207

A MODULARLY EXPANSIBLE MINIMAL MULTI-SCREEN EDITOR

by

Samuel Mize

B. S., Kansas State University, 1981

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

Approved by :

David A. Gustafson

Major Professor

CONTENTS

CONTENTS -- continued

LIST OF FIGURES:

I would like to acknowledge the help and encouragement of Dr. Dave Gustafson, my major professor. He has guided me, prodded me into doing work I really knew I ought to do but probably wouldn't have, made suggestions, motivated me when I was down and heard me out when I was up. It has been my pleasure to work with him, and my distinct honor to call him my friend.

I also thank Dr. Virgil Wallentine and Dr. William Hankley for acting as my committee. Their interest, advice, and interesting questions were greatly appreciated. In fact, their (sometimes embarrassing) questions were more appreciated in retrospect than may have been apparent at the time.

CHAPTER 1
MOTIVATION AND NEEDS FOR A NEW EDITOR AT KSU

1.1 INTRODUCTION

A screen editor (also called a display editor) displays a section of an edited file on an interactive display. Editing can be done on any currently displayed part of the file. The cursor is moved to locate the current working position in the file, and changes to the file cause immediate changes to the displayed section. A multiple window screen editor has a screen which can be divided into more than one editing area, showing either different parts of the same file or different files. Each editing area is referred to as a "window."

This paper describes the design and implementation at Kansas State University of a multi-window screen editor with a simple, modular command language. This editor allows multi-window access to a single file, or to several files. Instead of a myriad of specialized commands, it has a fairly minimal command set and facilities to expand the commands.

This editor is useful both as a simple editor for beginners and as a tool for editor experimentation, since the code is available for alteration and recombination. Since it is designed top-down and implemented bottom-up, basic editing operations are defined by procedures and functions. Recombining them into a new editor, if desired, would simply require writing a new driver routine. The C programming language provides the structures needed for the code to be both well structured and fairly efficient.

The term "function" is frequently used in this paper, both in the context "editing function" and to distinguish a procedure with a returned value. An "editing function" is not necessarily a nondestructive subroutine with a return value; the meaning intended is "something you would do while editing."

1.2 GENERAL DISCUSSION OF EDITING, FACILITIES, AND POSSIBILITIES

According to Adams <2>, "Visualization is an important thinking mode which is especially useful in solving problems where shapes, forms, or patterns are used." Certainly programming in a structured language uses patterns; many types of documents also follow patterns. An editor which uses a display capability which helps the user visualize these patterns should be helpful. It can also serve as a form of graphic imagery, which is (according to <2>) able to "aid in one's own process of thinking or to aid in communication with others." Shneiderman <21> discusses the syntactic/semantic model of thought. This may be a model for understanding part of the superiority of a display editor : the programmer immediately remembers about seven "chunks" of information about the program; the display of the editor serves as an extended immediate memory, since one need only look at it. Also, displaying larger areas may permit a better grasp of the underlying semantic objects of the program when they spread over more than one or a few lines.

Shneiderman <22> goes into more detail concerning the relation of the syntactic/semantic model and screen editing. The immediate feedback allows a more correct internal model of the

semantic object being built in the file, and allows the user to be sure that he is correct about the actions of a given command. Syntactic confusion from moving between editors may be lessened by overlapping their syntaxes; if the user can do such syntactic changes on the editor himself, he may be able to significantly improve his performance by tailoring his command names to his own preferences and understandings.

Screen (or display) editing is put forward as being better than line editing in a number of sources, at least as early as 1979 <11>. Shneiderman <22> recommends them for better performance and learning time, citing as advantages the display of context, cursor as visible editing position, intuitive movement in the file, immediate display of results of user actions, and ease of reversing commands (changes, being visible, can be easily changed back). Jong <12>, MacDonald <14>, Finseth <10> and Fraser <11> agree that display editing is superior to line editing, for much the same reasons as well as empirical tests of speed and accuracy. Bates <5> discusses the advantages of a display-oriented format in a data entry system. Badre <4> points out that retention is improved when items presented in a visual format are related; presumably, the context of a program section has sufficient relationships that using a screen editor may help retention of the program or document's details.

A number of features are also cited as being of interest by various authors. Among the items included in the current editor, or which could be built up on it as it stands, are (from <10>, a list of functions compiled from comments by many users at MIT) :

editing a copy instead of the actual file; writing out to a new file name; writing the file out to disk (with the same or a different name) without ending the current edit session; editing large files without the user being aware of the memory and disk space management; editing in multiple windows; mnemonic commands; commands appropriate to the item being edited (i.e., commands for movement by procedures in programs, and by paragraphs in documents -- possible with macros); inserting as well as overwriting commands (an overwrite command is a very simple macro); modifiability. From <12> , in addition : a status window to show current activity status; procedural prompts in commands (i.e. "search for what pattern?"); assignment of keys to commands for discernible reasons, for instance assigning the movement keys mnemonically (UDLR for Up, Down, Left, Right), by physical analogy (WZAS for up, down, left, right, because of their keyboard positions), or by symbolic analogy (V for down, < for left, etc.).

Some of the desirable functions in <10> would require further work to support on this editor, especially : use of control characters instead of normal characters for commands (would require changing a few characters in the command interpreter routine); recovery commands from deletes; and saving the current editing state (although you can write out the current version of the file to a different name). Some of the functions listed in <12> are also not currently supported, specifically : display of special characters; word wrapping (moving entire words that cross the end of line down to next line); reformatting below changes (actually. formatting is not supported much at all); use of

function keys; retention of deletions, and the ability to reverse them.

Several guiding principles of editor design are stated in <12>. The need for user memorization should be minimized. Operations should be efficient, and engineered for errors. The system should respond to any user input. The system should be simple and responsive with immediate, unambiguous feedback to the user. The user should initiate all actions, and be able to quit at any time. The user should be able to customize the editor to suit his own preferences, or different types of text (document versus program). Interface design, for data entry in a screen oriented fashion, is discussed in <5>, and the design of the Star computer's graphic-icon and display editing interface is explained in <23>. The use of metaphor in teaching and learning about a new system is discussed in <7> (i.e., a terminal with a screen editor is like a blackboard, or like a typewriter, etc. to illustrate basic concepts).

In <7>, Carroll and Thomas also discuss ways that command language design affects ease of learning. Commands that form natural pairs should use command words that are natural pairs (for instance, Up and Down, not Up and Forward (with D as delete!). Commands should be formed in similar ways in all areas of the interface (hierarchical consistency), to minimize structural confusion.

Formation of keywords to use in an editor (or other command system) has a growing body of research. Eastman <9> notes that the most successful keyword choices seem to be words formed in

ways that new words in English are formed; (i.e., by phrases becoming words, as have "goto" and "motorcar," or by abbreviations such as "TV," "CRT," "proc" and "var"). The research of Benbasat and Wand indicates that people find truncation of commands the most natural form of abbreviation. As noted above, Jong in <12> promotes discernible reasons for command word and key choices, and Finseth <10> agrees that command choices should be mnemonic. Landauer et al <13> surveyed the used vocabulary of typists and found that the terms used in most editors are not from this vocabulary. They then compared the typists' performance with editors that use command words from that vocabulary and editors that use arbitrary words for activities. The results did not show an advantage for keywords from the typists' vocabularies, possibly because the nuances of meaning were different, possibly because the study was very limited in scope. Further such research may be interesting.

Aside from the question of what sort of fact the editor should present, there are also many ideas about different basic functions. Denning <8> writes about "smart editors" which have knowledge about program or document structure. Such editors would include the Cornell Program Synthesizer <25>, and systems described in <15>, <17>, and <26>. Work toward generalized syntax analysis, to allow such editors to be generated more easily, is described in <18> and <19>. The current editor and its basic functions could be used as a tool in the development of such a structure editor. One promising course would be to use a file parallel to the edited text, and altered in tandem, to maintain

the structure information. Even if the text file is abolished altogether and the program is kept in parse tree form, as in <15> and <25>, the basic display functions may be useful.

The current set of editing functions would also be most useful in a system designed to document programming as it is done, such as are described in <1>, <17>, <26> and <27>. The systems in <1> and <17> basically use a linear refinement of natural language plans to code; use of parallel windows as the description is changed and developed might allow greater ease of use, especially where the context of an activity could significantly change the best approach to it. All of these systems basically take a description, put it into a formal structure, and use the formal rules of that structure to refine the description. All could be implemented as one or more windows with the current editing functions. Possibly several versions of the refinement could be displayed in tandem, to allow comparison of varying solutions to the same problem.

The current functions, however, may be too slow for a production system. In particular, the scrolling system is slow. However, they are also useful for developing prototypes of a final system. The ability to rapidly prototype a new system is very valuable. Smith <23> points out that there is almost always a version of any significant new system thrown away. The availability of convenient components allows the observation of prototype behavior without a massive programming investment in the prototype.

1.3 NEED FOR A NEW EDITOR AT KSU

There is a clear need for a reconfigurable and easily alterable editor. The ability both to create new editing commands and to mask off old ones would be useful in a number of areas, as would the ability to add entirely new capabilities without having to redesign the overall editor or reprogram the basic editing functions.

One such area of usefulness is a minimal editor with which to teach the basic concepts of editing a file. Chris Rutkowski has shown <20> that a "seven function word processor" can be as useful for this as a four function calculator is for teaching the basics of using a pocket calculator.

Another possible use is comparative studies of different configurations. For example, what command words are easiest to understand and remember? Do people work better with long mnemonic commands or single-key commands? Such questions are amenable to direct experiment. but such comparison experiments would require a large number of slightly different editors. The obvious course is to use one basic editor, and re-configure it to act in different ways. It would also be possible to experiment with truly novel editing formats and ideas without having to re-invent the entire editing system. Indeed, the motivating factor for this work was a desire to observe the possible usefulness of a command added to the editor VI, and learning that it was virtually impossible for us to unsnarl that system enough to do significant work on it.

Another use would occur in the development of new capabilities for editing. They could be easily added to an editor

designed to be easily changed, and evaluated in context with other editing commands. New and specialized instruction sets could be easily generated. These could be either for common dissemination or for private use; they could easily be needed only for a single job, and thus far too unimportant to justify the programmer cost of adding them to an ordinary, established editor.

Creation of such an experimental editor would require provision of both an extensive macro-command system and access to the source code. The macro system would provide the capability for individual users to tailor the editor to their personal requirements. The accessible code would allow experimenters to alter both the actual editing capabilities available and the ways that the user can access these capabilities. It would be necessary for the code to be entirely modular, with all input to and output from each command and subroutine clearly marked. Also, the editor would need to have been designed for such experimentation.

There is no such editor at KSU. There are several line and screen editors, but none have the code accessibility and design versatility needed to be a good experimental editor. For these reasons, it was decided to implement such an experimental editor at Kansas State University.

1.4 REQUIREMENTS DEFINITION FOR THE EDITOR

An experimental editor should be simple, easy to learn and use. Although simple, it need not lack significant power; sufficient commands should be included to make it a useful tool.

It should include useful and significant capabilities, such as multiple windowing, multi-file access, and editing of arbitrary size files; and, it should be easy to add further capabilities to it, as and when they are needed, imagined and developed. It should be very easy to alter and extend on a short-term basis by users, so they can adapt its behavior to their individual tastes. It should also be modifiable on a long-term basis, by changing the code, so that an apparently completely different editor can be provided without significant reprogramming effort to provide the basic editing functions. Such modifiable attributes should include the way that editing functions are invoked by the user. the way that the user can extend his personal command set if such a function is made available, and additions to, deletions from, or alterations to the set of editing functions available.

To be easy to learn and use, an editor needs to have a small set of easily learned commands and an easy way to build new commands. A vast welter of slightly varying commands can confuse even an experienced computer user. A well-chosen set of commands can provide all the basic functions needed. With provision for building new commands from this basic set of "building blocks," all the power needed in an editor is made available; yet, the user is not overwhelmed with a problem in selection and memorization before he can start solving the problem for which he needed the editor.

The editor should have significant capabilities, and be easy to add more capabilities to. It won't be possible to test the effects of new or interesting capabilities if they cannot be put

into the editor. The basic functions of editing -- adding and deleting text, moving within a file, saving the edited file -- must be present, of course. More advanced functions, such as editing two or more files, showing several windows at once, or extracting text for addition at other places, should either exist or be easy to add. This is another argument for a design by functional decomposition, and for modularization of data into formally passed parameters, so that no analysis of variable inheritance or non-obvious data interaction is needed.

To build new commands, there should be provision for users to make small changes and create commands when they run the program, to accommodate their individual tastes and needs. It should also be easy to access the code and make radical changes in the way the editor appears to the user. This implies both a macro-command definition system, and a top-down design taking into consideration the usefulness of each design level's components -- a functional decomposition.

Overall, an editor designed not for immediate efficiency, but for flexibility as an experimental tool, should have an unconfusing but complete set of basic functions, an easy way to expand those functions on both a temporary and a permanent basis, an easy way to radically change the way those functions are invoked, and a design that makes it easy to add to the editor. It should include advanced capabilities, since experiments cannot be designed around such capabilities if they are not available; and, it should be easy to add further capabilities as the need occurs.

CHAPTER 2
DESIGN OF THE EDITOR

2.1 DESIGN DECISIONS

To design the experimental editor, we had to choose the minimal set of commands to include. We decided to include text addition and line deletion, even though these can be constructed from character insert and delete,, because they are extremely common actions. We decided not to include a text extract and reinsert command, for reasons of time. They would certainly be worthwhile additions, and would be difficult to add using the current set of commands. A full list of basic commands included in the editor is given in appendix 3.

The macro system design was very severely constrained by available time. A more extensive system would be a definite bonus. The system as implemented only includes sequencing of commands, repetition, and some primitive logic and comparison functions. The ideal would be almost a language of its own; some possible steps in that direction are presented in appendix 2.

The editor is designed to operate within editing windows. For this purpose, it uses the "curses" package developed by Arnold <3> to define areas of the screen and work in them on a higher conceptual level -- that is, it allows actions such as "print to window" and "clear window" instead of forcing the program to keep character-by-character track of what is done.

One decision that must be made is whether to truncate lines that are longer than the window's width, or to wrap them around to the next line. While wrapping around allows the entire line to be

viewed, it can also make indentation intended to show a program's structure virtually meaningless. It was decided to provide both, and allow the decision to be made by the user (or program) at the time the affected functions are called.

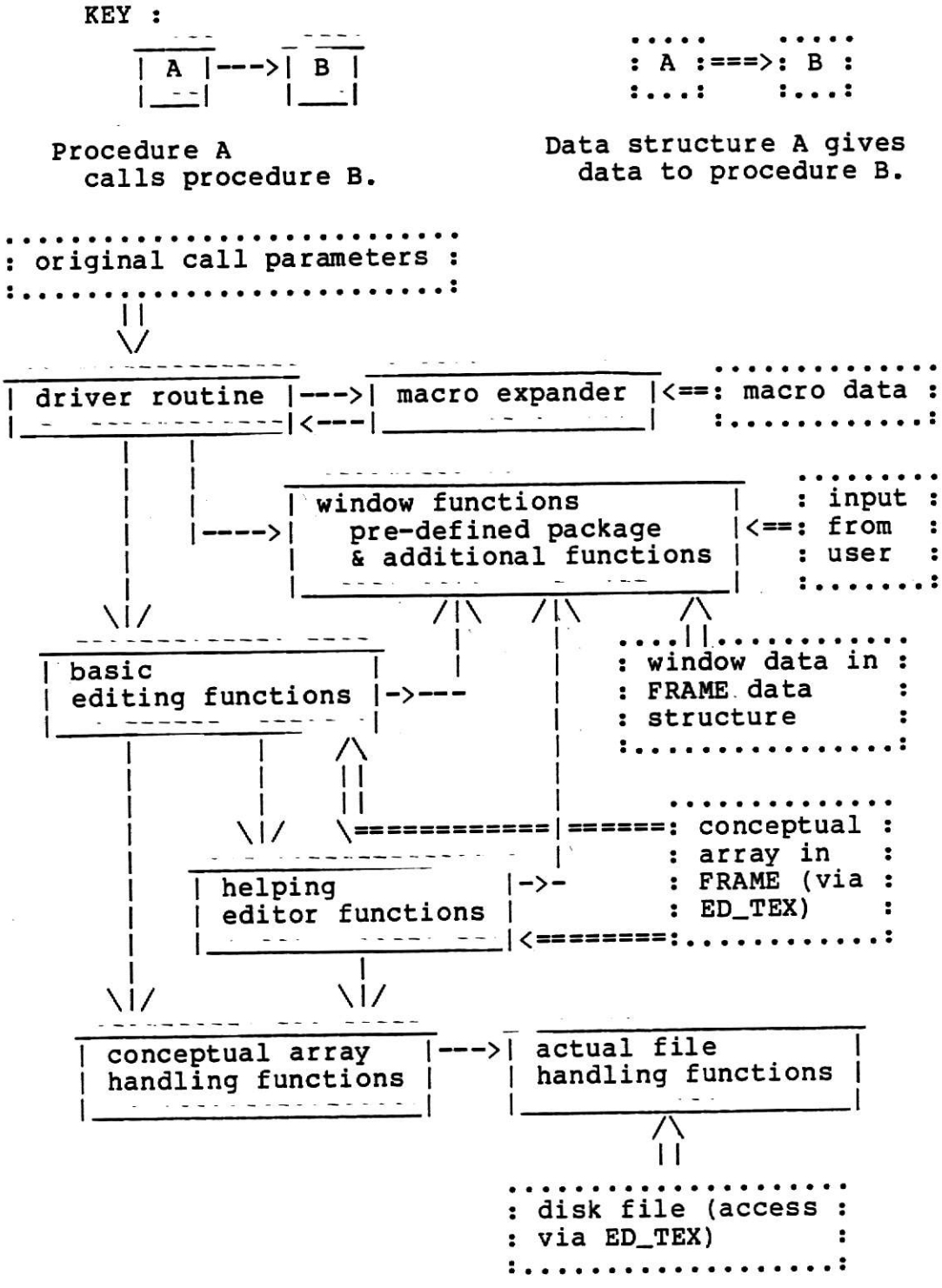
The editor is designed in a very modular fashion. This allows easy use of functions by each other, and also allows easy replacement of one part without affecting any others. Each editing command is implemented by a function. There are also some "helper" functions. These fall into two types. One type implements a basic function not typically called directly. For example, the "roll" function implements rolling the window up or down a given number of lines. This is usually part of a movement or line add/delete command. The other type of function accomplishes a part of a task. For instance, the insert and delete functions each have sub-functions which are specific for truncating or wrapping-around windows. This allows the basic "insert" function to be implemented by the same function call, whether the window truncates or wraps around with lines that are too long.

2.2 DESCRIPTION OF THE HIERARCHY OF CODE AND DATA STRUCTURES

The code and data structures are broken down hierarchically. for ease of understanding and modification. The main sections are : the driver, command processor and macro translator; the window functions; the editing functions themselves, and their helper functions; the array simulators (explained below); and the actual disk file management functions. These are described

separately below; figure 1 shows their interrelationships.

figure 1
Block diagram of the software system



2.2.1 DISCUSSION OF THE BLOCK DIAGRAM

The driver routine establishes the editing system from the parameters the program was invoked with. It then establishes the editing window, and gets further input through the window functions. If an input is not recognized, it is sent to the macro translator to be broken into basic commands. The basic commands are implemented as program functions and procedures. Some of these call helping procedures or functions extracted for reasons of clarity and readability. Both the editing functions and their helping functions access the editing window via the window functions. Both also access the editing text and line lengths in array format, via a conceptual (or VIRTUAL) array. Most of the editing text is actually stored on disk. This virtual array is provided by functions whose return values are the same as the array contents at that point would be. The file management is abstracted to file management functions, called by both array functions, which keeps track of what lines are currently in memory, and which lines are in the file. These functions, in turn, calls several helping functions of their own. This code structure is described in appendix 1, and the code listed in appendix 6 is shown in order of this structure. Appendix 5 is a discussion of how to use the functions in building a new editor.

