OPT, KEY_OPT : CHAR;

PROCEDURE IO_CHECK (RESPONSE : INTEGER);

BEGIN

IF RESPONSE > 0
THEN
BEGIN
GOTOXY (0,0);
WRITE (CHR (7), 'Illegal Response');
EXIT (TERMCODE);
END;
END;

END;

BEGIN

(\$I-)
RESET (T_FILE, '*9:TERMCODE.DATA');
IF IORESULT > 0 (ioresult #1)
THEN BEGIN
REWRITE (T_FILE, '*9:TERMCODE.DATA');
IF IORESULT > 0 (ioresult #2)
THEN BEGIN
WRITE ('Unable to open TERMCODE file');
EXIT (TERMCODE)
END (then ioresult #2)
ELSE FOR I := 1 TO 6 DO (new term file)
BEGIN
LEFT [I] := 0;
DOWN [I] := 0;
UP [I] := 0;


```

        RIGHT [ I ] := 0;
        CLRL [ I ] := 0;
        CLRS [ I ] := 0;
        CLRP [ I ] := 0;
    END ( else & ioresult #2 )
END ( then ioresult #1 )
ELSE BEGIN
    LEFT := T_FILE^;
    GET ( T_FILE ); DOWN := T_FILE^;
    GET ( T_FILE ); UP := T_FILE^;
    GET ( T_FILE ); RIGHT := T_FILE^;
    GET ( T_FILE ); CLRL := T_FILE^;
    GET ( T_FILE ); CLRS := T_FILE^;
    GET ( T_FILE ); CLRP := T_FILE^;
    GET ( T_FILE ); RAM_SYS := T_FILE^;
    CLOSE ( T_FILE )
END; ( else ioresult #1 )

GOTOXY ( 0,0 );
FOR I := 1 TO 23 DO
    BEGIN
        WRITE ( ' ' );
        WRITELN ( ' ' );
    END;
OK := FALSE;
GOTOXY ( 0,0 );
WRITELN ( ' Present Key Configuration' );
WRITELN;
LEAD_IN := FALSE;

WHILE NOT OK DO
    BEGIN
        GOTOXY ( 0,3 );
        WRITE ( 'A> Cursor Left -> ' );
        FOR I := 1 TO 6 DO
            IF LEFT [ I ] <> 0
                THEN WRITE ( LEFT [ I ], ' ' )
                ELSE WRITE ( ' ' );
            WRITELN;

        WRITE ( 'B> Cursor Right -> ' );
        FOR I := 1 TO 6 DO
            IF RIGHT [ I ] <> 0
                THEN WRITE ( RIGHT [ I ], ' ' )
                ELSE WRITE ( ' ' );
            WRITELN;

        WRITE ( 'C> Cursor Up -> ' );
        FOR I := 1 TO 6 DO
            IF UP [ I ] <> 0
                THEN WRITE ( UP [ I ], ' ' )
                ELSE WRITE ( ' ' );
            WRITELN;

        WRITE ( 'D> Cursor Down -> ' );
        FOR I := 1 TO 6 DO
            IF DOWN [ I ] <> 0
                THEN WRITE ( DOWN [ I ], ' ' )
                ELSE WRITE ( ' ' );
            WRITELN;
    END;

