The Design and Implementation of a Display-Oriented Editor Writing System

by

Owen Theodore Anderson

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
at the
Massachusetts Institute of Technology
January, 1979

Signature redacted

Signature of Author...
Department of Physics, January 1979

Signature redacted

Certified by...
Thesis Supervisor

Signature redacted

Accepted by...
Chairperson, Departmental Committee on Theses

The Design and Implementation of a Display-Oriented Editor Writing System

by

Owen Theodore Anderson

Submitted to the Department of Physics
on January 19, 1979 in partial fulfillment of the requirements
for the Degree of Bachelor of Science

Abstract

This thesis describes the design and implementation of an editing system in use on the MagicSix operating system at MIT's Architecture Machine Group. The foundation of the system is a simulated stack machine which implements all of the basic editing functions. This is the target machine for a LISP-like language compiler. This language is used to write a real-time display oriented editor with many advanced features.

Signature redacted

Name and Title of Thesis Supervisor: Nicholas Negroponte
Associate Professor of Computer Graphics

Table of Contents

Acknowledgments

This thesis is the culmination of a great deal of discussion and consultation with many people. I would like to thank the members of the Architecture Machine Group for supporting the work I have done on this project, the members of the Student Information Processing Board for introducing me to the world of computers and for much friendship along the way, and to the Artificial Intelligence Laboratory for adding another dimension to my view of computing at MIT.

Special thanks must go to Seth Steinberg, Lee Parks, Rich Kovalcik, Dan Weinreb, Bernie Greenberg and Richard Stallman for their invaluable advice.

Introduction

The Sine (Sine Is Not Eine) editing system described in this thesis contains three levels of description. The lowest level is the Sine Machine. This is an interpreter which processes binary machine-type code in a LISP-like object world. The next level is that provided by a language called Sine. This provides high-level control structure and uniform access to Sine Machine instructions, user defined functions and macros. The highest level is that of the editor. It provides buffers, screen management, modes, and a large number of general purpose functions. This three-level system tries to provide as much flexibility and power as possible without making simple editing chores difficult to accomplish.

This editing system was developed under the MagicSix timesharing system, running at the MIT Architecture Machine Group [MLK]. MagicSix runs on an Interdata 7/32 with segmentation hardware in 32 bit mode. It is a dynamic linking system which manages many address spaces for each process, each address space containing 16 segments. The operating system always uses 6 segments of each address space for the system, including the stack and other required segments. The remaining ten segments can be used by user programs.

The first section of the thesis describes the history and the environment in which I grew accustomed to using display-oriented editors. The next section outlines the goals for this project and justifies them in light of the historical evidence of the first section. The third section explains the system in which the editor is written. In the fourth section the editor itself is described and some details of its implementation are given. The last section relates some of the important observations made during the project and problems which came up during its development.

History

The Development of Programmable Editors

The standard editor encountered on most time-sharing systems is a very simple program which can be used to insert and correct text. Normally the editing commands are in the form of one or two character long commands and strings which describe the operations on the text being edited. One command enters insert mode which allows successive lines to be entered into the file. Edit mode is returned to by typing a special character sequence in the input text such as a blank line. This sort of editor allows the modification of text much as a keypunch operator would edit a deck of cards, with a few improvements. One can usually search for a line containing any sub-string and even to change part of a line without typing the whole line over.

Several additions were made to this kind of editor to make them more powerful and easy to use. One was the ability to perform a command over a range of lines. Thus all occurrences of a misspelled word could be corrected in one command. People wanted to do more and more complicated things; just iterating over each line wasn't good enough, or one substitution per line wasn't enough. Another step was taken to increase the power of the editor. This was to allow a set of commands to be saved away somewhere and executed once or many times later. This ability with even a simple looping and conditional facility made the editor much more powerful. Fairly complicated editing instructions could be described and performed by the invocation of a simple command to execute the saved command string. For example, find this string, delete a few characters in it, add a word after it and then find the next occurrence of the string, etc: this sort of description was possible and even quite easy. Most TECO implementations are at least this complex and some such as the one developed at ITS are much better [RMS1].