2.3 DESCRIPTION OF THE DESIGN IN DETAIL

The code sections and data structures sketched above are described in greater detail, and their relationships shown.

2.3.1 procedures and functions

2.3.1.1 THE COMMAND SET

The editor has a small, almost minimal set of basic editing instructions and a "macro" system for expanding the command set. The commands chosen are those which accomplish the basic editing functions of movement, altering the text, accommodating more than one file and more than one window (that is, altering the user's view of the edited material), and either saving the changed text or re-instating the old version.

As mentioned earlier, text addition and line deletion were implemented as basic functions, in addition to character addition and deletion. In the case of text addition, this is a convenience, done because text addition is such a common action. What is referred to here is addition until a terminating character, such as <escape>, is entered. It can consist of the addition of any number of lines. Addition of new lines is incorporated into the character addition routine when a new-line character is entered.

Line deletion is implemented as a basic function in addition to simple character deletion, rather than having the last character delete on a line delete the line. This preserves the conceptual distinction between deleting the last character from a

line (leaving an empty line), and deleting the line itself (closing up the hole in the text).

One command which cannot be built from other commands is a pattern search command. The current pattern search uses an extremely simple pattern matching algorithm.

All the lower level functions, especially the helping functions, assume that reasonable parameters have been given to them. For this reason, the higher level functions must check their input values before committing to an action. For instance, if the scrolling functions are called on to scroll to a negative line, they will scroll whatever garbage appears before the editing text in memory onto the screen. This forces the upper level movement routines to check for edge-of-text conditions. Note, however, that the lower level functions will also extend lines and generate new lines if told to move the current editing position beyond the end of the current line, or of the file. This allows the higher level functions to decide whether the areas beyond the already-entered text are not accessible without explicitly entering new characters, or whether they are considered defined as blank areas by default.

The chosen commands are now listed. For movement : up, down, left, right, search, absolute goto. For altering the text : character insert, character delete, text insert, line delete. For altering the user's view of the edited material : create window, close window, redraw window, edit new file. For ending editing session : save new version, kill new version. They are described in more detail in appendix 3.

2.3.1.2 THE MACRO SYSTEM

The "macro" system would ideally be more than text replacement, with logic and parameters allowing the creation of significant new commands and meta-commands -- i.e., a command which will build new commands of a certain type. However, this is a very complex and difficult task, and the system currently implemented is only a step toward such a system. Some ideas for a more flexible macro system are presented in appendix 2.

The macro system is a separate procedure which is called by the driver, and which calls the overall driver routine to execute its subcommands. Specifically, the driver routine knows the basic editing functions by some name. Any command input not recognized by the driver is sent to the macro translator. If it is a valid macro, the driver is sent the expansion; if necessary, a piece at a time is sent. Note that if one of the pieces is itself a macro, the recursive call cycle will continue. (A test for circularity would be a worthwhile addition to this system.) This allows varying types of macro calls and usages to be tried without having to unscramble the macro system from the driver, and vice versa. If a new macro command system is designed, only the macro translator need be replaced.

This set-up for the macro execution has a disadvantage : if an editing function used by a macro is not defined, this will not be discovered until its time for execution comes up. This may cause a partially-executed macro command to fail. To prevent this would require each macro to test for the existence of each of its

subcommands, and then each of their subcommands, until system primitive commands were reached. This would happen on each recursive entry of the macro translator, unless a special data structure showing whether the current command is a direct command or part of a macro expansion were defined and examined each time. This, however, would prevent commands with conditional sections from being executed unless the functions needed by all possible execution paths were in memory. The test for subcommand existence would not even be able to direct its search based on knowledge of the required conditions, since intermediate commands in the macro might change those conditions from their current state. This is a problem which LISP, and most interpretive systems, share.

It would be possible, and might even be feasible, to keep a difference file of changes made to the editing file during each macro execution, and then apply the differences to the editing file when the macro ends. This would require a large amount of programming effort for rather small benefits, although there may be circumstances where this is warranted.

The current macro system includes a way of sequencing several commands, a way of repeating a command a given number of times, a facility for naming macros for later recall and execution, a conditional execution and conditional looping construct, and some simple comparison functions to control the conditional constructs.

It is often necessary to add some predetermined literal text to the edited text during a macro execution. It is also possible that macros may be needed which could drop into an input command, to allow the user to enter different text at different locations.

For this reason, an "add literal" facility is included in the macro system, in addition to the ability to call the input functions.

2.3.1.3 THE EDITING FUNCTIONS

Each editing function is implemented by a separate function, which is passed a "frame." A "frame" is a data structure with the screen window description, the editing text description, and other data needed to perform functions on the file showing in a particular editing window. This allows explicit passing to the function of all necessary data, so that there is no hidden inheritance precedence to be interpreted from the code, and no hidden or unexpected interaction between functions, especially functions that call each other. Since C passes parameters by value, it is necessary to pass a reference to the frame, instead of the frame itself, to allow changes made to the frame to last after procedure and function returns.

Implementing these editing functions as a set of independent functions and procedures allows the complete reconfiguration of the system by simply writing a new driver routine to call the functions as needed. It also allows new functions to call old ones, if desired.

The functions which implement commands are designed to work on an editing text stored as an in-memory array, and a parallel line-length array. This allows the designer of new commands to work with a very simple conceptual model, without having to worry

about file management constraints. There are file-management function calls which replace the array references. These functions ensure that the needed line is in memory, and return a pointer to the line or line-length needed. To access a line in the edited text, the array reference edt->e_txt[line_no] is replaced by the function call t_line(edt,line_no), which returns a pointer to the line, now guaranteed to be in memory. To access the line length, the array reference edt->e_tx_len[line_no] is replaced by the function call t_len(edt,line_no), which returns a pointer to an integer. It does not return the integer value itself because the function may need to change the value, as for example when adding or deleting characters. Note that two lines may not be referred to at the same time, as the function may rotate one out of memory to access the other. These functions take advantage of the fact that a string (or array) in C is a pointer anyway, so a function returning a pointer can be used the same as an array.

There are also file management function calls to read a file in and establish the data needed to provide its virtual editing array, and to write a file out from the FILE data structure and its temporary file.

2.3.1.4: FILE HANDLING

The files are conceptually handled in the same way that Theodore Socolofsky's editor <24> handled its files (see figure 2a). Two temporary stack files and an in-memory file section are maintained. One of the files is a reversed file of the editing

text already passed (the backward looking file), the other a stack of the editing text in front of the current array contents (the forward looking file). The in-memory section is split into the working array, the forward buffer not yet read into the array, and the backward buffer pushed out of the array, but not yet out of memory, by forward movement. As the editing location moves forward, lines are read from the forward buffer into the array, and (as space is needed) pushed from the array to the backward buffer. As the buffers fill and empty, they are read and written to their files. Moving backward, lines are read from the backward buffer and file, and pushed into the forward buffer, to be written back into the forward file.

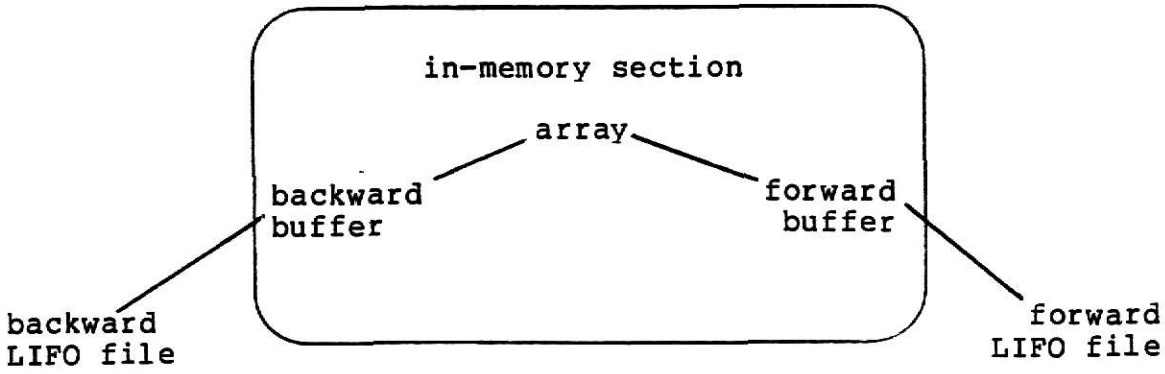
This file schema is appropriate in some ways for an editor, since most editing movement is one or a few lines at a time. For this kind of local movement, it is not grossly expensive to read the lines and write them out as the editing location moves in the file. However, repositioning from the start to the end of a large file would force the editor to read and write hundreds or thousands of lines that are not needed for the job at hand. This is inefficient.

The use of in-memory pointers to represent file structure allows, at minimal memory cost, arbitrary repositioning in the file without having to read long sequences from one stack and write them onto the next. Instead of reading and writing the lines, the in-memory records (see figure 2b) are scanned until the appropriate location in the file is found. The current buffer contents are written to disk, and the new location is read in

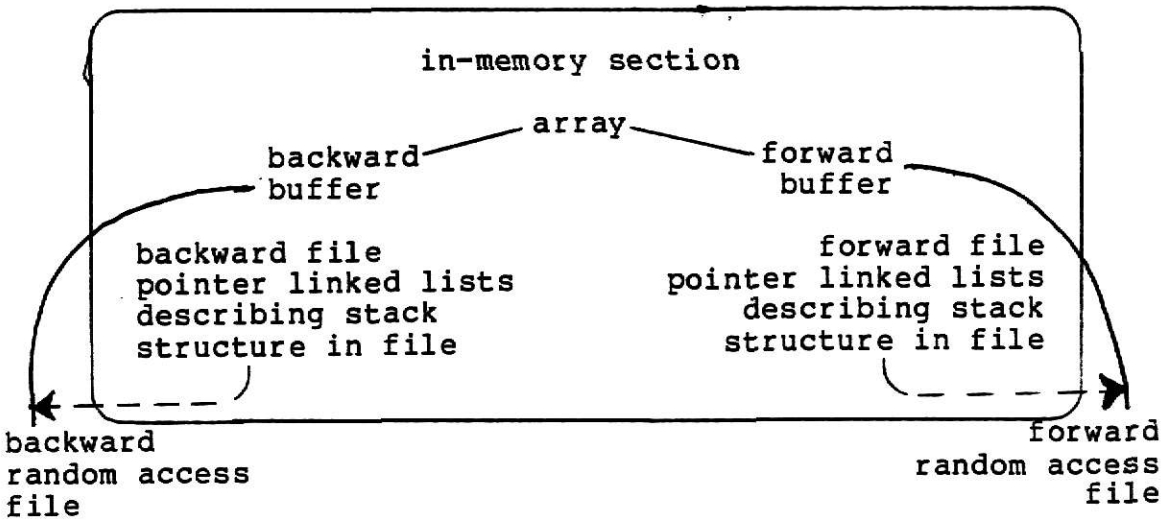
without having to disk access all the lines in between. The file segment length is a multiple of 512 for efficiency. Lines are not allowed to overlap segments, to allow easy insertion of new segments. Since the file order is kept in a linked list, lines can be added without having to move the whole disk file if the array size is exceeded; new file segments can simply be inserted into the list.

figure 2
Disk file schema -- conceptual and actual

2a : conceptual



2b : actual



2.3.1.5 WINDOW FUNCTIONS

The actual screen manipulation will be done by calls to the package "curses," already defined <3> for window definition and usage. This allows the editing program to concern itself with the screen and windows on a more conceptual level, without having to define the details of movement and window manipulation. A few of the basic functions needed are missing, most noticeably a general scrolling function. This, along with functions to fill parts of the window both with and without scrolling from an array, have been written into the editor. If a line insert and delete had been available, they would have been useful; not having them, the window filling functions were simply used to overwrite the old window, having shifted the lines up or down one as necessary.

2.3.2 permanent data structures : FRAME and EDIT-TEXT

figure 3
Contents of the FRAME and EDIT-TEXT data structures

```
*FRAME
  *window :
    *actual window structure
    *window size
    *current window location
    *offset of lines from 0
  *current text location
  *pointer to EDIT-TEXT
  *status information for window

*EDIT-TEXT
  *conceptual arrays -- text and line lengths :
    *text array
    *array of line lengths
    *length of arrays (conceptual)
    *current in-memory start line
    *number of lines in memory
  *file information :
    *file identifier number
    *file name
    *part of file before in-memory array :
      *linked list of segments used, in line number order
      *number of lines in backward buffer
      *amount left in backward buffer
    *part of file after in-memory array :
       (same information as for part of file before array).
      *linked list of segments used, in line number order
      *number of lines in backward buffer
      *amount left in backward buffer
```

2.3.2.1 THE FRAME

The window structure defined in the CURSES package is not sufficient for the needs of this editor. There is need for ready information about where the window is on the screen, where the cursor is, and the window's size. Also, since more than one window may open onto the same text, the window must keep track of where in the text the window is, and have a pointer to the possibly shared editing text.

The "FRAME" structure is defined around the window structure to include the needed data. It is not called "window," even though it is the data structure for the editing window, because the term "window" has already been taken by the screen control package. Since the frame of a window holds the window and supports it, the term seemed appropriate. It has no connection with the "frame" of Artificial Intelligence research.

The FRAME structure holds redundant data about the window structure. This was to keep data access consistent. It was felt that having to reference the structure itself for some information, and the window it points to for other information, would be confusing and inconsistent.

2.3.2.2 THE EDIT-TEXT

Conceptually, the editing text is simply an array of character strings. However, it is necessary to know at least the number of lines currently in the array. Since it is frequently necessary to know the number of characters in a line, an array

containing the character count for each line is also included. It is questionable whether this is really worth it, or whether a function that would count the characters on a line when needed would be more efficient; however, this decision had to be made fairly early in the design process. Changing over would be fairly easy; each reference to the length array would be replaced with a function call.

The actual array of character strings is likely to be much smaller than the actual file; so, there must be a temporary file containing the rest of the editing file. The edit text also contains the information needed about this file; this is covered in the next section.

2.3.2.3 THE FILE

Originally, the FILE was going to be a separate structure. Each FRAME would have its own EDIT-TEXT, and that EDIT-TEXT would have a pointer to the appropriate file; there could be more than one EDIT-TEXT pointing to the same file. However, it was realized that this could lead to a race condition if two windows overlapped, since part of the edited material is contained in the array in the EDIT-TEXT. If one EDIT-TEXT made a change in its internal array, that would not be mirrored in the other EDIT-TEXT's view of the material. For this reason, the file information is included in the EDIT-TEXT, and all editing in a given file, through one window or several, goes through the same instance of the EDIT-TEXT structure. This can be inefficient when

two widely separated windows are alternated between, as the file record must move back and forth.

The information kept about the file is : file identification, where in the file the array starts, how many lines are read into the array, the status of the forward and backward buffers and the buffers themselves, the next empty space in the file (to assign new segments), the forward and backward linked lists, and a list of empty segments already allocated file space (for instance, segments whose text has been read in or deleted). If there are any allocated but unused segments, they are re-used before new segments of the file space are allocated.

2.3.3 variable data structures : The COMMAND and MACRO-DEF

The COMMAND and MACRO-DEF data structures used depend on the needs of the command driver and macro system implemented. The current system's structures are described below.

2.3.3.1 COMMAND

The command could be passed to the command interpreter from the input routine as a single character, since it uses single character, immediate execution commands. However, since the macro routine will pass items of possibly arbitrary length to the command interpreter, the commands are passed as strings.

2.3.3.2 MACRO-DEF

The macro definitions are simply stored as strings whose length is the width of a screen, since definition entry is limited to one line.

CHAPTER 3

IMPLEMENTATION DESCRIPTION

3.1 ORDER OF IMPLEMENTATION

The basic editing functions were defined first. Next, a driving procedure to use these functions to provide a minimal editor was written.

At this point, the core project was complete. The editor was available for use, and for other interested people to expand and modify. However, it was still extremely minimal, and it was difficult to do significant work using it.

A second version was developed by implementing a macro system, to allow expansion and modification of the editor without examining the code or recompiling. This macro system was not extremely powerful, but shows the basic principles of how a more powerful system could be implemented in the current structure.

3.2 GENERAL IMPLEMENTATION DESCRIPTION

This system was implemented on the Interdata 8/32 at KSU in C. The C language was chosen for several reasons. For one, it is an interesting and useful language, and I wanted to learn it. A tutorial class is nice, but the only way to really learn a language inside and out is to flex it on your own on a project that tests some of its more advanced characteristics. This project, with its arrays, several-deep referenced variables, file manipulation, and basic computation requirements allows the use of some of C's best features. Also, the looping and scanning

characteristic of text processing call for C's very flexible loop constructs, especially the FOR loop.

Another reason that C was chosen is that it is a very good language for text manipulation. It has a string type, but that string type is based on arrays, and the full array manipulation capability is available. There are many system features in C that allow convenient file manipulation and i/o with the user.

Also, the window package written for the C language had many of the screen functions needed for a screen editor. Much of this package consists of macros that expand into C-language calls to procedures that are linked in at compile time. It would have been a complex undertaking to translate these to another language, even if the WINDOW data structure could be equivalently defined in both languages.

As a general rule, the implementation of the functions was guided more by what would be simple and understandable than by what would be more efficient. For example, the fileout function, which writes the file out after editing, moves to the start of the file, and then uses the helper function "pull" to pull up the next line to write to the file until there are no more lines. It happens that "pull" calls "push" when the in-memory array is full, to push a line from the array to the temporary file. Fileout could have been written more efficiently by in-line coding a scanning algorithm, but it would not have been as clear to someone reading the code what was happening.

It may be of interest to note that the file management functions -- t_line, t_len, filein and fileout -- were tested

separately, using a simple driver which allowed the user to request a particular lines in the input file, printed them out on the screen, and wrote the file to a new file for later examination to verify the fileout function. Meanwhile, the editing commands were first programmed using references to the actual arrays in the EDIT-TEXT structure; the file was only read in as far as the array could accommodate. Once the functions tested out properly, these array references were replaced with calls to t_len and t_line. These functions were first mocked up with simple versions that did no file handling, but simply returned the address of the appropriate line in the EDIT-TEXT array. This ensured that function-returned pointers were acting the same as array-element pointers. Note that, when a reference to a line in the EDIT-TEXT text array is replaced, the function call is simply used, since a character string in C is a pointer anyway; however, references to the EDIT-TEXT line length array must be replaced by a de-referenced function call. That is, the call must look like *t_len() instead of simply t_len(). This causes the program to follow the pointer returned by the function call to the integer needed. This is necessary because simply returning the value would make it impossible to alter the line length value; the address is needed to update the value as characters are added or deleted.

The full code of the editor is given in appendix 6.

3.3 SOME OF THE INTERESTING PROBLEMS IN IMPLEMENTATION

This is not a consistent section with a point, but simply some random observations about the process of programming this project.