```

```

WRITE ( 'E> Clear Line -> ' );
FOR I := 1 TO 6 DO
  IF CLRL [ I ] <> 0
    THEN WRITE ( CLRL [ I ], ' ' )
    ELSE WRITE ( ' ' );
WRITELN;

WRITE ( 'F> Clear screen -> ' );
FOR I := 1 TO 6 DO
  IF CLRS [ I ] <> 0
    THEN WRITE ( CLRS [ I ], ' ' )
    ELSE WRITE ( ' ' );
WRITELN;

WRITE ( 'G> Clear Remainder of Screen -> ' );
FOR I := 1 TO 6 DO
  IF CLRP [ I ] <> 0
    THEN WRITE ( CLRP [ I ], ' ' )
    ELSE WRITE ( ' ' );
WRITELN;

IF RAM_SYS [ 1 ] = 1
  THEN WRITELN ( 'H> Ram Disk System -> YES ' )
  ELSE WRITELN ( 'H> Ram Disk System -> NO ' );

GOTOXY ( 0,12 );
KEY_OPT := ' ';
WRITE ( ' Enter Selection To Change, <RETURN> To Exit: ' );
READ ( KEYBOARD, KEY_OPT );
IF ( KEY_OPT < 'A' ) OR ( KEY_OPT > 'z' )
  THEN OK := TRUE
  ELSE BEGIN
    GOTOXY ( 0,12 );
    WRITE ( ' ');
    WRITE ( ' ');
  END;
GOTOXY ( 0,12 );

CASE KEY_OPT OF
  'A','a' : BEGIN
    FOR I := 1 TO 6 DO LEFT [ I ] := 0;
    WRITELN ( 'Press the Cursor Left key' );
    WRITELN ( ' Followed by a <return>' );
    READ ( KEYBOARD, CURSOR_IN );
    IO_CHECK ( IORESULT );
    FOR I := 1 TO LENGTH ( CURSOR_IN ) DO
      LEFT [ I ] := ORD ( CURSOR_IN [ I ] );
    IF LENGTH ( CURSOR_IN ) > 1 THEN LEAD_IN := TRUE;
    READLN ( KEYBOARD );
    WRITELN
  END;
  'B','b' : BEGIN
    FOR I := 1 TO 6 DO RIGHT [ I ] := 0;
    WRITELN ( 'Press the Cursor Right key' );
    WRITELN ( ' Followed by a <return>' );
    READ ( KEYBOARD, CURSOR_IN );
    IO_CHECK ( IORESULT );
    FOR I := 1 TO LENGTH ( CURSOR_IN ) DO

```

```

        RIGHT [ I ] := ORD ( CURSOR_IN [ I ] );
    IF LENGTH ( CURSOR_IN ) > 1 THEN LEAD_IN := TRUE;
    READLN ( KEYBOARD );
    WRITELN
END;
'C', 'c' : BEGIN
    FOR I := 1 TO 6 DO UP [ I ] := 0;
    WRITELN ( 'Press the Cursor Up key' );
    WRITELN ( ' Followed by a <return>' );
    READ ( KEYBOARD, CURSOR_IN );
    IO_CHECK ( IORESULT );
    FOR I := 1 TO LENGTH ( CURSOR_IN ) DO
        UP [ I ] := ORD ( CURSOR_IN [ I ] );
    IF LENGTH ( CURSOR_IN ) > 1 THEN LEAD_IN := TRUE;
    READLN ( KEYBOARD );
    WRITELN
END;
'D', 'd' : BEGIN
    FOR I := 1 TO 6 DO DOWN [ I ] := 0;
    WRITELN ( 'Press the Cursor Down key' );
    WRITELN ( ' Followed by a <return>' );
    READ ( KEYBOARD, CURSOR_IN );
    IO_CHECK ( IORESULT );
    FOR I := 1 TO LENGTH ( CURSOR_IN ) DO
        DOWN [ I ] := ORD ( CURSOR_IN [ I ] );
    IF LENGTH ( CURSOR_IN ) > 1 THEN LEAD_IN := TRUE;
    READLN ( KEYBOARD );
    WRITELN
END;
'E', 'e' : BEGIN
    FOR I := 1 TO 6 DO CLRL [ I ] := 0;
    WRITELN ( 'Enter Ascii Codes to Clear a Line ' );
    WRITELN ( 'End sequence with a "0" ' );
    WRITELN;
    WRITE ( 'Ascii Code #1: ' );
    READLN ( CLRL [ 1 ] );
    IO_CHECK ( IORESULT );
    I := 1;
    WHILE ( CLRL [ I ] <> 0 ) AND ( I < 6 ) DO
        BEGIN
            I := SUCC ( I );
            WRITE ( 'Ascii Code #', I, ': ' );
            READLN ( CLRL [ I ] );
            IO_CHECK ( IORESULT );
        END
    END;
END;
'F', 'f' : BEGIN
    FOR I := 1 TO 6 DO CLRS [ I ] := 0;
    WRITELN ( 'Enter Ascii Codes to Clear the Screen ' );
    WRITELN ( 'End sequence with a "0" ' );
    WRITELN;
    WRITE ( 'Ascii Code #1: ' );
    READLN ( CLRS [ 1 ] );
    IO_CHECK ( IORESULT );
    I := 1;
    WHILE ( CLRS [ I ] <> 0 ) AND ( I < 6 ) DO
        BEGIN
            I := SUCC ( I );
            WRITE ( 'Ascii Code #', I, ': ' );
            READLN ( CLRS [ I ] );
        END
    END;
END;