The editor had in fact become a programming language. The usefulness of a particular editor as a language depends upon what commands the editor has available and how easy it is to use them. Only a rather small subset is necessary to make any task possible, if not necessarily easy. The editor was serving a double purpose, both as an editor which received individual commands from the user, and as a programming language for making complicated changes to a text file.

This duality had its problems though. The two aspects were not always compatible in their requirements on the editor's design. This inevitably led to complication. The editor side wanted the commands to be short and powerful: one letter commands to delete or print a line, begin an insertion and other common functions; two or three letters to print the whole text file or change all occurrences of one string to another. In general, concise, powerful commands were needed to make editing easy and fast. On the other hand, programming had different constraints. The program should be easy to write, easy to read and efficient in execution. While using the same commands as were used while editing made the programs relatively easy to write, their brevity tended to make the programs very hard to read. The same compactness that made the commands easy to type meant that the editor's interpreter had a big job making explicit all the implicit information that people can so easily encode into the commands. Strings had to be parsed and saved somewhere, arguments had to be gathered up and defaults inserted; all this occurred every time a command was executed. For a programming language things should be simple and explicit so that a minimum amount of preprocessing is required. Otherwise a compiler should be employed to expand the implicit information and parse the strings and produce a simplified representation for execution.

The Development of Display-Oriented Editors

The advent of inexpensive display terminals over the last few years has sparked considerable interest in display-oriented editors. These are editors which maintain on the screen of the terminal a page of text which is updated with each command to reflect the current state of the text being edited. The command syntax is completely different in this situation. Instead of having "insert mode" and "edit mode" as conventional editors do, these editors are always inserting and editing. Any printing character that is typed goes into the text buffer and is displayed on the screen. The commands to move about in the buffer and delete text are in the form of non-printing control characters.

Not surprisingly, these editors have introduced a whole host of new problems. How should one best optimize the amount of display needed to keep the screen image up to date? What should each of the control characters do? This last question also brings up the whole issue of standardization between various implementations. The many different types of terminals which can be used with a display oriented editor vary radically in the functions they can perform and they way they are performed. This makes the job of supporting a wide variety of terminals very difficult.

All implementations are different both in appearance and in internal design, but the basic idea is clear in all. The user can see what is happening to his text at all times. The feedback for editing commands is immediate and so mistakes do not go unnoticed. Operations on the text are very intuitive: move left, add a word, go down two lines, skip over two words, delete a word. The most striking evidence for their superiority is that once people get used to using one of these editors they become addicted. The feeling of working in the dark while using an editor that is not display-oriented is a sure symptom of this addiction. The major force behind the spread of these editors seems to be programmers who move to a site where there is no display-oriented editor; often their frustration is so great that they write one for the site rather than try to use a conventional editor.

The Combination

The solution to the dilemma of how to make an editor work as a programming language and vice-versa came to me quite slowly. I must credit Richard Stallman with providing the right idea. He suggested that the editor should have a LISP-like syntax. What I couldn't understand was how it would be usable as an editor. It would be too verbose and the functions didn't seem at all LISP-like. So I put his advice to the back of my mind and implemented my first editor in the style of the TECO on the ITS PDP-10's. This had a macro facility but in no way solved the redisplay problems. TECO's command syntax consists of single or double characters optionally preceded by one or two numeric arguments and/or followed by one or more string arguments terminated by a special character (alt-mode or escape). Though this is a very nice syntax for typing quickly, I found that its efficient implementation was made very difficult by the task of parsing the input and supplying defaults. In addition, a string of TECO commands is almost unreadable.

I could not avoid the idea of a super-flexible editor syntax, like TECO's, but I could not see how to implement a reasonably efficient and readable programming language with such a syntax. Then, at some point after I learned about the real-time display-oriented editor mode in ITS TECO, I realized the solution. For the editor, you use a screen editor which has a simple command syntax and is really vastly better for editing than regular TECO or any other non-display editor. Then the language you use for programming complicated editing tasks doesn't really matter. It can, and should, be much different from the commands used for editing and be as clean and elegant as it can be made. With the realization of the fundamental separation between the programming and editing aspects of an editor, I was able to begin thinking seriously of producing a programmable editor with a reasonable (display-oriented) human interface.