At one point during the development of the movement routines, untraceable problems were occurring. Insertion of code to display intermediate results for debugging changed the problems occurring, even though the debugging code was simply printing variable values. I tried compiling and running several times, and got the same error each time; then, I put in changes, and the constant error changed to a new error. Finally, I tried simply putting in and taking out comments, and found that this also changed the program's behavior. I then assumed that there was some problem with the interaction of the text macros that the window package uses, its library routines, the TERMCAP package that the window package uses, and my code. I simply started over -- re-programmed all of the affected routines -- and the problem disappeared.

Some of the terminals at KSU do not properly support the windowing package, which makes this editor also unusable on them. I believe that this is because they either are assigned the wrong entry in the TERMCAP terminal description database, or because their descriptions are wrong or do not contain all of the items needed by the package. Unfortunately, there is no documentation available explaining which items in TERMCAP the window package needs; nor can we access the code to find out.

Unfortunately, I did not establish formal consistency constraints on the structure fields in the FRAME and EDIT-TEXT

variables. This resulted in some difficulty debugging, until I went through the code and made sure ex post facto that all usages of the FRAME and EDIT-TEXT kept all the affected fields up to date. I especially had trouble with end-of-item counts, such as the length-of-line arrays and the count of lines in the edited file. Some places these would be treated as if they were the count of lines or characters, other places as if they were the index of the highest item in the array; since C starts its arrays at zero, these two numbers are different by one. After I formally established my conventions on these matters, and went through the code carefully to make sure that any place that changed an item in a data structure also updated any other parts of the structure which should reflect the new state, I had little trouble in debugging the code. Like most lessons in Software Engineering, this one seems to have been learnable only by experience, no matter how often it is stated in theory and in classes.

The failure of C to differentiate integers and pointers as parameters to procedures created some interesting problems when I occasionally got parameters out of order in calling a routine.

Even though the conceptual editor model I was using as a target only breaks the screen into windows vertically, I programmed the functions assuming that horizontal breaks might also be used. This became useful in debugging. I could simply display debugging information next to the line it is relevant to, by establishing the editing window as narrower than the screen, and putting a debugging window next to it.

It would have been possible to use the WINDOW structure's

character array as the editing text. However, this would have involved trying to guess what consistency constraints would have to be maintained for the window package functions to operate properly. I felt it simpler to use an array with redundant information, especially since the window package's code is not available.

The window package function "scroll" actually moves the cursor up one line. To scroll the window, a new function had to be written. This wound up being an advantage, for two reasons : it was very educational for my learning about the C pointer reference system, and it encouraged the development of the sideways scrolling capability.

CHAPTER 4
CONCLUSIONS

4.1 SUMMARY

A screen editor has been implemented with a small command set, instead of trying to provide every possible command, and a capability for a programmer to easily expand a particular version's command set. It has multiple window and file capabilities.

The code was made as modular as possible, the overall structure being determined by a functional decomposition of the editing tasks to be implemented. This is aided by working in the C language, in which all procedures are independent, so there is no inheritance of variables. Variables are either global or local. All global variables used must be declared in each procedure that uses them. This forces the explicit inclusion of any and all information used. The editing text and window information were brought together into structures, and the editing routines were passed pointers to these structures if needed.

We have tried to include advanced features in the editor. The multiple window capability and concurrent multiple file editing are two such features. We have also tried to make sure that the design allowed for easy addition of new features as they are chosen by future experimenters.

It is hoped to provide it with a more extensive macro capability for command set extension and modification.

4.2 POSSIBLE AND USEFUL EXTENSIONS

There are a number of possible extensions to the current system which would make it even more useful and powerful. Descriptions of some ideas in this area follow.

A first priority would be expanding the macro system. A description of a very useful and powerful macro system is provided in appendix 2. The current version is a simplified version of this system. The only major type of facility not included is the ability to define macro variables. Both macro definitions with parameters and temporary control variables are valuable features. Since this basically involves creation of a new language translator, complete with variable storage allocation and invocation stacks, it was considered to be outside the scope of this project.

Another addition that could be very useful, and would not require as significant an effort, would be the addition of a macro library function. This would allow pre-defined sets of macros to be read into the system. This would also make editor comparison experimentation easier; instead of having several versions of the system, the different versions could be created by the different macro libraries.

It would be worthwhile to have several widely different macro and driver systems available for comparison. If they were simply made available in general, some interesting conclusions might be drawn by observing which were more popular.

As the system is currently implemented, if the system crashes the edited file is lost, because the pointers that make sense of

the temporary file are in live memory. It would not be difficult to change the file management functions to cache these into a parallel file, and then write a recovery routine to allow continued editing, or at least reclamation of the affected file.

A feature which has wide usefulness is text buffers. These can be used in the movement of text blocks, in creating standard paragraphs or inserting common sets of in-line commands for text processing systems. A related feature, often of even wider usefulness, is the ability to read from and send to files, so that paragraphs and text blocks can be saved from one editing session to the next, or used in several different contexts. This capability can also be used to build an index generating system.

Another popular and useful function which would require either a macro system with variables or some new basic functions is a tagging capability. This would allow the user to tag a place in the text, then add and delete lines and still be able to return to that place, even though he no longer knows its absolute address in the file. The most useful method would be to have a number of symbolic tags available, instead of a single tag for each file. The single tag, however, might be easier to implement.

A more general pattern matching algorithm could easily be implemented. This would either replace the current one, or a different search command could be made by having both pattern match algorithms available, depending on how the search command was called. It would be possible to have a wide variety of pattern matching algorithms available, all using the same basic circular search command as a calling frame.

An example of a special purpose extension that could be useful is a facility for making pica counts. That is, when letters are typeset instead of typed, they take up different widths. A pica count, for each letter, is the number of standard space units the letter takes up. The total pica count allows a better estimation of the space that a typeset piece will take up than a simple letter count. By having a pica-count function to provide the pica count for each letter, and a function to add to or subtract from the overall total count, a running total could be kept. The driver would call the adding function whenever it called a text add or subtract routine.

4.3 CONCLUSIONS

We have simultaneously shown that creating an editor for the purpose of editor experimentation is possible, and provided such an editor for Kansas State University. It is designed to be modified easily, instead of being designed to run efficiently and fast. It is still fast enough to use.

We have shown by doing so that editing functions can be considered modules within an overall framework, and developed separately. The command language used to invoke these modules is not affected by what they do. The set of basic editing functions possible is rather small. Many varying commands can be built from these finite foundations.

REFERENCES

<1> Abbott, Russell J. "Program Design by Informal English Descriptions". Communications Of The ACM, v. 26, no. 11 (November 1983), pp. 882 - 894.

<2> Adams, James L. Conceptual Blockbusting. Stanford Alumni Association, Stanford, California, 1974, in series : The Portable Stanford.

<3> Arnold, Kenneth C. R. C. "Screen Updating And Cursor Movement Optimization : A Library Package." Computer Science Division, Department Of Electrical Engineering And Computer Science, University Of California at Berkley, Berkeley, California.

<4> Badre, Albert N. "Selecting and Representing Information Structures for Visual Presentation." IEEE Transactions on Systems, Man, and Cybernetics, v. SMC-12, n. 4, March/April 1982, pp. 495 - 504.

<5> Bates, Madeline and Vittal, John. "Tools for the Development of Systems for Human Factors Experiments : An Example for the SSA." IEEE Transactions on Systems, Man, and Cybernetics, v. SMC-12, n. 2, March/April 1982, pp. 133 - 148.

<6> Benbasat, Izak and Wand, Yair. "Command Abbreviation Behavior in Human-Computer Interaction." Communications Of The ACM, v. 27, no. 4 (April 1984), pp. 376 - 383.

<7> Carroll, John M. and Thomas, John C. "Metaphor and the Cognitive Representation of Computing Systems." IEEE Transactions on Systems, Man, and Cybernetics, v. SMC-12, n. 2, March/April 1982, pp. 107 - 116.

<8> Denning, Peter J. "Smart Editors." Communications Of The ACM, v. 24, no. 8 (August 1981), pp. 491 - 493.

<9> Eastman, C. M. "A Comment on English Neologisms and Programming Language Keywords." Communications Of The ACM, v. 25, no. 12 (December 1982), pp. 938 - 940.

<10> Finseth, Craig A. "Managing Words : What Capabilities Should You Have with a Text Editor?" Byte, v. 7, no. 4 (April 1982), pp. 302 - 310.

<11> Fraser, Christopher W. "A Compact, Portable CRT-based Text Editor." Software, Practice and Experience, v. 9 (1979), 121-125.

<12> Jong, Steven. "Designing a Text Editor? The User Comes First." Byte, v. 7, no. 4 (April 1982), pp. 284 - 300.

<13> Landauer, T. K.; Galotti, K. M. and Hartwell, S. "Natural Command Names and Initial Learning : A Study of Text-Editing Terms." Communications Of The ACM, v. 26, no. 7 (July 1983), pp. 495 - 503.

<14> MacDonald, Alan. "Visual Programming." Datamation, v. 28, no. 11 (October 1982), pp. 132 - 140.

<15> Medina-Mora, Raul. Syntax-Directed Editing : Towards Integrated Programming Environments. PhD Dissertation, Department Of Computer Science, Carnegie-Mellon University, 1982.

<16> Miara, Richard J.; Musselman, Joyce A.; Navarro, Juan A. and Shneiderman, Ben. "Program Indentation And Comprehensibility." Communications Of The ACM, v. 26, no. 11 (November 1983), pp. 861 - 867.

<17> Nakamoto, Yukikazu; Iwamoto, Tadahiro; Hori, Masato; Hagihara, Kenichi and Tokura, Nobuki. "An Editor For Documentation In Pi-System To Support Software Development And Maintenance." Proceedings, Sixth International Conference On Software Engineering, pp. 330 - 339.

<18> Reps, Thomas; Teitelbaum, Tim and Demers, Alan. "Incremental Context-Dependent Analysis for Language-Based Editors." ACM Transactions on Programming Languages and Systems, v. 5, no. 3 (July 1983), pp. 449 - 477.

<19> Rubin, L. F. "Syntax-Directed Pretty Printing -- A First Step Towards A Syntax-Directed Editor." IEEE Transactions on Software Engineering, v. SE-9, no. 2, (March 1983), pp. 119 - 127.

<20> Rutkowski, Chris.
"An Introduction To The Human Applications Standard Computer Interface."
Byte Magazine, v. 7, no. 10, pp. 291.

<21> Shneiderman, Ben and Mayer, Richard. "Syntactic/Semantic Interactions in Programmer Behavior : A Model and Experimental Results." International Journal Of Computer And Information Sciences, v. 8, no. 3, (June 1979), pp. 219 - 238.

<22> Shneiderman, Ben. "Direct Manipulation : A Step Beyond Programming Languages." Computer, August 1983, pp. 57 - 69.

<23> Smith, Dr. David Canfield; Irby, Charles; Kimball, Ralph; and Verplank, Bill. "Designing the Star User Interface." Byte, v. 7, no. 4, (April 1982), pp. 242 - 282.

<24> Socolofsky, Theodore John. "A Full Screen Editor Implemented In PASCAL." Masters Degree Report, Kansas State University, 1981.

<25> Teitelbaum, Tim and Reps, Thomas. "The Cornell Program Synthesizer : A Syntax-Directed Programming Environment."

Communications Of The ACM, v. 24, no. 9 (September 1981), pp. 563 - 573.

<26> Waters, R. C. "The Programmer's Apprentice : Knowledge Based Program Editing." IEEE Transactions on Software Engineering, v. SE-8, no. 1 (January 1982), pp. 1 - 12.

<27> Wile, David S. "Program Developments : Formal Explanations of Implementations." Communications Of The ACM, v. 26, no. 11 (November 1983), pp. 902 - 911.

APPENDIX 1.

Structure and partitioning of the code

Data structures :

COMMAND
MACRO-DEF
FRAME
WINDOW (usually accessed by FRAME->f_win)
EDIT-TEXT (usually accessed by FRAME->f_t)
FILE (actually part of EDIT-TEXT)

Code calling structure :

CODE PARTITION

driver
  (calls macro translator,
         edit functions

macro translator
  (calls driver)

edit functions
  (call helping edit functions,
        old and new window functions,
        virtual array functions)

helping edit functions
  (call old and new window functions)

new window functions
  (call old window functions)

virtual array functions
  (call file management functions)

file management functions
  (call helping file management functions)

helping file management functions
  (call none)

APPENDIX 2.
Some ideas toward implementation of an expansion system

A macro expansion system needs to be able to expand a single command into a series of actions. This may be a simple pre-defined series of activities, or the appropriate activities may depend on conditions in the editing environment before or during the command execution. It may be necessary to repeat a single activity a number of times, or for an unspecified number of repetitions until a target condition is obtained in the environment.

Examples of these needs are easy to devise. A simple series of actions would be to find the next instance of a word, do several character deletes to erase the entire word, then several character insertions to insert a replacement. If more than one word should be replaced with the new word -- as, for instance, if two separate data structures were being combined, and references to either should be replaced with references to the new structure -- the number of character deletes might change depending on which word had been found. The repetition of deletes shows looping behavior. This could either be a set number of repetitions, or could repeat the deletion until a blank or other delimiter was encountered.

This implies a control flow ability much like a normal general purpose programming language, with sequencing, conditional blocks, and looping structures. With macro replacements, the language is completed with subprocedure calls.

Some editing functions require parameters. Searching and text insertion are immediately obvious examples of a functions requiring more than the current text and the editing location. A macro system would therefore need to provide for some type of parameter passing into the executing macros. A minimum requirement is repetition counts. Ideally, integers, characters, and strings could be allocated and passed. Note that this will require a general memory allocation system as well, to provide space for these objects.

A further parameter that could well be included would be the ability to pass references to other macros. This would allow the definition of a structure into which varying activities would fit. For instance, looping is actually a structure (repetition) into which varying activities can be inserted. The ability to pass references to other macros would allow the user to develop his own control structures. While this facility could be easily misused, it may be better to allow a user to develop intuitively clear control structures for particular types of activities than to force him to use a pre-defined set of "elegant" control structures which may not easily fit the situation at hand.

It is necessary, for controlling the conditional structures, to have some information about the environment available. This may be as simple as functions returning the current line and cursor position, current character, and end-of-line and end-of-file status. It could easily be made more complex, especially with the addition of more complex data structures to contain the returned information.

Finally, it would be a great convenience to be able to store sets of macros in files in the system, and to be able to update their definitions if necessary. This would not only allow users to store their personally favorite or convenient commands for future use, but would allow experimenters wishing to compare apparently different editors to define them as macro sets instead of new code, and to simply read in the version they need for any given test or experiment.

The final expansion system would then be described as containing :

*Control structures
  *sequencing
  *conditionals
  *loops - repetition and conditional
*Variables
  *characters
  *strings
  *numbers
  *pointers to macros
*Functions
  *system-provided environment status functions
  *user-definable functions
*Naming and definition of macros
*Storage of macros in libraries

The system to provide these abilities would be a substantial programming effort. As an interpretive system, it would require a control flow interpreter able to select the correct execution path from the possibilities established in the macro definition. This interpreter would need to be able to call itself, to handle nested control constructs. It would also have to have available an expression interpreter, to interpret the conditions selecting the control paths and the expressions defining values to be assigned to variables. The expression interpreter would have to be able to handle nested expressions, and would need to be able to access values of variables. The variables themselves would have to be

provided, both macro parameters and variables defined inside the scope of individual macros. Typing on the variables and expressions might be provided. The expression interpreter would also need to be able to both provide the system-defined status functions, and to interpret the user function definitions to provide the values of the user's functions. A memory allocation system would have to be provided so that space for the variables is defined, and a file accessing strategy and system would need to be developed for storage of macro definitions, and alteration of previously stored definitions.

In list form, the macro system could be implemented as :

- *A system to recall the definition for a given macro command
- *A control flow interpreter
  - *interpretation of nested structures
- *An expression interpreter
  - *interpretation of nested expressions
  - *values of variables
  - *functions
    - *system-defined functions
    - *interpretation of user-defined functions
- *Variable allocation and typing
- *Filing, recall, and alteration of macro definitions

Programming such a macro system would be equivalent to defining and programming a new, small language. It would require syntax definition, semantics definition, translator design, significant data structure design, and development of appropriate memory allocation algorithms.

APPENDIX 3.
Basic Editing Functions Included

***ENTERING AND LEAVING EDITOR

EDIT : This starts the editing. It is recursively called to edit a new file, then return to the current file. It is NOT called anew to concurrently edit two files; rather, a the command driver sets up the data space for the new file under the same invocation. There is a different function, invisible to the user, that establishes the new data structure.

SAVE NEW FILE : The new version is written to the system, replacing the old version.

RESTORE OLD FILE : The new version is trashed without changing the old file.

CONVENIENT FURTHER EDIT ENTRY/LEAVING FUNCTIONS

DELAY/RESTART : stop editing and do something else, without losing temporary file.
SAVE TO NEW NAME : save the edited file under a new file name.

***ENTERING AND LEAVING EDIT WINDOWS

START NEW EDITING WINDOW : Requires a starting location on screen, width, depth; file; starting line in file. It is up to the calling driver routine to mediate collisions if editing windows overlap.

CLOSE WINDOW : The window data structure is removed from the screen. This DOES NOT automatically close the file, since there may be another window editing the same file. The driver must resolve when to close a file. It must also see to either closing all a file's windows when the file is closed, or re-opening the file when the next window is entered. If they are to be re-opened, it will be impossible to restore the old file.

REDRAW WINDOW : Available in case the user thinks that the screen has failed to echo the file's status properly.

***MOVEMENT

UP ONE LINE : Without cursor movement left and right. Optionally, it can shift left to end-of-line if current location is past it, and keep track of where it's "trying" to be.

DOWN ONE LINE : As per UP ONE LINE.

LEFT ONE SPACE : Optionally, checking for start of line.

RIGHT ONE LINE : Optionally, checking for end of line.

SEARCH FOR PATTERN : This is currently a very simple pattern match. It could be easily made more complex, as the matching function is separate.

ABSOLUTE GOTO : Handy for debugging. Could be built by going up until no longer possible, then counting down to the line. If waiting for this, bring a lunch.

CONVENIENT FURTHER MOVEMENT FUNCTIONS :

MOVE TO START OF LINE
MOVE TO END OF LINE
MOVE TO TOP OF FILE (same as ABSOLUTE GOTO 0)
MOVE TO END OF FILE (cannot do with an ABSOLUTE GOTO unless last line number is known)
TAG current location, UNTAG location, MOVE TO TAG

***ALTERATIONS

INSERT ONE CHARACTER : Not including a move right.

DELETE ONE CHARACTER : Not including a move left.

TEXT ADD : Again, the driver routine could build this up, or it could be a macro, but it would complicate things and is a very common action. Furthermore, a macro or routine in the driver would have to implement shifting the array down one space to accommodate the new line by starting at the end of the conceptual array and shifting all lines from there to the current one. This would require moving to the end of the file, including writing out the current buffers and reading in the last segment, then reading and re-writing all the file segments from the end of the file to the current position. Making this and LINE DELETE basic routines pointed up the need for a shift-array routine that would know enough to start at the end of the actual array, preventing all this unnecessary file i/o.

LINE DELETE : See comment for LINE ADD. Also, there is a difference between deleting one remaining character from a line, leaving an empty line, and deleting the line itself.