```

```

        IO_CHECK ( IORESULT );
    END
END;
'G', 'g' : BEGIN
    FOR I := 1 TO 6 DO CLR [ I ] := 0;
    WRITELN ( 'Enter Ascii Codes to Clear the Remainder of Screen ' );
    WRITELN ( 'End sequence with a "0" ' );
    WRITELN;
    WRITE ( 'Ascii Code #1: ' );
    READLN ( CLR [ 1 ] );
    IO_CHECK ( IORESULT );
    I := 1;
    WHILE ( CLR [ I ] <> 0 ) AND ( I < 6 ) DO
        BEGIN
            I := SUCC ( I );
            WRITE ( 'Ascii Code #', I, ': ' );
            READLN ( CLR [ I ] );
            IO_CHECK ( IORESULT );
        END;
    END;
END;
'H', 'h' : BEGIN
    SYS_OPT := ' ';
    WRITE ( 'Do you wish to use a RAM-DISK system (Y/N)? ' );
    READ ( KEYBOARD, SYS_OPT );
    IF ( SYS_OPT = 'Y' ) OR ( SYS_OPT = 'y' )
        THEN RAM_SYS [ 1 ] := 1
        ELSE RAM_SYS [ 1 ] := 0;
    END;
END;
GOTOXY ( 0, 12 );
FOR I := 12 TO 23 DO
    WRITELN ( '
END;

WRITE ( 'Are These Ascii Codes Correct? Y/N ' );
KEY_OPT := ' ';
READ ( KEYBOARD, KEY_OPT );

IF ( KEY_OPT = 'Y' ) OR ( KEY_OPT = 'y' )
    THEN BEGIN
        IF ( LEAD_IN ) AND
            ( ( LEFT [ 1 ] <> RIGHT [ 1 ] ) OR
              ( LEFT [ 1 ] <> UP [ 1 ] ) OR
              ( LEFT [ 1 ] <> DOWN [ 1 ] ) OR
              ( RIGHT [ 1 ] <> UP [ 1 ] ) OR
              ( RIGHT [ 1 ] <> DOWN [ 1 ] ) OR
              ( UP [ 1 ] <> DOWN [ 1 ] ) )
            THEN BEGIN
                WRITE ( 'The supplied Cursor codes are not supported by the ' );
                WRITE ( 'HLSEW system, Please refer to your user manual' );
            END
        ELSE BEGIN
            REWRITE ( T_FILE, '#9:TERMCODE.DATA' );
            IF IORESULT > 0 THEN EXIT ( TERMCODE );
            T_FILE^ := LEFT;      PUT ( T_FILE );
            T_FILE^ := DOWN;      PUT ( T_FILE );
            T_FILE^ := UP;        PUT ( T_FILE );
            T_FILE^ := RIGHT;     PUT ( T_FILE );
            T_FILE^ := CLRL;      PUT ( T_FILE );
        END
    END

```

```

        T_FILE := CLRS;      PUT ( T_FILE );
        T_FILE := CLRP;      PUT ( T_FILE );
        T_FILE := RAM_SYS;    PUT ( T_FILE );
        CLOSE ( T_FILE, LOCK )
    END

    ($I+)

END;

GOTOXY ( 0,0 );
FOR I := 1 TO 24 DO
    BEGIN
        WRITE ( ' ');
        WRITELN ( ' ');
    END;
END.

```

THE DISK STORAGE SYSTEM OF THE
HIGH LEVEL SOFTWARE ENGINEERING WORKSTATION
(HLSEW)

by

Russell J. Holt

B. S., Washburn University of Topeka, 1980

An Abstract of a Master's Report

submitted in partial fulfillment of the

requirements for the degree

Master of Science

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Ks

1985

Abstract

The author has developed the Data Storage System of an interactive workstation referred to as the High Level Software Engineering Workstation (HLSEW). This interactive workstation which functions as an "intelligent terminal", is designed to aid programmers who use a pseudo-English programming language called a Program Design Language (PDL). An intelligent terminal can do some local processing without communicating with a host computer, they offer users flexibility while freeing the host for other tasks. The HLSEW functions as an intelligent terminal which allows: 1) the creating and editing of a PDL file, 2) the calculation of program metrics, and 3) the translating of a PDL file into a compilable source code file. The eventual purpose of the HLSEW project is to develop an intelligent workstation that helps to identify problems as the user enters the PDL code.

The HLSEW project was originally designed on the PDQ-3. The architecture of the PDQ-3 consists of a LSI-11 CPU, 128K RAM memory, dual 8 inch floppy disk drives, and runs the University of California at San Diego (UCSD) p-system. Further development of the project resulted in the system being transferred to an IBM or IBM-compatible micro computer.