I came to this realization none too fast though. The evidence around me was quite clear. The ITS TECO real-time mode had recently been greatly enhanced by the introduction of a editor called EMACS [ECC] [RMS2]. This is an editor that is in fact written in ITS TECO, the programming language, by using real-time mode which allows a macro to be invoked for every key that is typed. Thus, most printable characters just call a macro which inserts that letter. But the control keys call macros which can do much more complicated things: move to the next line, uppercase the next word, justify the current paragraph, or grind the current LISP function.

This editor became a model for several other implementations. The first was one written for the LISP Machine at MIT's Artificial Intelligence Laboratory by Daniel Weinreb [DLW], not surprisingly in LISP. This is a very good language to write an editor in, but I will say much more about that later. One reason it is so good is that the functions written for the editor can be used just like built-in LISP functions and so form a LISP-like editing language. Another implementation started somewhat after mine was written by Bernard Greenberg on Honeywell's Multics operating System also in LISP [BSG1] [BSG2]. This program places more emphasis than the LISP Machine version on creating an editing language from the functions used to produce the real-time responses to keystroke commands. In addition to the functions needed by all the commands, many other functions were added to make writing editor extensions in this extended LISP easier.

This basically describes the environment in which the ideas that lead to the development of this thesis were produced and developed. In addition to my rather general conviction that a display-oriented editor was best I had several constraints which had much effect on the eventual implementation. The machine on which the editor was to be written was an Interdata 7/32 running a segmented, but not paged, operating system. The address space was fairly large, but because the unit of swapping was the segment, the segments had to be small to minimize the memory load on the system.

The Goals for the Editor System

Quite a few goals exist for the system, some of which it has achieved and some it has not. Some of the goals are self-evident but many require elucidation. I will list the major goals here, and then proceed to explain them in depth in the following paragraphs.

1) Display-Orientation
2) Dynamic LISP-like Environment
3) Easy-to-Use Programming Language
4) General-Purpose Text Processing System
5) High Code Density
6) Space Efficiency
7) Reasonable Speed Efficiency

I have explained the need for a display-oriented editor above but here is a summary. Display-orientation is very good for graphically displaying text and changes being made to it. It provides excellent feedback for changes made to a text buffer. A display-oriented editor has built into it a display manager. This is a program which is called from time to time and whose job it is to make sure that the text on the screen reflects the contents of one or more internal text buffers. This is not a simple program, and one purpose for developing a general purpose system is to allow the display mechanism to be used easily by many different programs.

One of the biggest advantages of a program like Emacs is that almost everyone using it has a personalized version. The editor is aware of the specific properties of the terminal being used and adjusts its line length and page length and takes advantage of special capabilities the terminal may have. More interesting, though no more important, is its ability to change which program is called by a key. This frequently-used capability means that if I do not like the standard search routine I can have control-S (the Emacs search command) use my favorite flavor of search. Some people go to the extreme of changing around almost every character so that their version of the editor is almost unusable by anyone else. Not only can other built-in functions be re-bound to keys (the function that is invoked when a key is typed is called its binding), but entire packages of functions can be loaded in and bound to keys. The potential for specialization is almost unlimited.

A LISP-like environment provides all the necessary flexibility to enable run time modifications to the program environment. The primary requirement of the programming environment is the ability be able to hold a function as an object. This allows the rebinding of keys to be implemented as the assignment of a function into the appropriate cell, so that it will be called when the key is typed. Also needed is the ability to make a new function known to the system and to redefine an old function.

LISP is particularly well suited to handle these requirements and so makes a good model for the projected system.

The usability of a programming language is extremely hard to predict in advance of extensive use. The best one can do is to incorporate features of languages which are known to be easy to use. For this reason I used a LISP-like syntax and semantics for this project [MOON]. The issue of which functions to build into the system as primitives, though, is very difficult, and can only be decided by examining many programs and looking for common functions and sequences of functions.

Since one of the most common tasks on a computer is to deal with text in some way, it is reasonable to have a system that is specialized for processing text. Mail systems, editors, text justifiers and documentation readers are a few examples of important systems on any computer which deal mostly with text. Most of these also have some use for a display-oriented human interface. Thus there are many areas which can benefit from a convenient, easy-to-use system for processing text.