CONVENIENT FURTHER ALTERATION FUNCTIONS :

DELETE TO END OF LINE
DELETE FROM START OF LINE
DELETE TO NEXT BLANK

```
***INPUT AND OUTPUT (none included in minimal set implemented)

     READ FROM AUXILIARY FILE
     WRITE TO AUXILIARY FILE
     WRITE TO BUFFER
     READ FROM BUFFER
     CLEAR BUFFER
```

APPENDIX 4.
Manual for using the editor as configured

ENTERING THE EDITOR

The editor is invoked, like any other program, with the name of the executable code file. This may be followed with the name of the file to be edited. If it is not, the editor will ask what file is to be edited. Once it knows this, it will clear the screen, draw an editing window into the file, establish an interaction window at the top of the screen for questions and printing information, and await the first command.

IMMEDIATE COMMANDS

The following commands are executed immediately upon their entry.

a (APPEND CHARACTER) : puts a character into the file after the current cursor character.

A (APPEND TEXT) : puts several characters into the file after the current character. Ends with <escape>. <Return> will create a line break.

c (CLOSE WINDOW) : the current editing window is closed, and the other windows are expanded to use the space vacated. If this was the last window into a file, it asks whether to save the file or not.

d (DOWN) : moves cursor down one line.

e (EDIT) : establishes a new editing window. It will ask for the name of the new file to be edited, and add a window to the current set on the screen.

f (FIND) : find a pattern. It will ask for the pattern to be matched. It can currently only find an exact match.

g (GOTO) : goto a line number. It will ask for the line number.

i (INPUT CHARACTER) : input a character into the current editing position.

I (INPUT TEXT) : input text. It will enter the text input at the current editing position, until <escape> is pressed. <Return> will create a line break.

l (LEFT) : moves cursor left one character.

m (MACRO DEFINE) : see next section.

n (NEXT WINDOW) : moves editing cursor into the next window DOWN on the screen.

p (PREVIOUS WINDOW) : moves editing cursor into the next window UP on the screen.

q (QUIT) : stop editing this file without saving.

r (RIGHT) : moves cursor right one character.

s (SAVE FILE) : writes current working file out to disk.

S (SAVE AND QUIT) : writes current working file out to disk, then closes all windows into that file.

u (UP) : moves cursor up one line.

x (DELETE CHARACTER) : deletes the current character.

X (DELETE LINE) : deletes the current line.

MACRO COMMANDS

Macros can be defined using the m command. Once a macro is defined, it may be used like an immediate command, except that any macro requires a <return> or <escape> at the end to indicate that the macro name is complete. This allows, for instance, both DIE and DIET to be defined without DIE being immediately executed each time the user tries to enter DIET. Note that macros starting with c, d, e, f, g, i, I, l, m, q, r, s, S, u, x, or X will never be executed, since these are immediate commands. Also, macros should not be defined starting with (, ?, or a digit since these have special meanings to the macro interpreter. The simplest course is to make all macro names start with upper case, avoiding A, I, S, and X.

MACRO DEFINITION

When the m command is entered, it will ask for a definition line. You may enter a single line, which will replace the command from then on as you use the macro. The macro definition syntax may also be used directly.

SEQUENCING

To enter a sequence of commands, separate them with semicolons and enclose them in parentheses. Example : to sequence UP, RIGHT, and INPUT CHARACTER, enter "(u;r;i)".

REPETITION

To repeat a command a given number of times, enter an integer number, then the command. For instance, to do 15 DOWN commands,

enter "15d". To repeat the sequence UP, RIGHT, DELETE CHARACTER three times, enter "3(u:r:x)".

CONDITIONAL EXECUTION

To execute a command only if a condition is true, enter ?, the condition, a colon, and the command. The legal conditions are :

! <condition> : negation of the condition.

ch<number> : cursor is at character <number> in the line (first letter is character 1),

eol : end of line.

eot : end of text (last line).

li<number> : cursor is in line <number> (first line is line 1),

For instance, to insert text if the cursor is in the first line, enter "?1i 1:I".

CONDITIONAL LOOPING

To execute a command until a condition is false, enter ??, the condition, a colon, and the command. The same conditions are used. For example, to move to the last line, enter "??!eot:d".

APPENDIX 5.
Use of the functions and procedures of the editor in building a new editor

The functions of the code will be discussed generally in the order of the structure described in appendix 1.

Note that the code depends on the window package, which depends on the TERMLIB package. Thus, the compilation command looks like :

cc <program name> -lcurses -ltermlib

The current system also requires about 50K of memory to start up. Its file name is "edit.c", so its compilation command is

cc edit.c -lcurses -ltermlib -k 50000

The command system and macro system could easily be replaced. The new system would use and build on the basic editing functions.

At the other end of the system, the file system could be replaced with another system. No claim is made for the efficiency of the current system; it works, and it uses an easy to follow conceptual model.

A new window package might be written that would have some of the special functions needed by this system, such as a partial-screen scroll.

THE COMMAND AND MACRO FUNCTIONS

Note that, in the main routine, all the functions are declared. This prevents annoying redeclaration errors later.

Function ed_loop() basically sets up the first editing window, then loops through input until there are no more files being edited. It calls several procedures whose function are made explained by the user documentation concerning the commands that

cause them to be called.

The macro functions are a bit more complex. Macro_do() accepts the first letter of a command, then inputs the rest of it. It then parses it, if possible. It calls bexp() to parse boolean expressions. It calls Mac_select() to determine whether a name has been defined as a macro, and if so, it calls Mac_out() to return the character string that defines the macro.

THE EDITING FUNCTION PROCEDURES

Note that all of these return a value, and that that value is zero if the function failed.

set_ed_status() and en_ed_status() are not really editing functions, but they set up and take down the window package as required by the editing functions.

fr_disp() displays the frame given. It is used for refreshing frames after the screen is cleared, or if the user gets lost in the frame somehow.

nu_size() is not a basic editing function; it allows the system to redefine the length and starting location of an editing window.

edit() starts editing, using frame f, giving it edit text edt, truncation status trunc, limits status lim, and end-of-line checking status eolchk; at line number line and character chr of the edit text, with window depth deep and width wide.

up(), down(), left() and right() move the cursor in frame f in those directions, if possible.

app_cl() appends character c past the current character location in frame f. ins_cl() inserts character c before the

current character.

txt_insert() and txt_append() input and insert or append text at the current character in frame f. The text is terminated by an <escape>.

l_delete() deletes the current line in frame f.

del_cl() deletes the current character in frame f.

seek_circle() searches for pattern s in frame f. It starts at the current cursor location, and searches to the end of the file; then, it moves to the start of the file and searches to the current location. If the string is matched, it moves to that location.

search() searches from st_line through en_line, starting at character st_ch, for pattern pat using the matching algorithm in match (see below). It returns its values in done, ol and och.

get_to() moves the cursor in frame f to line l, character c.

WINDOW FUNCTIONS

skroll() moves the window one character left, right, up or down, using roll() to move the window and t_add_line() or w_add_line() to put on new lines at top and bottom.

frfill() fills the frame anew from its edit text. It uses either t_frfill(), which uses t_fil_lines(), or w_frfill(), which uses w_fil_lines(). The fil_lines functions are more general; they will fill from window line st to fin. w_fil_lines() also has a tag to determine whether the window should remain offset from the first window line of its first text line, if it is.

HELPING FUNCTIONS

Most of the helping functions are special-purpose for

particular editing functions. Those that may be of use are discussed here.

cur_char() returns the current text line character.

las_line() returns the text line number of the last line on the window.

w_line() returns the text line number of line win_address on the window.

eol_chek() moves the cursor back where on the line to where characters have been added, if end-of-line checking is in effect.

err_sound() is to signal errors. It would need to be implemented, most probably as a terminal bell signal.

match() contains the matching algorithm used by search(), and seek_circle. If you want to make them a more general pattern matching system, you need only modify match().

in_str() inputs a string from window w of length not more than l. It is returned in string variable s.

ARRAY FUNCTIONS

Note that these functions do not guarantee access to more than one line of the conceptual array at a time.

t_line() returns a pointer to the string for line target.

t_len() returns a pointer to the length record for line target.

DISK FUNCTIONS

These are the disk functions called by the array functions.

filein() reads a file (name in infile) into edit text edt.

l_in_mem() assures that line target is in the in-memory array for edit text edt.

close_file() closes down a file in an edt. It does NOT write the file out to disk.

fileout writes the file from edit text edt to the file name in f_ou_name.

DISK HELPER FUNCTIONS

These are called by the disk functions. They are not called separately.

GENERAL USE FUNCTION

Function copy(), which simply copies a string from b to a, sees use in many functions in several of the categories.