Since it is desirable to use this system for many programs, it should generate dense code, at least in as much as doing so is consistent with speed considerations. Systems like Emacs, which are written in TECO have the problem that TECO doesn't compile into anything; it is interpreted. This is mostly a result of the fact that TECO was originally an editor, not a programming language. Both the LISP Machine and Multics editor implementations mentioned above are in LISP and therefore are compiled as a matter of course. All of these other systems, though, are paged and therefore code size, especially pure code size, is of relatively little consequence. Our experience using MagicSix shows that memory of any sort is at a premium and so the density of compiled code is an important factor.

The other aspects of space efficiency are also important. Especially critical is the issue of how much impure area will be needed for each user in the system. On a time-sharing system it is vital to keep the per-user data bases as small as possible. The choice of language and the decision of what functions to make built-in have a primary influence on this.

The usual space-time tradeoff strikes here. The smaller the code that is to be run, the more processing it will take to prepare it for execution, and the more compact the data structures are the more decoding they will require for use. All you can do is to make the best set of compromises and to try to find clever strategies that will allow both space and time efficiency.

The System Implementation

The Sine system, as it is called, is made up of several levels of programs. The lowest of these is a pseudo-machine called the Sine Machine. This is an interpreter which interprets a binary object code and dispatches to a variety of internal routines. The compiler for Sine is written in LISP and produces the object code used by the Sine Machine. The language implemented by this compiler, in addition to compiling references to Sine Machine instructions, compiles calls to Sine functions and expands macros. All instructions, function calls, and macro invocations appear as nested, parenthesized lists, a la LISP. In addition, a great many, very useful programs are part of the editor, called TVmacs, so that they collectively represent almost another whole language.

The Pseudo-Machine

The decision to implement the lowest level of the Sine system as a pseudo-machine was not easy. At least two other possibilities were seriously considered. One was to follow the lead of Dan Weinreb and Bernie Greenberg and use LISP. This certainly would have been a lot easier since a fair sized portion of the effort required to implement this system was that of writing routines that are already in LISP. In fact a LISP did exist on our system but there was no compiler for it and it was very slow. If the editor was to be at all useful it had to be fast. The other possibility was to compile Sine code into Interdata 7/32 machine language. For most simple things, the compiler could generate the appropriate instructions; for more complicated functions it would generate calls to operators which would find their arguments on the stack or somewhere. The problem with this is that it would generate much more code than the proposal described below for most of the functions since it would have to manage the stack and other data spaces itself. The major advantage of this scheme is that it would make the instruction decoding very fast since it would be done by the compiler not at run time. If an appreciable amount of time is spent in the operators though, even this scheme would not help make it fast. Thus this plan was rejected on grounds of code density. This left the proposal to use a pseudo-machine.

This idea allowed a very nice modular approach to implementing the system. Each instruction can be viewed as a subroutine call with simple, shared argument-processing code. Those aspects of LISP which make it slow or which are not needed can be left out. The pseudo-machine is a LISP machine optimized for text processing work. In describing the Sine Machine I will assume a working knowledge of LISP since a great many aspects of Sine are like LISP. Ample documentation on LISP is available and I will not attempt to duplicate such work [MOON].

This decision to use a pseudo-machine as a vehicle for describing editing functions still left a great many questions, such as: How big are opcodes and operands? What kind of encoding should be present in each? How should the stack look? What are the basic data types and what do they look like? I will not go through the whole, long decision process, but in describing the Sine Machine I will comment when appropriate on the issues involved.

Most crucial to the performance of the pseudo-machine is the design of its architecture. Like almost any system that must support recursion and pure code, the Sine system must have a stack. The need for global static information also seemed very important, so some kind of impure area associated with each function whereby it could find global variables was added. The global variables, strings, buffer headers and other permanent information were put in a heap area called the string space. The addressing architecture should be able to address any of these areas as well as addressing the code so that PC-relative constants and branches are possible. In addition, immediate constants are important to save space and time.