APPENDIX 6.
Code -- partitioned as described in Appendix 1.

```
#

/*********************************************************/
/* definitions of procedure and function identifiers     */
/*********************************************************/
                          /* procedures                  */
#define proc int
                          /* integer return functions    */
#define functproc int
                          /* boolean return functions    */
#define bfunctproc bool
                          /* character return functions  */
#define cfunctproc char
                          /* structure return functions  */
#define sfunctproc struct

/*******************************************/
/* include the pre-defined windows package */
/*******************************************/
#include <curses.h>

/*************************************************************************/
/* type array[n] specifies array with n elements 0..n-1.                 */
/* length of strings is 1 less than MAX_LEN -- zero at end takes a space */
/*************************************************************************/

/*************************************/
/* DEFINITION OF EDIT-TEXT STRUCTURE */
/*************************************/

#define MAX_LINES 10

#define MAX_LEN 90

#define MAX_F_NAM_LEN 100
```

```
#define SEG_SIZE 512

  /* one element in editing text's list of file segments */
struct FINDEX
 {struct FINDEX
       *fi_nxt,          /* next element in list                    */
       *fi_bak;          /* preceding element in list               */
  int  fi_l_count.       /* count of lines in this segment          */
       fi_ch_count;      /* count of characters in this segment     */
  long fi_f_add;         /* file address of this segment            */
  };

  /* the editing text structure */
struct ED_TEX
 {char e_txt [MAX_LINES][MAX_LEN]; /* the edit text -- lines x letters      */
  int  e_tx_len[MAX_LINES],        /* length of lines -- if len=8 chars 0..7 */
       e_lines_total,              /* number of lines -- if 5, lines 0..4    */

         /* information about where in-memory array is in file */
       e_offset.                   /* start index of in-memory array         */
       e_num_in,.                  /* number of lines in in-memory array     */

         /* file information, including buffers and lists of segments */
       e_f_id,.                    /* file number, temporary file            */

       e_b_slines,.                /* # lines, start (back) section of file  */
       e_b_elines,.                /* # lines, end (forward) section of file */
       e_b_schars,.                /* # chars, start (back) section of file  */
       e_b_echars;                 /* # chars, end (forward) section of file */
  char e_bf_st[SEG_SIZE],          /* start (backward) buffer                */
       e_bf_en[SEG_SIZE],          /* end (forward) buffer                   */

       e_f_name[MAX_F_NAM_LEN],    /* name of file edited                    */
       e_t_f_name[12];             /* name of temporary file                 */

  struct FINDEX
       *e_st_front.                /* points to front, start (back) section  */
       *e_st_back,.                /* points to back, start (backward) sect. */

       *e_en_front,.               /* points to front, end (forward) section */
```

```
        *e_en_back,                 /* points to back, end (forward) section  */
        *e_mt_list;                 /* points to next empty segment on list   */
  long e_nx_mt;                     /* next empty space in file not in mt_list*/
  };

/*********************************/
/* DEFINITION OF FRAME STRUCTURE */
/*********************************/

  /* status information -- can be different each frame */
struct STATUS
 {int s_truncate;     /* whether to truncate or wrap-around lines           */
  int s_limits;       /* whether to create lines past last line input       */
  int s_eol_chek;     /* whether to create blanks past last input character */
  };

struct FRAME
 {int f_w_line,       /* current line in the window (starting at 0),        */
      f_t_line,       /* current line in the editing text (start 0),        */
      f_char,         /* current character in the line (actual) (start 0),  */
      f_col,          /* current column if no end-of-line (start 0),        */
      f_st_line,      /* starting line of the window in the text            */
      f_depth,        /* length of the window (if f_depth=3, lines 0..2),   */
      f_width,        /* width of the window (if f_width=3, cols are 0..2), */
      f_offset,       /* how far right from start of line the window starts */
      f_wrap[50];     /* if wrapping lines, how many letters in line starts */
  WINDOW *f_win;      /* the editing window -- see the CURSES package       */
  struct ED_TEX *f_t; /* the edited text, including length and line lengths */
  struct
    STATUS *f_status; /* the current editor status, editing in this frame   */
  };

/*****************************************/
/* DEFINITIONS USED BY EDITING FUNCTIONS */
/*****************************************/

  /* scrolling directions*/
#define UP 1
```

```
#define DOWN 2
#define LEFT 3
#define RIGHT 4

  /* fill wrapping frame from new start of line, or old (keeping partial */
  /*      line at top, if any)                                           */
#define NEW 1
#define OLD 0

/*********************************************************/
/* DEFINITION OF ITEMS USED BY COMMAND AND MACRO SYSTEM */
/*********************************************************/

  /* minimum allowed depth of one editing window */
#define MIN_DEEP 6

  /* one element in list of frames */
struct FM_LS_ELEM
 {struct FRAME      *fml_f;          /* current frame in list */
  struct FM_LS_ELEM *fml_nxt;        /* next element in list  */
  };

  /* one element in list of macros */
struct MC_LS_ELEM
 {char               mac_name[21];          /* name of macro       */
  char               mac_body[MAX_LEN];     /* definition of macro */
  struct MC_LS_ELEM *mac_nxt;               /* next macro in list  */
  };

  /* list of frames (and macros) */
struct FRAM_LIS
 {int                frl_count;     /* count of frames being edited          */
  struct FRAME      *frl_curr;      /* current frame (NOT frame list element) */
  struct FM_LS_ELEM *frl_first;     /* first frame list element              */
  struct MC_LS_ELEM *frl_macros;    /* first in list of macros               */
  };
```

```
WINDOW *stat_win;

/*****************************************************************************/
/*************************  main driver routine  *****************************/
/*****************************************************************************/

main (argc, argv)
  int argc; char **argv;
 {int tmpi;
  char str[MAX_F_NAM_LEN];


extern WINDOW *stat_win;

proc buf_out(), close(), copy(), dlim_scan(), dshift(), e_de_allocate(),
     ed_loop(), edit(), en_ed_status(), err_sound(), f_ch_out(),
     f_de_allocate(), frfill(), get_to(), in_str(), l_in_mem(), line_on(),
     load(), nu_size(), pull(), push(), roll(), search(), seg_in(), seg_out(),
     set_ed_status(), skroll(), t_add_line(), t_donl(), t_fil_lines(),
     t_frfill(), t_lefl(), t_skrol(), t_upl(), w_add_line(), w_dlcl(),
     w_donl(), w_fil_lines(), w_frfill(), w_incl(), w_lefl(), w_mv_l(),
     w_skrol(), w_upl();

functproc cur_char(), filein(), fr_disp(), las_line(), mac_select(),
     *t_len(), w_line();

bfunctproc app_cl(), app_txt(), append(), bexp(), ch_delete(), ch_eq(),
     ch_insert(), com_do(), com_edit(), del_cl(), down(), edit_nu(),
     eol_chek(), fileout(), find(), fshut(), go_to(), ins_cl(), ins_txt(),
     l_delete(), left(), line_del(), mac_def(), macro_do(), match(),
     next_win(), no_edt(), nuline(), previous_win(), quit(), re_w_size(),
     right(), save(), seek_circle(), str_eq(), t_ritl(), txt_append(),
     txt_insert(), up(), w_r_sub(), w_ritl(), win_close();

cfunctproc digit(), *mac_out(), nonterm(), not_nl(), rdinch(),
     *t_line(), wrouch();


sfunctproc ED_TEX *nu_edt();
```

```
sfunctproc FINDEX *mt_findex(), *nu_index();

sfunctproc FM_LS_ELEM *nu_elem(), *rem_curr_frame();

sfunctproc FRAME *nu_frame();


  if (argc == 1)
     {printf("enter file name : ");
      gets(str);
      }
    else
      copy(str, argv[1]);

  set_ed_status();

  stat_win = newwin(2, COLS,.LINES - 3, 0);

  ed_loop(str);

  en_ed_status();

  }

/*************************************************************************/
/************************  end of main routine  **************************/
/*************************************************************************/
```

```
/*****************************************************************************/
/*****************************************************************************/
/***************************** command system ********************************/
/***************************** routines **************************************/
/*****************************************************************************/
/*****************************************************************************/

/* loop through editing commands until all files are closed */
proc ed_loop(fillnam)
  char *fillnam;
 {char c[MAX_LEN];
  struct FRAM_LIS fl, *fr_lis = &fl;
  int tmp1, tmp2;

  fr_lis->frl_count = 0;
  fr_lis->frl_curr = NULL;
  fr_lis->frl_first = NULL;
  fr_lis->frl_macros = NULL;

  if (com_edit(fillnam, fr_lis, 1, 1))
     {wmove(fr_lis->frl_curr->f_win, fr_lis->frl_curr->f_w_line,
                                     fr_lis->frl_curr->f_char);
      wrefresh(fr_lis->frl_curr->f_win);

      while(fr_lis->frl_count)
        if (nonterm (c[0] =  wgetch(fr_lis->frl_curr->f_win)) )
           {c[1] = '\0';
            com_do(c, fr_lis, FALSE);

            if (fr_lis->frl_count)
                fr_disp(fr_lis->frl_curr);
            }
      }
  }

/* create a new editing text */
sfunctproc ED_TEX *nu_edt(infile)
  char *infile;
 {struct ED_TEX et[2], *edt;
  int tmp1, tmp2;
```

```
  tmp1 = &(et[1]);
  tmp2 = &(et[0]);
  edt = malloc(tmp1 - tmp2);

  return(edt);
  }

/* do a command -- return true if command succeeds */
bfunctproc com_do(c, fr_lis, com_entered)
  char *c;
  struct FRAM_LIS *fr_lis;
  bool com_entered;
 {bool ret;

  switch(c[0])
   {case 'a': ret = append(fr_lis->frl_curr);          break;
    case 'A': ret = app_txt(fr_lis->frl_curr);         break;
    case 'c': ret = win_close(fr_lis);                 break;

    case 'd': ret = down(fr_lis->frl_curr);            break;
    case 'e': ret = edit_nu(fr_lis);                   break;
    case 'f': ret = find(fr_lis->frl_curr);            break;

    case 'g': ret = go_to(fr_lis->frl_curr);           break;
    case 'i': ret = ch_insert(fr_lis->frl_curr);       break;
    case 'I': ret = ins_txt(fr_lis->frl_curr);         break;

    case 'l': ret = left(fr_lis->frl_curr);            break;
    case 'm': ret = mac_def(fr_lis);                   break;
    case 'n': ret = next_win(fr_lis);                  break;

    case 'p': ret = previous_win(fr_lis);              break;
    case 'q': ret = quit(fr_lis);                      break;
    case 'r': ret = right(fr_lis->frl_curr);           break;

    case 'S': ret = save(fr_lis->frl_curr);
              if (ret)
                  quit(fr_lis);
                else
```

```
                err_sound();                                    break;
    case 's': ret = save(fr_lis->frl_curr);                     break;
    case 'u': ret = up(fr_lis->frl_curr);                       break;

    case 'w': frfill(fr_lis->frl_curr); ret = 1;                break;
    case 'x': ret = ch_delete(fr_lis->frl_curr);                break;
    case 'X': ret = line_delete(fr_lis->frl_curr);              break;

    default : ret = macro_do(fr_lis, c, com_entered);           break;
    }

  return(ret);
  }

/* command to go to the next editing window on the screen */
bfunctproc next_win(fr_lis)
  struct FRAM_LIS *fr_lis;
 {struct FR_LS_ELEM *elem;

  for (elem = fr_lis->frl_first;
        ( (elem != NULL) && (elem->fml_f != fr_lis->frl_curr) );
       elem = elem->fml_nxt);

  if (elem != NULL)
      elem = elem->fml_nxt;

  if (elem == NULL)
      elem = fr_lis->frl_first;

  fr_lis->frl_curr = elem->fml_f;

  return(1);
  }

/* command to go to the next window up on the screen */
bfunctproc previous_win(fr_lis)
  struct FRAM_LIS *fr_lis;
 {struct FR_LS_ELEM *elem;
  int tmp1, tmp2;
```

```
  tmpl = 0;
  for (elem = fr_lis->frl_first;
        ( (elem != NULL) && (elem->fml_f != fr_lis->frl_curr) );
       elem = elem->fml_nxt)
    tmpl++;

  if (tmpl)
     {tmp2 = 0;
      for (elem = fr_lis->frl_first;
            tmp2 < tmpl - 1;
            elem = elem->fml_nxt)
        tmp2++;
      }
    else
      for (elem = fr_lis->frl_first;
             ( (elem != NULL) && (elem->fml_nxt != NULL) );
            elem = elem->fml_nxt);

  fr_lis->frl_curr = elem->fml_f;

  return(1);
  }

/* remove current frame from frame list */
sfunctproc FM_LS_ELEM *rem_curr_frame(fr_lis)
  struct FRAM_LIS *fr_lis;
 {struct FM_LS_ELEM **fm_elem, *ret;

  fm_elem = &(fr_lis->frl_first);

  while( (*fm_elem != NULL) && ( ((*fm_elem)->fml_f) != fr_lis->frl_curr) )
    fm_elem = &( (*fm_elem)->fml_nxt );

  ret = *fm_elem;

  if (*fm_elem != NULL)
     {(fr_lis->frl_count)--;
      *fm_elem = (*fm_elem)->fml_nxt;
      if (*fm_elem) == NULL)
          fr_lis->frl_curr = fr_lis->frl_first->fml_f;
```

```
        else
          fr_lis->frl_curr = (*fm_elem)->fml_f;
      }
  return(ret);
  }

/* make all editing windows fit onto screen */
bfunctproc re_w_size(fr_lis)
  struct FRAM_LIS *fr_lis;
 {int tmp1, tmp2;
  struct FM_LS_ELEM *tmp_elem;
  extern WINDOW *stat_win;

  if (fr_lis->frl_count)
     {tmp1 = (LINES - 3) / (fr_lis->frl_count);

      if (tmp1 < MIN_DEEP)
         {wclear(stat_win);
          mvwprintw(stat_win, 0, 0, "WOULD CREATE TOO MANY WINDOWS");
          wrefresh(stat_win);
          return(0);
          }
        else
         {clearok(curscr, TRUE);
          tmp2 = 0;
          for (tmp_elem = fr_lis->frl_first;
               tmp_elem != NULL;
               tmp_elem = tmp_elem->fml_nxt)
           {nu_size(tmp_elem->fml_f, tmp1, tmp2, 0);
            tmp2 =+ tmp1;
            }
          }

      }
  }

/* returns true if no frame in frame list points to this edit text */
bfunctproc no_edt(fr_lis, edt)
  struct FRAM_LIS *fr_lis;
  struct ED_TEX *edt;
```

```
 {bool tmpb;
  struct FM_LS_ELEM *elem;

  elem = fr_lis->frl_first;
  tmpb = TRUE;

  while ( (elem != NULL) && (tmpb) )
   {tmpb = (elem->fml_f->f_t != edt);
    elem = elem->fml_nxt;
    }
  return(tmpb);
  }

/* shuts file, determining if should save it or just quit */
bfunctproc fshut(f)
  struct FRAME *f;
 {extern WINDOW *stat_win;
  char answer[MAX_LEN];

  answer[0] = '\0';
  while (   (answer[0] != 's') && (answer[0] != 'q')
         && (answer[0] != 'S') && (answer[0] != 'Q')   ),
   {wclear(stat_win);
    mvwprintw(stat_win, 0, 0, "last window on this file.  save or quit?");
    wmove(stat_win, 1, 0);
    wrefresh(stat_win);
    echo(); in_str(stat_win, answer, COLS); noecho();
    if ( (answer[0] == 's') || (answer[0] == 'S') ),
       {save(f);
        close_file (f->f_t);
        }
    if ( (answer[0] == 'q') || (answer[0] == 'Q') ),
        close_file (f->f_t);
    }
  free(f->f_t);
  }

/* de-allocates the frame and its OWNED sub-structures (not edit text) */
proc f_de_allocate(f)
  struct FRAME *f;
```

```
 {if (f != NULL)
     {if (f->f_win != NULL)
          delwin(f->f_win);
      if (f->f_status != NULL)
          free(f->f_status);
      free(f);
      }
  }

/* de-allocates the element, and the frame and its sub-structures (except its
                                                                    edit text) */

proc e_de_alloc(elem)
  struct FM_LS_ELEM *elem;
 {if (elem != NULL)
     {f_de_allocate(elem->fml_f);
      free(elem);
      }
  }

/* closes a window; if last window, closes file */
bfunctproc win_close(fr_lis)
  struct FRAM_LIS *fr_lis;
 {struct FM_LS_ELEM *tmp_elem;

  tmp_elem = rem_curr_frame(fr_lis);
  re_w_size(fr_lis);
  if (no_edt(fr_lis, tmp_elem->fml_f->f_t))
      fshut(tmp_elem->fml_f);
  e_de_alloc(tmp_elem);
  }

/* starts editing a new file, if there is screen space */
bfunctproc edit_nu(fr_lis)
  struct FRAM_LIS *fr_lis;
 {char nam[MAX_F_NAM_LEN];
  extern WINDOW *stat_win;
  if ( ((LINES - 3) / (fr_lis->frl_count + 1)) >= MIN_DEEP )
     {wclear(stat_win);
      mvwprintw(stat_win, 0, 0, "name of file to edit :");
      wmove(stat_win, 1, 0);
```

```
        wrefresh(stat_win);
        echo(); in_str(stat_win, nam, MAX_F_NAM_LEN); noecho();
        return(com_edit(nam, fr_lis, 1, 1));
        }
      else
       {wclear(stat_win);
        mvwprintw(stat_win, 0, 0, "TOO MANY WINDOWS");
        wrefresh(stat_win);
        return(0);
        }
  }

/* quit editing a file, all windows */
bfunctproc quit(fr_lis)
  struct FRAM_LIS *fr_lis;
 {struct FM_LS_ELEM **fm_elem, *tmp_elem;
  struct ED_TEX *tmp_edt;

  close_file (fr_lis->frl_curr->f_t);

  tmp_edt = fr_lis->frl_curr->f_t;
  free(fr_lis->frl_curr->f_t);

  for (fm_elem = &(fr_lis->frl_first);
       *fm_elem != NULL;
       /* null */
       ).
    if ((*fm_elem)->fml_f->f_t == tmp_edt)
       {if (fr_lis->frl_curr == (*fm_elem)->fml_f ).
            fr_lis->frl_curr =
                  ( (*fm_elem)->fml_nxt == NULL ?
                        NULL :
                        (*fm_elem)->fml_nxt->fml_f
                   );
        tmp_elem = *fm_elem;
        *fm_elem = (*fm_elem)->fml_nxt;
        e_de_alloc(tmp_elem);
        (fr_lis->frl_count)--;
        }
       else
```

```
        fm_elem = &( (*fm_elem)->fml_nxt );

  if (fr_lis->frl_curr == NULL)
      fr_lis->frl_curr = fr_lis->frl_first->fml_f;

  re_w_size(fr_lis);
  }


/* returns character c if it is a digit, zero otherwise */
cfunctproc digit(c)
  char c;
 {char cl = '\0';

  switch(c)
   {case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    cl = c;
    break;
    }
  return (cl);
  }

/* returns  TRUE if first part of TARGET matches PATTERN,          */
/*          FALSE if target doesn't match or ends before pattern.  */
bfunctproc ch_eq(target, pattern)
  char *target, *pattern;
 {bool ret = TRUE;

  while ((ret) && (*pattern))
     ret = (*target++ == *pattern++);
  return(ret);
  }

/* evaluates a boolean expression for the macro system */
bfunctproc bexp(c, fr_lis)
  char *c;
  struct FRAM_LIS *fr_lis;
 {int tmpi;
  bool ret;
```

```
  for (; *c == ' '; c++);

  if (*c == '!')
      ret = (! (bexp(++c, fr_lis)));
    else if (ch_eq(c, "ch"))
     {sscanf(c + 2, " %d", &tmpi);
      ret = (cur_char(fr_lis->frl_curr) == tmpi - 1);
      }
    else if (ch_eq(c, "eot"))
      ret = (fr_lis->frl_curr->f_t_line
                  >= fr_lis->frl_curr->f_t->e_lines_total - 1);
    else if (ch_eq(c, "li"))
     {sscanf(c + 2, " %d", &tmpi);
      ret = (fr_lis->frl_curr->f_t_line == tmpi - 1);
      }
    else if (ch_eq(c, "eol"))