Sine Machine Architecture

The decision of an instruction format is extremely important because it impacts very crucially on code density and decoding efficiency. Once the above requirements for the architecture are set, the remaining leeway is not too great. I decided in the interests of simplicity to use halfword (16 bits) opcodes and operands. Using only 8 bits for each was a possibility but that would have left little room for offsets or any kind of flag bits. Sixteen bits is a comfortable size. The opcode is only 12 bits long which leaves 4 bits for flags. Three of these bits are set so as to distinguish an opcode from almost all operands and the fourth specifies whether the result of the instruction should be pushed onto the stack. This allows the compiler to decide when a returned value should be saved on the stack and makes detection of wild branches quite easy by requiring all opcodes to have the correct bits set. If a branch is made to an operand the bits will be different.

```
declare          /* here is a declaration of the opcode as a
                    PL/1-style structure */
 1 opcode unaligned,
   2 designator bit(3),          /* must be "001"b */
   2 dont_push bit(1),           /* "1"b means don't push result */
   2 opcode_number bit(12);      /* index into opcode table */
```

The operand encoding is more complicated. The first (leftmost) bit is zero if the rest of the halfword is a signed immediate constant. If the first bit is a one then the next two bits specify an index register and the remaining 13 bits are a signed offset in halfwords for PC indexing and in fullwords for other index registers. The opcode indexes into a dispatch table which also contains information about how many operands should be fetched and an additional halfword of information about how each operand should be interpreted. The top two bits of this halfword indicate whether an address or a value is desired and whether the address, if any, is that of a variable. This additional bit is necessary because a variable is not simply a cell and sometimes the address of the value cell of the variable is wanted and sometimes the address of the variable itself is wanted; more about variables later. The remaining 14 bits form a bit mask for the allowed data types for this operand. Since each data object has type bits these can be checked against this mask when the value is fetched. This means that operations on illegal types are detected all the time and at very low cost.

```
declare /* operand bit layout */
 1 immediate_operand unaligned, /* immediate form */
   2 immediate bit(1),          /* must be "0"b */
   2 constant fixed binary(14),

 1 indexed_operand unaligned,   /* indexed form */
   2 immediate bit(1),          /* must be "1"b */
   2 index_register bit(2),     /* which index register 0-3 */
   2 offset fixed binary(12);

declare /* opcode table format */
 1 opcode_table (0:number_of_opcodes),
   2 operand_types (4) unaligned,       /* up to 4 operands */
     3 vbl_address bit(1),
        /* get the address of a variable */
     3 cell_address bit(1),
        /* get the address of a cell */
     3 allowed_types(0:13) bit(1),
        /* if allowed_types(13-data_type_number) = "1"b
                then type is legal for this operand. */
   2 instruction_type fixed binary(15),
        /* if this is 1 then this instruction has a return value
           otherwise it has nothing to return. */
   2 address_of_instruction bit(16);    /* where to branch */
```

The fact that four index registers exist but only the PC, the string space and the stack have been mentioned is not a mistake. In fact there are two index registers which point into the stack. The first of these points to the current stack frame. This is used to reference arguments and temporaries. The second is a pointer to the top of the stack, and is used to reference values left on the stack by previous instructions.

Stack Format

The format of a stack frame should now be explained. This format had to be worked out in concert with the design of the addressing format so that the important data areas could be easily and quickly addressed. As in most stack machines, much of the temporary storage is in the form of values pushed on the stack and calls (not instructions but subroutine calls) get their arguments from the top of the stack. In addition, this machine has the capability to address another kind of temporary, called automatic, whose extent is the duration of the invocation of a function. These are analogous to PL/1 automatic storage or LISP prog variables. The addresses of these must be known at compile-time and since the conventional top-of-stack pointer changes all the time, the concept of a stack frame had to be invented. The stack frame base pointer then can address arguments as negative offsets and automatics as positive offsets. Values that are pushed by instructions are then referenced by negative offsets off the top-of-stack pointer. The automatics are not the only thing in the stack frame though. The first things are the saved index registers. This constitutes most of the state of the processor and so it must be saved over a call, especially the PC, so the return instruction can find it. Next is the name of the function being called. This is taken from the name of the variable whose value cell contained the function object being called. This is why the call instruction needs the variable_address bit in its operand descriptor, but I will discuss functions and their naming later. Following the function name is a pointer to all the condition handlers active for this function. This points ahead into the stack frame where the active condition handlers are chained together, but I will talk more about these later too. Next is a halfword of flags which currently contain only the opcode of the call instruction so the return instruction can determine whether or not the returned value should be pushed onto the stack. Then follow three halfwords which keep count of the number of arguments to the function, the number of automatics in this frame, and the number of bound variables. The actual cells for the automatics follow, with the binding blocks for the variables bound in this frame after them.