      ret = (fr_lis->frl_curr->f_char >= *t_len(fr_lis->frl_curr->f_t,
                                          fr_lis->frl_curr->f_t_line) - 1);
    else
      ret = FALSE;
  return(ret);
  }

/* returns the body of macro # index */
cfunctproc *mac_out(fr_lis, index)
  struct FRAM_LIS *fr_lis;
  int index;
 {int tmpi;
  struct FR_LS_ELEM *tmpm;

  tmpm = fr_lis->frl_macros;
  for (tmpi = 1; tmpi < index; tmpi++)
    tmpm = tmpm->mac_nxt;
  return(tmpm->mac_body);
  }

/* scans for end of a phrase delimited by the same character that starts it */
proc dlim_scan(c_from, c_to)
  char *c_from, *c_to;
```

```
 {int tmp1;
  *c_to = '\0';
  if (*c_from)
     {for (tmp1=1; (*(c_from + tmp1)) && (*(c_from + tmp1) != *c_from); tmp1++)
        *(c_to + tmp1 - 1) = *(c_from + tmp1);
      *(c_to + tmp1 - 1) = '\0';
      }
  }

/* breaks down a macro body, or looks up its definition.  returns TRUE if */
/*      execution succeeds, FALSE if a subcommand fails or name not found */
bfunctproc macro_do(fr_lis, c, com_entered)
  struct FRAM_LIS *fr_lis;
  char *c;
  bool com_entered;

 {int tmp1, tmp2, tmp3;
  char tmpc[MAX_LEN];
  bool ret;
  extern WINDOW *stat_win;

  if (!com_entered)
     {for (tmp1 = 1;
           (tmp1 < COLS) && (c[tmp1] =
                        nonterm(wgetch(fr_lis->frl_curr->f_win)));
           tmp1++)
        if ((tmp1) && (c[tmp1]=='\b'))
          tmp1 = tmp1 - 2;
        c[tmp1] = '\0';
      }
  switch(c[0])
   {case '(':
      tmp1 = 0; /* parenthesis count */
      ret = TRUE; /* return value */
      for (tmp2 = 0; (c[tmp2] != ')') && (c[tmp2]) && (ret);tmp2 = tmp3)
       {for (tmp3 = tmp2 + 1;
             ( (c[tmp3]) &&
                ( (tmp1) || ((c[tmp3] != ';') && (c[tmp3] != ')')) )   ),
              );
             tmp3++)
```

```
        {tmpc[tmp3 - tmp2 - 1] = c[tmp3];
         if (c[tmp3] == '(')
              tmpl++;
         if (c[tmp3] == ')')
             tmpl--;
         }
       tmpc[tmp3 - tmp2 - 1] = '\0';
       ret = com_do(tmpc, fr_lis, TRUE);
       }
    break;

  case '?':
    if (c[1] != '?')
         if (bexp(c + 1, fr_lis))
            {for (tmpl = 1; *(c + tmpl) && (*(c + tmpl - 1) != ':'); tmpl++);
             copy(tmpc, c + tmpl);
             ret = com_do(tmpc, fr_lis, TRUE);
             }
           else
             ret = TRUE;
       else
        {for (tmpl = 2; *(c + tmpl) && (*(c + tmpl - 1) != ':'); tmpl++);
         copy(tmpc, c + tmpl);
         ret = TRUE;
         while ( (bexp(c + 2, fr_lis)) && (ret) )
          {ret = com_do(tmpc, fr_lis, TRUE);
           }
         }
    break;

  case '.':
    if ((c[1]=='i') && (c[2]))
       {dlim_scan(c + 2, tmpc);
        ret = TRUE;
        for (tmpl = 0; (tmpc[tmpl]) && (ret); tmpl++)
          ret = ins_cl(fr_lis->frl_curr, tmpc[tmpl]);
        }
       else if ((c[1]=='a') && (c[2]))
        {dlim-scan(c + 2, tmpc);
         ret = TRUE;
```

```
            for (tmpl = 0; (tmpc[tmpl]) && (ret); tmpl++)
              ret = app_cl(fr_lis->frl_curr, tmpc[tmpl]);
            }
          else if ((c[1]=='f') && (c[2]))
           {dlim_scan(c + 2, tmpc);
            ret = seek_circle(fr_lis->frl_curr, tmpc);
            }
          else
            ret = FALSE;
        break;

      default:
        if (digit(c[0]))
           {for (tmpl = 0; (tmpc[tmpl] = digit(c[tmpl])); tmpl++);
            sscanf(tmpc, " %d", &tmp2);
            ret = TRUE;
            for (tmp3 = 0; (tmp3 < tmp2) && (ret); tmp3++)
              ret = com_do(c + tmpl, fr_lis, TRUE);
            }
          else
            if (tmpl = mac_select(fr_lis, c))
                ret = com_do(mac_out(fr_lis, tmpl), fr_lis, TRUE);
              else
               {
                wclear(stat_win);
                mvwprintw(stat_win, 0, 0, "command %s", c);
                mvwprintw(stat_win, 1, 0, "not recognized");
                wrefresh(stat_win);
                ret = FALSE;
                }
        break;
       }
     return(ret);
     }

/* returns the index of the macro named <name> in fr_lis's macro list */
functproc mac_select(fr_lis, name)
  struct FRAM_LIS *fr_lis;
  char *name;
 {struct FM_LS_ELEM *tmpm;
```

```
  int tmpi, tmpi2;
  char *c1, *c2;
  bool found;
  extern WINDOW *stat_win;

  tmpm = fr_lis->frl_macros;
  tmpi2 = 0;
  found = FALSE;

  for (tmpi = 1; (tmpm != NULL) && (! found); tmpi++)
   {c1 = name;
    c2 = tmpm->mac_name;
    while( (found = (*c1++ == *c2++)) && ((*c1) || (*c2)) );
    if (found)
        tmpi2 = tmpi;
    tmpm = tmpm->mac_nxt;
    }
  if (tmpi2)
     {wclear(stat_win);
      mvwprintw(stat_win, 0, 0, "macro %s found", name);
      wrefresh(stat_win);
      }
  return (tmpi2);
  }

/* returns TRUE if both strings are the same, FALSE otherwise */
bfunctproc str_eq(c1, c2)
  char *c1, *c2;
 {bool ret;
  while ( (ret = (*c1 == *c2)) && (*c1) )
   {c1++;
    c2++;
    }
  return(ret);
  }

/* creates a new frame, including status but not edit text */
sfunctproc FRAME *nu_frame()
 {struct FRAME fr[2], *frme;
  struct STATUS ss[2], *st;
```

```
  int tmp1, tmp2, tmp3;

  tmp1 = &(fr[1]);
  tmp2 = &(fr[0]);
  frme = malloc(tmp1 - tmp2);

  tmp1 = &(ss[1]);
  tmp2 = &(ss[0]);
  frme->f_status = malloc(tmp1 - tmp2);

  return(frme);
  }

/* dREOTEM O IEJ\ UROHE WNMT EWEHEIT */
sfunctproc FM_LS_ELEM *nu_elem()
 {struct FM_LS_ELEM fl[2], *elem;
  int tmp1, tmp2;

  tmp1 = &(fl[1]);
  tmp2 = &(fl[0]);
  elem = malloc(tmp1 - tmp2);

  return(elem);
  }

/* command to starts editing a file */
bfunctproc com_edit(name, fr_lis, lim, eolchk)
  char *name;
  struct FRAM_LIS *fr_lis;
  int lim, eolchk;
 {int tmp1, truncate;
  char s_in[MAX_LEN];
  struct FM_LS_ELEM *fm_l_elem, *fml_tmp;
  struct ED_TEX *edt;
extern WINDOW *stat_win;

  fm_l_elem = nu_elem();

  for (fml_tmp = fr_lis->frl_first;
        (fml_tmp != NULL) && (!str_eq(fml_tmp->fml_f->f_t->e_f_name, name));
```

```
        fml_tmp = fml_tmp->fml_nxt)
    /* null statement */ ;

wclear(stat_win);

if (fml_tmp == NULL)
   {edt = nu_edt();
    tmpl = filein(name, edt);
    if (tmpl == 0)
       {free(fm_l_elem);
        free(edt);
        mvwprintw(stat_win, 1, 0, "edit failed");
        }
      else
       {if (tmpl == 1)
            mvwprintw(stat_win, 0, 0, "old file %s", name);
          else
            mvwprintw(stat_win, 0, 0, "new file %s", name);
        }
    }
  else
   {edt = fml_tmp->fml_f->f_t;
    tmpl = 1;
    mvwprintw(stat_win, 0, 0, "already editing %s", name);
    mvwprintw(stat_win, 1, 0, "  -- new window");
    }

wrefresh(stat_win);

if (tmpl)
   {mvwprintw(stat_win, 1, 0, "truncate (1) or not (0)? ");
    wmove(stat_win, 1, 25);
    wrefresh(stat_win);
    echo(); in_str(stat_win, s_in, COLS); noecho();
    sscanf(s_in," %d", &truncate);
    fm_l_elem->fml_f = nu_frame();
    edit(fm_l_elem->fml_f, edt, truncate, lim, eolchk,
                                          0, 0, LINES - 3, COLS - 10);

    fm_l_elem->fml_nxt = fr_lis->frl_first;
```

```
        fr_lis->frl_first = fm_l_elem;
        fr_lis->frl_curr = fm_l_elem->fml_f;
        (fr_lis->frl_count)++;
        re_w_size(fr_lis);
        }

    return(tmpl != 0);
    }


/* define a macro */
bfunctproc mac_def(fr_lis)
    struct FRAM_LIS *fr_lis;
   {extern WINDOW *stat_win;
    struct MC_LS_ELEM size[2], *mc_elem;
    int tmpl, tmp2;

    tmpl = &(size[1]);
    tmp2 = &(size[0]);
    mc_elem = malloc(tmpl - tmp2);
    wclear(stat_win);
    mvwprintw(stat_win, 0, 0, "macro name :");
    wmove(stat_win, 0, 14);
    wrefresh(stat_win);
    echo(); in_str(stat_win, mc_elem->mac_name, 20); noecho();
    wclear(stat_win);
    mvwprintw(stat_win, 0, 0, "macro %s definition", mc_elem->mac_name);
    wmove(stat_win, 1, 0);
    wrefresh(stat_win);
    echo(); in_str(stat_win, mc_elem->mac_body, MAX_LEN); noecho();

    mc_elem->mac_nxt = fr_lis->frl_macros;
    fr_lis->frl_macros = mc_elem;
    }

/* command to append a character */
bfunctproc append(f)
    struct FRAME *f;
   {extern WINDOW *stat_win;
```

```
  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "append character");
  wrefresh(stat_win);

  return(app_cl(f, wgetch(f->f_win)));
  }

/* command to insert a character */
bfunctproc ch_insert(f)
  struct FRAME *f;
 {extern WINDOW *stat_win;

  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "insert character");
  wrefresh(stat_win);

  return(ins_cl(f, wgetch(f->f_win)));
  }

/* command to insert text */
bfunctproc ins_txt(f)
  struct FRAME *f;
 { extern WINDOW *stat_win;

  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "input mode -- escape to end");
  wrefresh(stat_win);

  txt_insert(f);

  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "end input mode                ");
  wrefresh(stat_win);
  return(1);
  }

/* command to append text */
bfunctproc app_txt(f)
  struct FRAME *f;
 {extern WINDOW *stat_win;
```

```
  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "aPPEID HADE 22 EMYOPE TA EID");
  wrefresh(stat_win);

  txt_append(f);

  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "end append mode                    ");
  wrefresh(stat_win);

  return(1);
  }

/* command to delete a line */
bfunctproc line_del(f)
  struct FRAME *f;
 {extern WINDOW *stat_win;

  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "line delete");
  wrefresh(stat_win);

  return(l_delete(f));
  }

/* command to delete a character */
bfunctproc ch_delete(f)
  struct FRAME *f;
 {extern WINDOW *stat_win;
  bool ret;

  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "delete character");
  wrefresh(stat_win);

  return(del_cl(f));
  }

/* command to find a pattern */
```

```
bfunctproc find(f)
  struct FRAME *f;
 {char a[MAX_LEN], *al = &(a[0]);

extern WINDOW *stat_win;

  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "find : ");
  wmove(stat_win, 0, 7);
  wrefresh(stat_win);
  echo(); in_str(stat_win, al, MAX_LEN); noecho();

  return(seek_circle(f, al));
  }

/* command to go to a given line number */
bfunctproc go_to(f)
  struct FRAME *f;
 {char in[MAX_LEN];
  int l;
  extern WINDOW *stat_win;

  wclear(stat_win);
  mvwprintw(stat_win, 0, 0, "go to :");
  wmove(stat_win, 0, 8);
  wrefresh(stat_win);
  echo(); in_str(stat_win, in, COLS - 8); noecho();
  l = 0; sscanf(in, " %d", &l);
  if ( (l)
       && ( (1 <= f->f_t->e_lines_total) || (f->f_status->s_limits) )
     )
      return(get_to(f, l - 1, cur_char(f)));
    else
      return(err_sound());
  }

/* command to save a file */
bfunctproc save(f)
  struct FRAME *f;
 {extern WINDOW *stat_win;
```

```
    char y_or_n[5], al[MAX_LEN], fn[MAX_F_NAM_LEN];

    al[0]='x';
    while(   (y_or_n[0] != 'y') && (y_or_n[0] != 'n')
          && (y_or_n[0] != 'Y') && (y_or_n[0] != 'N') )
     {wclear(stat_win);
      mvwprintw(stat_win, 0, 0, "save %s", f->f_t->e_f_name);
      mvwprintw(stat_win, 1, 0, "name correct? ");
      wmove(stat_win, 1, 14);
      wrefresh(stat_win);
      echo(); in_str(stat_win, y_or_n, 4); noecho();
      if ( (y_or_n[0] == 'n') || (y_or_n[0] == 'N') )
         {wclear(stat_win);
          mvwprintw(stat_win, 0, 0, "correct name :");
          wmove(stat_win, 1, 0);
          wrefresh(stat_win);
          echo(); in_str(stat_win, fn, MAX_F_NAM_LEN); noecho();
          }
      if ( (y_or_n[0] == 'y') || (y_or_n[0] == 'Y') )
          copy(fn, f->f_t->e_f_name);
      }

    return(fileout(f->f_t, fn));
    }

/*****************************************************************************/
/************************** editing function *********************************/
/******************************** routines ***********************************/
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/

/* set window package staus for editing functions */
proc set_ed_status()
 {initscr();
  noecho();
  crmode();
  }

/* end window package */
```

```
proc en_ed_status()
 {endwin();
  }

/* refresh the display of a frame */
functproc fr_disp(f)
  struct FRAME *f;
 {
  wmove(f->f_win, f->f_w_line, f->f_char);
  wrefresh(f->f_win);
  }

/* change size of editing window to size, start location to st_y, st_x */
proc nu_size(f, size, st_y, st_x)
  struct FRAME *f;
  int size, st_y, st_x;
 {while (size <= f->f_w_line)
   {skroll(UP, f);
    (f->f_w_line)--;
    if (f->f_wrap[0] == 0)
        f->f_st_line++;
   }
  f->f_depth = size;
  delwin(f->f_win);
  f->f_win = newwin(f->f_depth, f->f_width, st_y, st_x);

  if (f->f_status->s_truncate)
      t_frfill(f);
    else
      w_fil_lines(f, 0, f->f_depth - 1, OLD);

  eol_chek(f);

  wrefresh(f->f_win);
  }

/* starts a new editing frame -- returns pointer to it */
proc edit(f, edt, trunc, lim, eolchk, line, chr, deep, wide)
  struct FRAME *f;
  struct ED_TEX *edt;
```

```
  bool trunc, lim, eolchk;
  int line, chr, deep, wide;

 {f->f_status->s_truncate = trunc;
  f->f_status->s_limits = lim;
  f->f_status->s_eol_chek = eolchk;
  f->f_w_line = 0;
  f->f_t_line = line;
  f->f_st_line = line;
  f->f_depth = deep;
  f->f_width = wide;
  f->f_offset = 0;
  f->f_t = edt;
  f->f_win = newwin(deep, wide, 0, 0);

  f->f_char = 0;
  f->f_col = 0;


  frfill(f);

  get_to(f, line, chr);
  wmove(f->f_win, f->f_w_line, f->f_char);

  return(f);
  }

/* move up one line, if possible */
bfunctproc up(f)
  struct FRAME *f;
 {bool ret;
  if (f->f_t_line)
     {if (f->f_status->s_truncate)
          t_upl(f);
        else
          w_upl(f);
      eol_chek(f);
      ret = 1;
      }
    else
```

```
        ret = err_sound();
    return(ret);
    }

/* move down one line, if possible or creating lines for movement */
bfunctproc down(f)
    struct FRAME *f;
{bool ret;
 if ((f->f_t_line < f->f_t->e_lines_total - 1)
          || (!(f->f_status->s_limits)))
     {if (f->f_status->s_truncate)
          t_donl(f);
        else
          w_donl(f);
      eol_chek(f);
      ret = 1;
      }
   else
      ret = err_sound();
    return(ret);
    }

/* move one character to the left, if possible */
bfunctproc left(f)
    struct FRAME *f;
{bool ret;
 ret = 0;
 if (f->f_t->e_lines_total)
     if (cur_char(f))
        {if (f->f_status->s_truncate)
             t_lefl(f);
           else
             w_lefl(f);
         ret = 1;
         }
       else
         err_sound();
 f->f_col = f->f_char;
 return(ret);
 }
```

```
/* move one character to right if possible, or if current character is */
/*      less than MAX_LEN - 2 and not checking end of line             */
bfunctproc right(f)
  struct FRAME *f;
 {bool ret;
  int tmp1;

  ret = FALSE;
  if ( (!(f->f_status->s_eol_chek))
         || (cur_char(f) < *t_len(f->f_t,f->f_t_line) - 1) )
     {if (f->f_status->s_truncate)
          t_ritl(f);
        else
          w_ritl(f);
      ret = TRUE;
      }
  return(ret);
  }

/* function to append a character */
bfunctproc app_cl(f, c)
  struct FRAME *f;
  char c;
 {bAAW THP1, tmp2;

  tmp1 = f->f_status->s_eol_chek;
  f->f_status->s_eol_chek = FALSE;
  right(f);
  tmp2 = ins_cl(f, c);
  f->f_status->s_eol_chek = tmp1;
  return(tmp2);
  }

/* function to insert a character */
bfunctproc ins_cl(f,c)
  struct FRAME *f;
  char c;
 {int tmp1, tmp2, tmp3;
  char *cl, b[SEG_SIZE], *buf = &(b[0]);
```

```
l_in_mem(f->f_t, f->f_t_line); /* creates to and through this line if */
                               /*      it doesn't already exist       */
if (nuline(c))
   {dshift(f);

    if ( (tmpl = cur_char(f)) < *t_len(f->f_t,f->f_t_line) )
       {copy(buf, t_line(f->f_t,f->f_t_line) + tmpl);
        copy(t_line(f->f_t,f->f_t_line + 1), buf);
        for (tmp2 = 0; *buf++ != '\0'; tmp2++);
        *t_len(f->f_t,f->f_t_line + 1) = tmp2;
        }

    t_line(f->f_t,f->f_t_line)[tmpl] = '\0';
    *t_len(f->f_t,f->f_t_line) = tmpl;
    if (f->f_status->s_truncate)
        t_fil_lines(f, f->f_w_line, f->f_depth - 1);
      else
        w_fil_lines(f, f->f_w_line, f->f_depth - 1, OLD);
    down(f);
    get_to(f, f->f_t_line, 0);
    return(1);
    }
  else
   {if ( (tmpl = *t_len(f->f_t,f->f_t_line)) < MAX_LEN - 1)
       {if (tmpl < (tmp3 = cur_char(f)))
           {cl = t_line(f->f_t,f->f_t_line);
            for (tmp2 = tmpl; tmp2 < tmp3; tmp2++)
              *(cl + tmp2) = ' ';
            *(cl + (tmpl = tmp3)) = '\0';
            *t_len(f->f_t,f->f_t_line) = tmpl;
            }
        cl = t_line(f->f_t,f->f_t_line);

        tmpl++;
        *t_len(f->f_t,f->f_t_line) = tmpl;

        tmp3 =  cur_char(f);
        while(tmpl > tmp3)
         {*(cl + tmpl) = *(cl + tmpl - 1);
```

```
            tmpl--;
            }
          *(cl + tmpl) = c;

          if (f->f_status->s_truncate)
              t_add_line(f, f->f_w_line);
            else
              w_incl(f);
          return(1);
          }
        else
          return(0);
      }
  }

/* function to insert text */
bfunctproc txt_insert(f)
  struct FRAME *f;
 {int tmpi;
  char c;
                                                  /*escape*/
  for (tmpi = 0; (c = wgetch(f->f_win)) != '\033'; )
   {switch (c)
     {case '\b':
        if (tmpi)
           {left(f);
            del_cl(f);
            tmpi--;
            wmove(f->f_win, f->f_w_line, f->f_char);
            wrefresh(f->f_win);
            }
        break;
      case '\n':
        tmpi = 0;
      default:
        if (ins_cl(f, c))
           {if (c != '\n')
               {right(f);
                tmpi++;
                }
```

```
            wmove(f->f_win, f->f_w_line, f->f_char);
            wrefresh(f->f_win);
            }
          else
            err_sound();
        break;
      }
    }
  return(1);
  }

/* function to append text */
bfunctproc txt_append(f)
  struct FRAME *f;
 {char c;
  int tmpi, tmp_eol;

  for (tmpi = 0; (c = wgetch(f->f_win)) != '\033'; )
   {switch (c)
     {case '\b':
        if (tmpi)
           {del_cl(f);
            left(f);
            tmpi--;
            wmove(f->f_win, f->f_w_line, f->f_char);
            wrefresh(f->f_win);
            }
        break;
      case '\n':
        tmpi = 0;
      default:
        if (app_cl(f, c))
           {if (c != '\n')
                tmpi++;
            wmove(f->f_win, f->f_w_line, f->f_char);
            wrefresh(f->f_win);
            }
          else
            err_sound();
        break;
```

```
      }
    }
  eol_chek(f);
  return(1);
  }

/* function to delete a line */
bfunctproc l_delete(f)
  struct FRAME *f;
 {int tmpi;
  struct ED_TEX *edt;

  edt = f->f_t;
  if (f->f_t_line < edt->e_lines_total)
     {l_in_mem(edt, f->f_t_line);
      (edt->e_num_in)--;
      (edt->e_lines_total)--;

      if (!(edt->e_lines_total))
          f->f_char = f->f_col = f->f_wrap[0] = f->f_offset = 0;

                                                  /* new num_in -- 1 less */
      for (tmpi = f->f_t_line - edt->e_offset; tmpi < edt->e_num_in; tmpi++)
       {copy(edt->e_txt[tmpi], edt->e_txt[tmpi + 1]);
        edt->e_tx_len[tmpi] = edt->e_tx_len[tmpi + 1];
        }
      if ((f->f_t_line) && (f->f_t_line >= edt->e_lines_total))
         up(f);
      if (f->f_status->s_truncate)
          t_fil_lines(f, f->f_w_line, f->f_depth - 1);
        else
         {for (tmpi = f->f_w_line; (tmpi) && (f->f_wrap[tmpi]); tmpi--);
          w_fil_lines(f, tmpi, f->f_depth - 1,  NEW);
          }
      return(1);