The binding blocks and condition handlers deserve further mention. Though they are conceptually each on a separate stack, economics dictate that they both be incorporated into the one

Sine stack. This naturally leads to some conflict with the stuff that normally gets pushed onto the stack. Thus, the rule is that all binding blocks must be pushed before any other temporaries. They are of fixed size and the only way to determine their position is from the count of the number of automatics which precede them. The condition handlers are chained together so that their position on the stack is not important. As long as a condition handler is not created between the arguments to a function there should be no problem. In practice the use of handlers is done by macros and so any problems can be avoided.

The binding process is exactly like that of LISP. When a function is called, any global variables in the argument list are bound. The compiler generates code to do this and to assign the arguments to the variables that have been bound. In addition, a program can specifically ask to have a variable or an array cell bound. These requests must come at the beginning of the function to avoid the above-mentioned difficulty with binding blocks. The binding block contains three values: the object which contains the bound value (e.g. a global variable), the address of the value being bound (e.g. the address of the value cell in the global variable), and the saved value of that cell. This mechanism allows any fullword in the Sine environment to be bound, though in practice, mostly variables and an occasional array cell are.

The condition mechanism in Sine is useful for handling error conditions and for effecting non-local goto's; in this capacity the condition mechanism is more like LISP throws than real PL/1-style conditions. They do not cause a new activation but unwind the stack. The information contained in a condition handler is a pointer to the next handler, the address at which to start executing the handler, a contour pointer to tell how far to unwind the stack and the name of the error condition being handled. When a signal instruction is executed or a serious error occurs, such as the passing of an argument of illegal type to an instruction, the stack is traversed from the current frame to the base of the stack until a handler for that condition is found. If one is found, the stack is unwound to the specified point and control is passed to the saved location.

```
declare          /* stack frame format */
 1 stack_frame based (stack_frame_pointer),
   2 saved_pc_buffer pointer,
   2 saved_pc_offset_in_buffer fixed binary(31),
   2 saved_stack_frame_pointer pointer,
   2 saved_top_of_stack_pointer pointer,
   2 saved_variable_table_pointer pointer,
   2 name_of_function pointer,
   2 condition_handlers pointer,
   2 flag_bits bit(16), /* opcode of call instruction */
   2 number_of_arguments fixed binary(15),
    2 number_of_automatics fixed binary(15),
    2 number_of_binding_blocks fixed binary(15),
    2 automatics (number_of_automatics) pointer,
    2 binding_blocks (number_of_binding_blocks),
      3 bound_object pointer,
      3 bound_cell pointer,
      3 old_value pointer,

  1 condition_handler based,
    2 handling_pc_buffer pointer,
    2 handling_pc_offset fixed(31),
    2 contour pointer,   /* this is where the stack should be
                            unwound to */
    2 condition_name char(8);
```

String Space Data Types

The objects are what make this Sine and not LISP; a description of the denizens of the Sine world is next. The most important object in Sine is the buffer. This is an object that exists in most editors in some fashion or another but does not exist in LISP. In Sine, a buffer is a header and a description of a segment which contains text with possibly an embedded gap. Like ITS TECO and unlike the LISP Machine and Multics editors, the buffer in Sine contains a gap when it is being modified to alleviate the need to move all the text past the current point each time a character is inserted. The gap is positioned at the point of modification. Deletions simply require the expansion of the gap to include the deleted text. Insertions involve copying the new text into the gap and moving the bottom gap pointer past the new text. This makes insertion and deletion at a single point very cheap. Thus typing at the editor would simply involve copying the current character into the gap and incrementing the gap pointer by one. When the position changes, the gap does not need to be moved until an attempt is made to modify the buffer.