      }
    else
      return(0);
  }
```

```
/* function to delete a character */
bfunctproc del_cl (f)
  struct FRAME *f;
 {int *tmpl, tmp2;
  char *cl;
  if (*(tmpl = t_len(f->f_t,f->f_t_line)) > cur_char(f))
     {cl = t_line(f->f_t,f->f_t_line);
      for (tmp2 = cur_char(f); tmp2 < *tmpl; tmp2++)
        *(cl + tmp2) = *(cl + tmp2 + 1);
      *tmpl = (*tmpl) - 1;
      if (f->f_status->s_truncate)
          t_add_line(f, f->f_w_line);
        else
          w_dlcl(f);
      eol_chek(f);
      return(1);
      }
    else
      return(0);
  }

/* function to search in a frame to end of file, then from start to current */
/*      position, for a pattern 's'                                          */
bfunctproc seek_circle(f, s)
  struct FRAME *f#
  YSOR *s;
 {int ol, och = 0;
  bool done;

  search(f->f_t, f->f_t_line, f->f_t->e_lines_total, cur_char(f) + 1,
         s, &done, &ol, &och);
  if (!done)
      search(f->f_t, 0, f->f_t_line, 0, s, &done, &ol, &och);

  if (done)
     {get_to(f, ol, och);
      return(1);
      }
    else
      return(0);
```

```
  }

/* function to search an edit text from st_ch in st_line through en_line, */
/*      for pattern pat; if found, return done = TRUE and line found at   */
/*      in ol and character found at in och; if not found, return         */
/*      done = FALSE, ol and och undefined.                               */
proc search (edt, st_line, en_line, st_ch, pat, done, ol, och)
  struct ED_TEX *edt;
  int st_line, en_line, st_ch, *ol, *och;
  char *pat;
  bool *done;
 {int tmpc, tmpl, top, s_ch;
  char *cl;

  top = edt->e_lines_total;
  *done = FALSE;

  if ((s_ch = *t_len(edt,st_line) - 1) > st_ch)
      s_ch = st_ch;

  for (tmpl = st_line;
        (!(*done)) && (tmpl < top) && (tmpl <= en_line);
        tmpl++)
   {cl = t_line(edt,tmpl);
    for (tmpc = s_ch;
         (*(cl + tmpc)) && (! match(pat, 0, cl, tmpc));
         tmpc++);
    if (*(cl + tmpc))
        *done = TRUE;
    s_ch = 0;
    }
  tmpl--;
  if (*done)
     {*ol = tmpl;
      *och = tmpc;
      }
  }

/* assumes a legal line is input!*/
/* function to go to a given line number and character */
```

```
proc get_to(f, l, c)
  struct FRAME *f;
  int l,c;
 {int tmpl;

  while(f->f_t_line < l),       /* this is slow but it will create lines as */
    down(f);                    /* needed, if f->f_status->s_limits = FALSE */
  while(f->f_t_line > l),
    up(f);

  if (f->f_status->s_eol_chek)
      if ((tmpl = *t_len(f->f_t,f->f_t_line) - 1) < c)
          c = tmpl;
    else
      if ((tmpl = MAX_LEN - 2) < c)
          c = tmpl;

  if (c < 0),
      c = 0;

  while (cur_char(f) < c)
    right(f);
  while (cur_char(f) > c)
    left(f);

  return(l);
  }

/*****************************************************************************/
/************************** window routines  *********************************/
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/

/* rolls entire window */
proc skroll(direction, f),
  int direction;
  struct FRAME *f;
 {roll(direction, f, 0, f->f_depth - 1);
```

```
}

proc roll(direction, f, st, fin)
  int direction, st, fin;
  struct FRAME *f;
 {char **c_arr2, *c_arr1;
  int tmp, tmp2, tmp3, tmp4;
  c_arr2 = f->f_win->_y;
  switch(direction)
   {case UP:
      c_arr1= *(c_arr2 + st);
      for (tmp=st; tmp < fin; tmp++)
       {*(c_arr2 + tmp) = *(c_arr2 + tmp + 1);
        f->f_wrap[tmp] = f->f_wrap[tmp + 1];
        }
      *(c_arr2 + fin) = c_arr1;
      break;
    case DOWN:
      c_arr1 = *(c_arr2 + fin);
      for (tmp = fin; tmp > st; tmp--)
       {*(c_arr2 + tmp) = *(c_arr2 + tmp - 1);
        f->f_wrap[tmp] = f->f_wrap[tmp - 1];
        }
      *(c_arr2 + st) = c_arr1;
      break;
    case LEFT:
      for (tmp=st; tmp <= fin; tmp++)
       {c_arr1 = *(c_arr2 + tmp);
        tmp4 = *t_len(f->f_t,f->f_st_line + tmp) - f->f_offset;
        tmp2 = (f->f_width - 1 > tmp4 ? tmp4 : f->f_width - 1);
        for (tmp3=0; tmp3 < tmp2; tmp3++)
          *(c_arr1 + tmp3) = *(c_arr1 + tmp3 + 1);
        *(c_arr1 + tmp3) = ' ';
        };
        (f->f_offset)++;
      break;
    case RIGHT:
      for (tmp=st; tmp <= fin; tmp++)
       {c_arr1 = *(c_arr2 + tmp);
        tmp2 = *t_len(f->f_t,f->f_st_line + tmp) - f->f_offset + 1;
```

```
        if (f->f_width - 1 < tmp2)
            tmp2 = f->f_width - 1;
        for (tmp3 = tmp2; tmp3 > 0; tmp3--)
          *(c_arr1 + tmp3) = *(c_arr1 + tmp3 - 1);
        *c_arr1 = ' ';
        }
        (f->f_offset)--;
    }
  touchwin(f->f_win);
  }

/* adds a line onto a frame's window, assuming truncation */
proc t_add_line(f, win_address)
  struct FRAME *f;
  int win_address;
 {int tmp;
  char *c;
  f->f_wrap[win_address] = 0;
  wmove(f->f_win, win_address, 0);
  if (f->f_offset < *t_len(f->f_t,f->f_st_line + win_address))
     {c = t_line(f->f_t,f->f_st_line + win_address);
      line_on(f, win_address, c + f->f_offset);
      }
    else
      line_on(f, win_address," ");
  }

/* adds a line onto a frame's window, assuming wrapping around */
proc w_add_line(f, win_address)
  struct FRAME *f;
  int win_address;
 {int tmp1;
  char *c;
  tmp1 = w_line(f, win_address);
  wmove(f->f_win, win_address, 0);
  c = &(t_line(f->f_t,tmp1)[f->f_wrap[win_address]]);
  line_on (f, win_address, c);
  }

/* adds a character string onto a frame's window at an address */
```

```
proc line_on(f, win_address, c)
  struct FRAME *f;
  int win_address;
  char *c;
 {int tmp = 0;
  while ((*c) && (tmp < f->f_width))
    mvwaddch(f->f_win, win_address, tmp++, *c++);
  while (tmp < f->f_width)
    mvwaddch(f->f_win, win_address, tmp++, ' ');
  }

/* fills the frame's window from its edit text */
proc frfill(f)
  struct FRAME *f;
 {if (f->f_status->s_truncate)
      t_frfill(f);
    else
      w_frfill(f);
  }

/* fills the frame's window from its edit text, assuming truncation */
proc t_frfill(f)
  struct FRAME *f;
 {t_fil_lines(f, 0, f->f_depth - 1);
  }

/* fills lines st to fin inclusive in frame's window, assuming truncation */
proc t_fil_lines(f, st, fin)
  int st, fin;
  struct FRAME *f;
 {int top, line;
  struct ED_TEX *edt;

  edt = f->f_t;
  top = edt->e_lines_total;

  for (line = st; line <= fin; line++)
    if (line + f->f_st_line < top)
        t_add_line(f, line);
      else
```

```
        {wmove(f->f_win, line, 0);
         wclrtoeol(f->f_win);
         f->f_wrap[line] = 0;
         }
  }

/* fills the frame's window from its edit text, assuming wrapping around*/
proc w_frfill(f)
  struct FRAME *f;
 {w_fil_lines(f, 0, f->f_depth - 1, NEW);
  }

/* fills lines st to fin inclusive in frame window, assumes wrapping around */
proc w_fil_lines(f, st, fin, neworold)
  int st, fin, neworold;
  struct FRAME *f;
 {int top, line, tex_line, t_sol;
  struct ED_TEX *edt;

  edt = f->f_t;
  top = edt->e_lines_total;

  tex_line = f->f_st_line;

  if ((st == 0) && (neworold == NEW))
      t_sol = 0;
    else
      t_sol = f->f_wrap[0];

  for (line = 0; line < st; line++)
    if ((t_sol = f->f_wrap[line] + f->f_width) >= *t_len(edt,tex_line))
       {tex_line++;
        t_sol = 0;
        }

  f->f_wrap[line] = t_sol;
  if (tex_line < top)
      line_on (f, line, &(t_line(edt,tex_line)[t_sol]));
    else
      line_on (f, line, " ");
```

```
  for (; line < fin; line++)
   {if ((tex_line < top) &&
         (t_sol = f->f_wrap[line] + f->f_width) >= *t_len(edt,tex_line))
       {tex_line++;
        t_sol = 0;
        }
    if (tex_line < top)
       {f->f_wrap [line + 1] = t_sol;
        line_on (f, line + 1, &(t_line(edt,tex_line)[t_sol]));
        }
      else
       {f->f_wrap [line + 1] = 0;
        wmove(f->f_win, line + 1, 0);
        wclrtoeol(f->f_win);
        }
    }
  }

/*************************************************************************/
/*********************** editing function helper ************************/
/****************************** routines ********************************/
/************************************************************************/
/************************************************************************/
/************************************************************************/

/* current character, counting offset and wraparound */
functproc cur_char(f)
  struct FRAME *f;
 {return(f->f_char + f->f_offset + f->f_wrap[f->f_w_line]);
  }

bfunctproc w_ritl(f)
  struct FRAME *f;
 {bool ret;
  int tmpl;
  if (cur_char(f) < *t_len(f->f_t, f->f_t_line) - 1)
      w_r_sub(f);
    else    /* current char >= length - 1 */
     {if (!(f->f_status->s_eol_chek))
```

```
            {ret = TRUE;
             if (f->f_char < f->f_width - 1)
                w_r_sub(f);
               else  /* need to move to next line */
                {tmpl = f->f_wrap[f->f_w_line] + f->f_width;
                 if (f->f_w_line >= f->f_depth - 1) /* last win line */
                    {roll(UP, f, 0, f->f_w_line);
                     line_on(f, f->f_w_line, " ");
                     }
                   else    /* next win line exists */
                    {(f->f_w_line)++;
                     roll(DOWN, f, f->f_w_line, f->f_depth - 1);
                     line_on(f, f->f_w_line, " ");
                     }
                 f->f_wrap[f->f_w_line] = tmpl;
                 f->f_char = f->f_col = 0;
                 }
              }
            else
              ret = FALSE;
          }
      return(ret);
      }

    bfunctproc w_r_sub(f)
      struct FRAME *f;
     {if (f->f_char < f->f_width - 1)
          (f->f_char)++;
        else
         {w_mv_l(DOWN, f);
          f->f_char = 0;
          }
      f->f_col = f->f_char;
      return(TRUE);
      }

    /* assumes the line exists to move to */
    proc t_upl(f)
      struct FRAME *f;
     {(f->f_t_line)--;
```

```
    if (f->f_w_line)
        (f->f_w_line)--;
      else
        t_skrol(DOWN,f);
    }

/* assumes the line exists to move to */
proc w_upl(f)
  struct FRAME *f;
 {int tmp1, tmp2;
  tmp1 = f->f_wrap [f->f_w_line] + f->f_col + f->f_offset;
  (f->f_t_line)--;
  tmp2 = *t_len(f->f_t,f->f_t_line) - 1;
  tmp2 = (tmp2 < tmp1 ? tmp2 : tmp1);
  while (f->f_wrap [f->f_w_line])
    w_mv_l(UP, f);
  w_mv_l(UP, f);
  while (f->f_wrap [f->f_w_line])
    w_mv_l(UP, f);
  while (f->f_wrap [f->f_w_line] + f->f_width - 1 < tmp2)
    w_mv_l(DOWN, f);
  f->f_col = tmp1 - f->f_wrap [f->f_w_line];
  if (f->f_col > f->f_width - 1)
      f->f_char = f->f_width - 1;
  }

/* assumes the line exists to move to */
proc w_mv_l(direction, f)
  int direction;
  struct FRAME *f;
 {switch (direction)
   {case UP:
      if (f->f_w_line)
          (f->f_w_line)--;
        else
          w_skrol(DOWN, f);
      break;
    case DOWN:
      if (f->f_w_line < f->f_depth - 1)
          (f->f_w_line)++;
```

```
        else
          w_skrol(UP, f);
    }
  }

/* assumes the line exists to move to */
/* may leave f->f_char beyond end of line -- see eol_chek in procedure down */
proc t_donl(f)
  struct FRAME *f;
 {(f->f_t_line)++;
  if (f->f_w_line < f->f_depth-1)
      (f->f_w_line)++;
    else
      t_skrol(UP,f);
  }

/* assumes the line exists to move to */
/* may leave f->f_char beyond end of line -- see eol_chek in procedure down */
proc w_donl(f)
  struct FRAME *f;
 {int tmp1, tmp2;
  tmp1 = f->f_wrap [f->f_w_line] + f->f_col + f->f_offset;
  (f->f_t_line)++;
  tmp2 = *t_len(f->f_t,f->f_t_line) - 1;
  tmp2 = (tmp2 < tmp1 ? tmp2 : tmp1);          /*tmp2 <= tmp1*/
  w_mv_l(DOWN, f);                             /*tmp2 = correct f->char*/
          /* now not on first line of old line -- maybe first of next */
  while (f->f_wrap[f->f_w_line])
    w_mv_l(DOWN, f);
          /* now definitely on first of next line */
  while (f->f_wrap[f->f_w_line] + f->f_width - 1 < tmp2)
    w_mv_l(DOWN, f);
  f->f_col = tmp1 - f->f_wrap [f->f_w_line];
  if (f->f_col > f->f_width - 1)
      f->f_char = f->f_width - 1;
  }

/* assumes current character in text line > 0 */
proc t_lefl(f)
  struct FRAME *f;
```

```
 {if (f->f_char)
      (f->f_char)--;
    else
      t_skrol(RIGHT, f);
  f->f_col = f->f_char;
  }

/* assumes current character in text line > 0 */
proc w_lefl(f)
  struct FRAME *f;
 {if (f->f_char)
      (f->f_char)--;
    else
     {w_mv_l(UP, f);
      f->f_char = f->f_width - 1;
      }
  f->f_col = f->f_char;
  }

/* assumes line length already checked */
bfunctproc t_ritl(f)
  struct FRAME *f#
 {if (f->f_char < f->f_width - 1)
       (f->f_char)++;
    else
      t_skrol(LEFT, f);
  f->f_col = f->f_char;
  return(TRUE);
  }

/* assumes you won't scroll up or down without a line to pull onto window */
proc t_skrol(direction, f)
  int direction;
  struct FRAME *f;
 {char **c_arr2, *c_arrl;
  int tmp, tmp2, tmp3, tmp4, fin;
  switch(direction)
   {case UP:
      skroll(UP, f);
      (f->f_st_line)++;
```

```
  t_add_line(f, f->f_depth - 1);
  break;
case DOWN:
  skroll(DOWN, f);
  (f->f_st_line)--;
  t_add_line(f, 0);
  break;
case LEFT:
  skroll(LEFT, f);
  c_arr2 = f->f_win->_y;
  if ((fin = f->f_t->e_lines_total - f->f_st_line) > f->f_depth)
      fin = f->f_depth;
  for (tmp=0; tmp < f->f_depth; tmp++)
   {c_arr1 = *(c_arr2 + tmp);
    if (tmp < fin)
       {tmp4 = *t_len(f->f_t,f->f_st_line + tmp) - f->f_offset;
        tmp2 = (f->f_width - 1 > tmp4 ? tmp4 : f->f_width - 1);
        tmp2 = (tmp2 < 0 ? 0 : tmp2);

          *(c_arr1 + tmp2) =
            (f->f_width <= tmp4 ?
              t_line(f->f_t,f->f_st_line + tmp)[f->f_offset + tmp2] :
              ' '
              );
        }
      else
        *(c_arr1 + tmp2) = ' ';
    };
  break;
case RIGHT:
  skroll(RIGHT, f);
  c_arr2 = f->f_win->_y;
  if ((fin = f->f_t->e_lines_total - f->f_st_line) > f->f_depth)
      fin = f->f_depth;
  for (tmp=0; tmp < f->f_depth; tmp++)
   {c_arr1 = *(c_arr2 + tmp);
    if ((f->f_offset >= 0) && (tmp < fin) &&
          (f->f_offset < *t_len(f->f_t,f->f_st_line + tmp)))
        *(c_arr1) =
          t_line(f->f_t,f->f_st_line + tmp)[f->f_offset];
```

```
            else
              *(c_arr1) = ' ';
          }
      }
    }

/* assumes you won't scroll without a line to pull onto window */
proc w_skrol(direction, f)
  int direction;
  struct FRAME *f;
 {int tmp1, tmp2;
char c;
  switch (direction)
   {case UP:
      tmp1 = f->f_wrap [f->f_depth - 1] + f->f_width;
      if (tmp1 > *t_len(f->f_t,las_line(f)))
          tmp1 = 0;                                    /*tmp1 = wrap[bottom]*/
      skroll(UP, f);
      if (!(f->f_wrap[0]))
          (f->f_st_line)++;
      f->f_wrap [f->f_depth - 1] = tmp1;
      w_add_line(f, f->f_depth -1);
      break;
    case DOWN:
      if (f->f_wrap[0])
         {
          tmp1 = f->f_wrap[0] - f->f_width;
          tmp1 = (tmp1 < 0 ? 0 : tmp1);
          }
        else
         {(f->f_st_line)--;
          tmp1 = *t_len(f->f_t,f->f_st_line);
          tmp1 = (tmp1 ?
                    ( (tmp2 = tmp1 % (f->f_width)) ?
                         tmp1 - tmp2
                         : tmp1 - f->f_width)
                    : 0);
          }
      skroll(DOWN, f);
      f->f_wrap[0] = tmp1;
```

```
      w_add_line(f, 0);
    }
  }

/* last line on window -- assumes window is full */
functproc las_line(f)
  struct FRAME *f;
 {return (w_line(f, f->f_depth - 1));
  }

/* assumes current text line exists */
/* checks for past end of line -- 0 means had to pull char back */
bfunctproc eol_chek(f)
  struct FRAME *f;
 {int tmp, tmpl;
  bool ret;

  if (f->f_status->s_eol_chek)
      if (f->f_col + (tmpl = f->f_offset + f->f_wrap[f->f_w_line])
           > (tmp = *t_len(f->f_t,f->f_t_line) - 1))
         {f->f_char = (tmp < tmpl ? 0 : tmp - tmpl);
          ret = 0;
          }
        else
         {f->f_char = f->f_col;
          ret = 1;
          }
    else
      ret = 1;

  if (f->f_char > f->f_width - 1)
     {f->f_char = f->f_width - 1;
      ret = 0;
      }

  if (cur_char(f) > MAX_LEN - 2)                /* MAX_LEN - 2 because last      */
     {f->f_col = f->f_char = MAX_LEN - 2;       /*   character of a line is '\0' */
      ret = 0;
      }
```

```
  return(ret);
  }

/* causes an error sound on a bad input;must implement if desired */
proc err_sound()
 {}

/* assumes lines exist on window from start of window to given address */
/* returns the text line number of the line at the window address */
functproc w_line(f, win_address)
  struct FRAME *f;
  int win_address;
 {int tmp1, tmp2;
  tmp1 = f->f_st_line;
  for (tmp2 = 1; tmp2 <= win_address; tmp2++)
    if (!(f->f_wrap [tmp2]))
        tmp1++;
  return(tmp1);
  }

/* returns true if c is a new line character                          */
/*      allows option of recognizing other characters as new line */
bfunctproc nuline(c)
  char c;
 {return ( (c == '\n' ? 1 : 0) );
  }

/* adds a line by making sure that current line is in in-memory array, then */
/*      shifting contents of array -- don't have to shift lines on disk     */
proc dshift(f)
  struct FRAME *f;
 {struct ED_TEX *edt;
  int tmpi, tmp2;
  char buf[MAX_LEN];

  edt = f->f_t;
  if (edt->e_num_in > MAX_LINES - 1)
      if (edt->e_num_in - 1 == f->f_t_line - edt->e_offset)
          push(UP, edt);
        else
```

```
            push(DOWN, edt);
    for (tmpi = edt->e_num_in; tmpi > f->f_t_line - f->f_offset + 1; tmpi--)
     {copy(edt->e_txt[tmpi], edt->e_txt[tmpi - 1]);
      edt->e_tx_len[tmpi] = edt->e_tx_len[tmpi - 1];
      }
    edt->e_tx_len[tmpi] = 0;
    edt->e_txt[tmpi][0] = '\0';

    (edt->e_num_in)++;
    (edt->e_lines_total)++;
    }

/* does window character insert for wrapping frames */
proc w_incl(f)
  struct FRAME *f;
 {int tmpl, tmp2;
  for (tmpl = f->f_w_line;
       (tmpl < f->f_depth - 1) && (f->f_wrap [tmpl + 1]);
       tmpl++);
  if ((tmpl < f->f_depth - 1) &&
      ((tmp2 = f->f_wrap[tmpl] + f->f_width)
             <= *t_len(f->f_t,f->f_t_line)))
     {tmpl++;
      roll(DOWN, f, tmpl, f->f_depth - 1);
      f->f_wrap[tmpl] = tmp2;
      }
  w_fil_lines(f, f->f_w_line, tmpl, OLD);
  }

/* delete a character from a wrapping window's line */
proc w_dlcl(f)
  struct FRAME *f;
 {int tmpl, tmp2, lasline;
  lasline = f->f_depth - 1;
  for (tmpl = f->f_w_line;
       (tmpl < lasline) && (f->f_wrap[tmpl + 1]);
       tmpl++);
  if ((tmp2 = f->f_wrap[tmpl]) && (tmp2 >= *t_len(f->f_t,f->f_t_line)))
     {if (f->f_w_line == tmpl)
         {if (!(f->f_w_line))
```

```
                tmpl++;
            w_mv_l(UP, f);
            }
        if (tmpl < lasline)
           {roll(UP, f, tmpl, lasline);
            if ((tmp2 = f->f_wrap [lasline - 1] + f->f_width)
                 <= *t_len(f->f_t,w_line(f, lasline - 1)))
                f->f_wrap[lasline] = tmp2;
              else
                f->f_wrap[lasline] = 0;
            w_add_line(f, lasline);
            }
        if (tmpl == lasline)  /*not just else for above because wrong with */
           {f->f_wrap[lasline] = 0;                    /* window depth of 1 */
            w_add_line(f, lasline);
            }
        tmpl--;
        }
    w_fil_lines(f, f->f_w_line, tmpl, OLD);
    }

/* determine whether pattern p1 is start of line p2 */
bfunctproc match(p1, c1, p2, c2) /*pattern, then test line*/
  char *p1, *p2;
  int c1, c2;
 {char tmp1;
  bool tmp2;

  tmp2 = FALSE;

  while( (tmp1 = *((p1++) + c1))
          && (tmp2 = *((p2++) + c2))
          && (tmp2 = (tmp1 == tmp2)) );
  return(tmp2);
  }

/* input a string, ended by any terminal (see nonterm()), through a window */
proc in_str(w, s, l)
  WINDOW *w;
  char *s;
```

```
 int l;
{char c;
 int y, x, tmpl = 0;
 while ((c=nonterm(wgetch(w))) && ((tmpl < l) || (c == '\b'))),
  {y = w->_cury;
   x = w->_curx;
   if (c != '\b'),
       *(s + tmpl++) = c;
     else
       if (tmpl),
           tmpl--;
  }
 *(s + tmpl) = '\0';
 if (c != '\0'),
     mvwaddch(w, y, x, ' ');
 wmove(w, 0, 0);
 wmove(w, y, x);
 wrefresh(w);
 }

/*****************************************************************************/
/************************** array simulator ***********************************/
/****************************** routines **************************************/
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/

/* returns pointer to line # <target> in file, making sure it is in memory */
cfunctproc *t_line(edt, target)  /* replaces edt->e_txt[target] */
  struct ED_TEX *edt;
  int target;
 {
  l_in_mem(edt, target);
  return(&(edt->e_txt[target - edt->e_offset]));
  }

/* returns pointer to length entry for line # <target> in file, making sure */
/*      it is in memory                                                      */
functproc *t_len(edt, target) /* replaces edt->e_tx_len[target] */
  struct ED_TEX *edt;
```

```
  int target;
 {l_in_mem(edt, target);
  return(&(edt->e_tx_len[target - edt->e_offset]));
  }

/*****************************************************************************/
/************************* disk file handling *******************************/
/****************************** routines ************************************/
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/

/*****************************************************************/
/* filein returns :                                              */
/*      0 if cannot open temporary file                          */
/*      1 if opened old file                                     */
/*      2 if file didn't exist (new file)                        */
/*                                                               */
/*****************************************************************/
functproc filein(infile, edt)
  char *infile;
  struct ED_TEX *edt;
 {struct FINDEX **p1, **p2;
  int filin, fl, cur_in, cur_ou, bufct, lines, las_nl;
  char template[14];

  template[10]='\0';
  template[11]='\0';
  template[12]='\0';
  template[13]='\0';
  template[14]='\0';
  copy(template, "edtmpXXXXXX");

  copy(edt->e_t_f_name, mktemp(template));
  if ((edt->e_f_id = creat(edt->e_t_f_name, 00600))==-1)
      return(0);
    else
     {close(edt->e_f_id);
      edt->e_f_id = open(edt->e_t_f_name, 2);
      edt->e_st_front = edt->e_st_back = edt->e_mt_list = NULL;
```

```
        edt->e_b_slines = edt->e_b_elines = edt->e_b_schars = edt->e_b_echars
            = 0;
        copy(edt->e_f_name, infile);
        edt->e_offset = edt->e_num_in = 0;
        if ((filin = open(infile, 0))==-1)
           {edt->e_en_front = edt->e_en_back = NULL;
            edt->e_nx_mt = lseek(edt->e_f_id, 0L, 1);
            edt->e_lines_total = 0;
            /* new file */
            close(filin);
            return(2);
            }
          else
           {/* successfully opened target file */
            p1 = &(edt->e_en_front);
            p2 = &(edt->e_en_back);
            fl = edt->e_f_id;
            cur_ou = cur_in = lines = edt->e_lines_total = 0;
            bufct = read(filin, edt->e_bf_st, SEG_SIZE);
            while
              ( wrouch( edt->e_bf_en, &cur_ou, &lines, &las_nl, fl, p1, p2,
                      rdinch(edt->e_bf_st, &bufct, &cur_in, filin),
                      &(edt->e_lines_total) ));
/*note that this will end on a character zero embedded in a file*/
            edt->e_nx_mt = lseek(fl, 0L, 1);
            close(filin);
            return(1);
            }
        }
    }

/* ensures that line # <target> is in memory */
proc l_in_mem(edt, target)
  struct ED_TEX *edt;
  int target;
 {int tmp1, tmp2;
  if ((tmp1 = target - edt->e_offset) >= 0)
      if (tmp1 > MAX_LINES * 2 - 2)
          load(edt, target);
        else
```

```
            for (tmp2 = edt->e_offset + edt->e_num_in - 1;
                 tmp2 < target;
                 tmp2++)
              pull(UP, edt);
      else
       {tmp1 = -tmp1;
        if (tmp1 > MAX_LINES - 1)
            load(edt, target);
          else
            while (edt->e_offset > target)
              pull(DOWN, edt);
        }
    }

/* function to close a file                                          */
/* NOTE THAT IT DOES NOT FREE MEMORY FROM THE EDIT TEXT */
proc close_file (edt)
  struct ED_TEX *edt;

 {close(edt->e_f_id);
  unlink(edt->e_t_f_name);
  }

/* write file out to file f_ou_name (NOT in segmented-pointered format) */
bfunctproc fileout(edt, f_ou_name)
  struct ED_TEX *edt;
  char *f_ou_name;
 {int tmpi, count_out, fil_out;
  char *inbuf, outbuf[SEG_SIZE];

  count_out = 0;
  if ((fil_out = creat(f_ou_name, 00600)) != -1)
     {for (tmpi = 0; tmpi < edt->e_lines_total; tmpi++)
       {for (inbuf = t_line(edt, tmpi); *inbuf; inbuf++)
          f_ch_out(fil_out, outbuf, &count_out, *inbuf);
        f_ch_out(fil_out, outbuf, &count_out, '\n');
        }
      if (count_out)
          write(fil_out, outbuf, count_out);
      return(TRUE);
```

```
      }
    else
      return(FALSE);
  }

/*************************************************************************/
/********************* disk file handling helper *************************/
/****************************** routines *********************************/
/*************************************************************************/
/*************************************************************************/
/*************************************************************************/

/* returns 0 if c is a new line, escape or return; returns c otherwise */
cfunctproc nonterm (c)
  char c;
 {char c1;

  c1 = c;
  switch(c1)
   {case '\n':
    case '\033':
    case '\r':
      c1 = '\0';
    }
  return (c1);
  }

/* read in character from file to buffer; used by filein */
cfunctproc rdinch(buf, bufct, curr, fyl)
  char *buf;
  int *bufct, *curr, fyl;
 {if (*curr < *bufct)
      return(*(buf + ((*curr)++)));
    else
     {*bufct = read(fyl, buf, SEG_SIZE);
      *curr = 0;
      if (*bufct)
          return(*(buf + (*curr)++));
        else
          return('\0');
```

```
        }
    }

/* write character to buffer. if buffer full write to file; used by filein */
cfunctproc
        wrouch(buf, curr, lines, las_nl, fyl, st_lis, en_lis, ch_in, tot_lines)
    char *buf, ch_in;
    int *curr, *lines, *las_nl, fyl, *tot_lines;
    struct FINDEX **st_lis, **en_lis;
   {if ((ch_in == '\0') && (buf[*curr - 1] != '\n'))
       {wrouch(buf, curr, lines, las_nl, fyl, st_lis, en_lis, '\n', tot_lines);
        }
    if ((*curr == SEG_SIZE) || (ch_in == '\0'))
        buf_out(buf, curr, lines, las_nl, fyl, st_lis,en_lis);
    *(buf + ((*curr)++)) = ch_in;
    if (ch_in == '\n')
       {(*lines)++;
        (*tot_lines)++;
        *las_nl = *curr - 1;
        }
    return(ch_in);
    }

/* create a new file segment index */
sfunctproc FINDEX *nu_index()
   {struct FINDEX size[2];
    int s1,s0;
    char *a;
    s1 = &(size[1]);
    s0 = &(size[0]);
    a = malloc(s1 - s0);
    return(a);
    }

/* write a buffer out as part of temporary file; used by filein */
proc buf_out(buf, curr, lines, las_nl, fyl, st_lis, en_lis)
    char *buf;
    int *curr, *lines, *las_nl, fyl;
    struct FINDEX **st_lis, **en_lis;
   {struct FINDEX *tmp;
```

```
  char buffer[SEG_SIZE];
  int tmpi;

  tmp = NULL;
  tmp = nu_index();
  tmp->fi_nxt      = NULL;
  tmp->fi_bak      = *en_lis;
  tmp->fi_l_count  = *lines;
  tmp->fi_ch_count = *las_nl + 1;
  tmp->fi_f_add    = lseek(fyl, 0L, 1);
  if (*st_lis == NULL)
      *st_lis = tmp;
    else
      (*en_lis)->fi_nxt = tmp;
  *en_lis = tmp;
  for (tmpi = 0; tmpi <= *las_nl; tmpi++)
    buffer[tmpi] = *(buf + tmpi);
  for (;tmpi < SEG_SIZE; tmpi++)
    buffer[tmpi] = ' ';
  write(fyl, buffer, SEG_SIZE);

  for (tmpi = 0; tmpi < SEG_SIZE - *las_nl - 1; tmpi++)
    *(buf + tmpi) = *(buf + tmpi + *las_nl + 1);
  *curr = *curr - *las_nl - 1;
  *lines = 0;
  *las_nl = 0;
  }

/* dumps current in-memory array contents to file, loads from line number */
/*      <target> in file                                                   */
proc load(edt, target)
  int target;
  struct ED_TEX *edt;
 {int tmpi;

  while (edt->e_num in)
    push(DOWN, edt);
  seg_out(UP, edt);
  seg_out(DOWN, edt);
  if ((edt->e_st_back != NULL) && (edt->e_offset > target))
```

```
    {if (edt->e_en_front != NULL)
         edt->e_en_front->fi_bak = edt->e_st_back;
       else
         edt->e_en_back = edt->e_st_back;
     edt->e_st_back->fi_nxt = edt->e_en_front;
     edt->e_en_front = edt->e_st_front;
     edt->e_st_front = edt->e_st_back = NULL;
     edt->e_offset = 0;
     }
  while( (edt->e_en_front != NULL)
         && (edt->e_offset + edt->e_num_in + edt->e_en_front->fi_l_count - 1
                          < target) )
   {edt->e_offset =+ edt->e_en_front->fi_l_count;
    if (edt->e_st_back == NULL)
        edt->e_st_front = edt->e_en_front;
      else
        edt->e_st_back->fi_nxt = edt->e_en_front;
    if (edt->e_en_front->fi_nxt != NULL)
        edt->e_en_front->fi_nxt->fi_bak = NULL;
      else
        edt->e_en_back = NULL;
    edt->e_en_front->fi_bak = edt->e_st_back;
    edt->e_en_front = edt->e_en_front->fi_nxt;
    if (edt->e_st_back == NULL)
        edt->e_st_back = edt->e_st_front;
      else
        edt->e_st_back = edt->e_st_back->fi_nxt;
    edt->e_st_back->fi_nxt = NULL;
    }
  while(edt->e_offset + edt->e_num_in - 1 < target)
    pull(UP,edt);
  }

/* pulls a line down or up into the in-memory array, pushing one out to get */
/*      room if necessary                                                    */
proc pull (direction, edt)
  int direction;
  struct ED_TEX *edt;
 {char *tmpc1, *tmpc2;
  int tmpi;
```

```
switch(direction)
 {case UP:
    if (edt->e_num_in > MAX_LINES - 1)
        push(UP, edt);
    if (edt->e_b_elines == 0)
        seg_in(UP, edt);
    tmpc1 = &(edt->e_txt[edt->e_num_in][0]);
    tmpc2 = &(edt->e_bf_en[0]);
    edt->e_tx_len[edt->e_num_in] = 0;
    while(*tmpc1++ = not_nl(*tmpc2++))
     {(edt->e_b_echars)--;
      (edt->e_tx_len[edt->e_num_in])++;
      }
    (edt->e_b_echars)--;
    (edt->e_b_elines)--;
    (edt->e_num_in)++;
    tmpc1 = &(edt->e_bf_en[0]);
    for(tmpi = 0; tmpi < edt->e_b_echars; tmpi++)
      *tmpc1++ = *tmpc2++;
    if ((tmpi = edt->e_offset + edt->e_num_in) > edt->e_lines_total)
        edt->e_lines_total = tmpi;
    break;

  case DOWN:
    if (edt->e_num_in > MAX_LINES - 1)
        push(DOWN, edt);
    if (edt->e_b_slines == 0)
        seg_in(DOWN, edt);
    for (tmpi = edt->e_num_in; tmpi > 0; tmpi--)
     {copy(edt->e_txt[tmpi], edt->e_txt[tmpi - 1]);
      edt->e_tx_len[tmpi] = edt->e_tx_len[tmpi - 1];
      }
    (edt->e_num_in)++;
    (edt->e_offset)--;
    tmpc1 = &(edt->e_bf_st[ --(edt->e_b_schars) - 1]);
    while( (edt->e_b_schars) && not_nl(*tmpc1) )
     {(edt->e_b_schars)--;
      tmpc1--;
      }
```

```
        tmpc1++;
        (edt->e_b_slines)--;
        edt->e_tx_len[0] = 0;
        tmpc2 = &(edt->e_txt[0][0]);
        while(*tmpc2++ = not_nl(*tmpc1++))
          (edt->e_tx_len[0])++;
        break;
      }
    }

/* returns character ch if it is not a new line, zero otherwise */
cfunctproc not_nl(ch)
  char ch;
 {
  return ( (ch == '\n' ? '\0' : ch) );
  }

/* pushes a line up or down out of the in-memory array */
proc push(direction, edt)
  int direction;
  struct ED_TEX *edt;
 {int tmp1, tmp2;
  switch(direction)
   {case UP:
      if (edt->e_tx_len[0] + edt->e_b_schars + 1 > SEG_SIZE)
          seg_out(UP, edt);
      copy(&(edt->e_bf_st[edt->e_b_schars]), edt->e_txt[0]);
      (edt->e_b_slines)++;
      edt->e_b_schars =+ edt->e_tx_len[0];
      edt->e_bf_st[edt->e_b_schars] = '\n';
      (edt->e_b_schars)++;
      for (tmp1 = 1; tmp1 < edt->e_num_in; tmp1++)
       {copy(edt->e_txt[tmp1 - 1], edt->e_txt[tmp1]);
        edt->e_tx_len[tmp1 - 1] = edt->e_tx_len[tmp1];
        }
      (edt->e_offset)++;
      break;
    case DOWN:
      if (edt->e_tx_len[edt->e_num_in - 1] + edt->e_b_echars + 1 > SEG_SIZE)
          seg_out(DOWN, edt);
```

```
        tmp2 = edt->e_tx_len[edt->e_num_in - 1] + 1;
        for (tmp1 = edt->e_b_echars - 1; tmp1 >= 0; tmp1--)
          edt->e_bf_en[tmp1 + tmp2] = edt->e_bf_en[tmp1];
        copy(edt->e_bf_en, edt->e_txt[edt->e_num_in - 1]);
        (edt->e_b_elines)++;
        edt->e_b_echars =+ edt->e_tx_len[edt->e_num_in - 1] + 1;
        edt->e_bf_en[edt->e_tx_len[edt->e_num_in - 1]] = '\n';
        break;
      }
    (edt->e_num_in)--;
    }

/* reads a file segment from the file to one of the buffers */
proc seg_in(direction, edt)
  int direction;
  struct ED_TEX *edt;
 {struct FINDEX *tmpf;
  int tmpi;
  switch(direction)
   {case UP:
      if ((tmpf = edt->e_en_front) == NULL)
         {edt->e_bf_en[0] = '\0';
          edt->e_b_echars = 1;
          edt->e_b_elines = 1;
          }
        else
         {edt->e_en_front = tmpf->fi_nxt;
          if (edt->e_en_front == NULL)
              edt->e_en_back = NULL;
            else
              edt->e_en_front->fi_bak = NULL;
          tmpf->fi_nxt = edt->e_mt_list;
          edt->e_mt_list = tmpf;
          edt->e_b_echars = tmpf->fi_ch_count;
          edt->e_b_elines = tmpf->fi_l_count;
          lseek(edt->e_f_id, tmpf->fi_f_add, 0);
          read(edt->e_f_id, edt->e_bf_en, SEG_SIZE);
          }
      break;
    case DOWN:
```

```
        if ((tmpf = edt->e_st_back) == NULL)
           {edt->e_bf_st[tmpi] = '\0';
            edt->e_b_schars = 1;
            edt->e_b_slines = 1;
            }
          else
           {edt->e_st_back = tmpf->fi_bak;
            if (edt->e_st_back == NULL)
                edt->e_st_front = NULL;
              else
                edt->e_st_back->fi_nxt = NULL;
            tmpf->fi_nxt = edt->e_mt_list;
            edt->e_mt_list = tmpf;
            edt->e_b_schars = tmpf->fi_ch_count;
            edt->e_b_slines = tmpf->fi_l_count;
            lseek(edt->e_f_id, tmpf->fi_f_add, 0);
            read(edt->e_f_id, edt->e_bf_st, SEG_SIZE);
            }
        break;
      }
    }

/* returns an empty file index, with segment */
sfunctproc FINDEX *mt_findex(edt)
  struct ED_TEX *edt;
 {struct FINDEX *tmpf;
  char buf[SEG_SIZE];

  if ((tmpf = edt->e_mt_list) != NULL)
      edt->e_mt_list = tmpf->fi_nxt;
    else
     {tmpf = nu_index();
      tmpf->fi_f_add = edt->e_nx_mt;
      lseek(edt->e_f_id, edt->e_nx_mt, 0);
      write(edt->e_f_id, buf, SEG_SIZE);
      edt->e_nx_mt = lseek(edt->e_f_id, 0L, 1);
      }
  return(tmpf);
  }
```

```
/* writes a file segment out to its address */
proc seg_out(direction, edt)
  int direction;
  struct ED_TEX *edt;
 {int tmpi;
  struct FINDEX *tmpf;

  if ((direction == UP
        ? (edt->e_b_slines != 0)
        : (edt->e_b_elines != 0))
                                  ),
      switch(direction)
       {case UP:
          for (tmpi = edt->e_b_schars; tmpi < SEG_SIZE; tmpi++)
            edt->e_bf_st[tmpi] = '\0';
          tmpf = mt_findex(edt);
          tmpf->fi_nxt = NULL;
          tmpf->fi_bak = edt->e_st_back;
          tmpf->fi_l_count = edt->e_b_slines;
          tmpf->fi_ch_count = edt->e_b_schars;
          if (edt->e_st_front == NULL)
              edt->e_st_front = tmpf;
            else
              edt->e_st_back->fi_nxt = tmpf;
          edt->e_st_back = tmpf;
          lseek(edt->e_f_id, tmpf->fi_f_add, 0);
          write(edt->e_f_id, edt->e_bf_st, SEG_SIZE);
          edt->e_b_slines = edt->e_b_schars = 0;
          break;
        case DOWN:
          for (tmpi = edt->e_b_echars; tmpi < SEG_SIZE; tmpi++)
            edt->e_bf_en[tmpi] = '\0';
          tmpf = mt_findex(edt);
          tmpf->fi_bak = NULL;
          tmpf->fi_nxt = edt->e_en_front;
          tmpf->fi_l_count = edt->e_b_elines;
          tmpf->fi_ch_count = edt->e_b_echars;
          if (edt->e_en_back == NULL)
              edt->e_en_back = tmpf;
            else
```

```
                edt->e_en_front->fi_bak = tmpf;
            edt->e_en_front = tmpf;
            lseek(edt->e_f_id, tmpf->fi_f_add, 0);
            write(edt->e_f_id, edt->e_bf_en, SEG_SIZE);
            edt->e_b_elines = edt->e_b_echars = 0;
            break;
          }
    }

/* write the character to the buffer; if buffer full, write to file */
proc f_ch_out(file, buffer, buf_count, ch)
  int file, *buf_count;
  char *buffer, ch;
 {if (*buf_count >= SEG_SIZE)
     {write(file, buffer, *buf_count);
      *buf_count = 0;
      }
  buffer[*buf_count] = ch;
  (*buf_count)++;
  }

/*************************************************************************/
/*************************************************************************/
/************************ general use routine  ***************************/
/*************************************************************************/
/*************************************************************************/
/*************************************************************************/

/* copy string b to string a (same direction as assignment) */
proc copy(a,b)
  char *a, *b;
 {while(*a++ = *b++);}
```

A MODULARLY EXPANSIBLE MINIMAL MULTI-SCREEN EDITOR

by

Samuel Mize

B. S., Kansas State University, 1981

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

This paper describes a multi-window screen editor with a simple, modular command language.

A screen editor displays a section of the edited file on an interactive display. The cursor is moved to locate the current working position in the file, and changes to the file cause immediate changes to the displayed section. A screen editor whose screen can be divided into more than one editing area, showing either different parts of the same file or different files, has multiple windows. Each editing area is referred to as a window.

The editor described allows multi-window file access, and simultaneous access to multiple files.

Instead of a myriad of specialized commands, it has a fairly minimal command set, and facilities to expand the commands. It was designed to be modifiable, both by an individual user to suit his own taste,and by a programmer who seeks to provide a significantly different editor, as for experimentation concerning the value of different editing scheme, without a prohibitive programming effort.

This editor is useful both as a simple editor for beginners and as a tool for editor experimentation, since the code is available for alteration and recombination. Since it is designed top-down and implemented bottom-up, basic editing operations are defined by procedures and functions. Recombining them into a new editor, if desired, will simply require writing a new driver routine. It is written in C, a language which can provide both structure and efficiency.