Due to the memory and address space problems on the 7/32, the text of each buffer is in a separate segment which may or may not be mapped into the address space. Thus part of the information in the buffer is about the buffer's state with respect to addressability. Whenever a buffer is addressed by an instruction, the Sine Machine calls a routine to make sure that the buffer is present in the address space. This applies not only to text but also to Sine code, which is also stored in buffers. This address space management has the advantage that Sine code can be present in very large quantities without causing performance problems since buffers containing code can be swapped in and out of the address space. There is no limit of two or three code segments for fear of running out of segments. The fact that buffers are not always addressable adds a bit of complication, since the program counter is no longer just an address, but also a buffer object. Also, functions like call, return, and even the condition handler must make sure the current code is swapped in.

The information associated with a buffer thus includes: address space swapping information, pointers to the gap and the top of the text, and the current position where inserts and deletes take place. In addition, associated with every buffer is a list of marks which hold places in that buffer. The position held by a mark floats with the text; it does not move when text is inserted or deleted before it in the buffer. Two of these marks are used to point to the first and last modified point in the buffer.

```
declare
 1 buffer based,
   2 next_buffer pointer,        /* not used */
   2 text_base pointer,          /* address of base of segment
                                    containing buffer text */
   2 swapping_info bit(32),      /* used for swapping between
                                    address spaces */
   2 location fixed binary(31), /* "point" */
   2 gap_start fixed binary(31),
   2 gap_end fixed binary(31),
   2 top_of_text fixed binary(31),
   2 beginning_of_modifications pointer,
   2 end_of_modifications pointer,
   2 mark_chain pointer,
   2 flags bit(32);              /* read_only, wired */
```

The marks are another data type that is built into the Sine Machine. The position held by a mark is recorded as an absolute position. When the gap is moved or when insertion or deletion occurs at a mark the mark is moved to point at the same text. Marks are all chained to the buffer to which they point so that when the gap is moved all the marks can be updated. One problem that has come up with marks is that a mark does not know which buffer it refers to. This happened because I did not want them to be three words long instead of two, but it has caused some problems and perhaps some scheme should be worked out. It has been suggested that the marks be on a circular chain which would both start at the buffer and end there. This would make finding the end of the chain more difficult but would not increase the size any of the marks any. The space-time tradeoff problem rears its ugly head again.

```
declare
 1 mark based,
   2 next_mark_in_chain pointer,
   2 offset_in_buffer fixed binary(31);
```

The next object is the string: a character string preceded by a halfword count. It is the primary mechanism for giving names to variables and for constant strings. Originally constant strings existed only in the code section and were referenced with a PC-indexed operand. When code was moved into buffers, however, these references, when passed to a function in another buffer, would become invalid if the calling code had been swapped out. The compiler was changed to cause the strings which were passed as arguments or saved to be copied into string space when the code section was loaded into a buffer.

```
declare
 1 string based,
   2 length fixed binary(15),
   2 text character(length);
```

A third sort of text object is a gnirt (string spelled backwards). This solves a problem with strings which is that they are not modifiable. A gnirt can have text appended to its end or deleted from its end; it is essentially a stack of characters. Thus it is mostly suitable for appending strings when a buffer is too expensive. The gnirt is a two word block, the first of which contains two halfword numbers, the maximum length and the current length. The second word is a pointer to the actual text. Both of these are allocated in string space. When an insertion would grow the length of a gnirt beyond the maximum length a new, larger text section is allocated and the old text copied in, then the insertion is completed.

```
declare
 1 gnirt based,
   2 maximum_length fixed binary(15),
   2 real_text_length fixed binary(15),
   2 text_area_ptr pointer,
 text_area character(maximum_length) based(text_area_ptr);
```

The fourth and last sort of text object is called a window. It is little used and mainly meant to pass around parts of buffers. A window has a pointer to a buffer and two marks, one to the beginning of the area and one to the end. Windows are of dubious worth and significant complexity. The use of the name window for these little used objects was extremely unfortunate. In common terminology a window is a portion of a terminal display screen which shows a buffer. There may be several of these on the terminal at once. Future uses of this word in the thesis will refer to a part of a terminal display not to this object.

```
declare
 1 window based,
   2 starting_point pointer,
   2 ending_point pointer,
   2 buffer pointer;
```

One of the very nice features of Sine is that insert is a generic operator. It will take any string object including a small integer which is interpreted as a single ascii character and insert it into either a buffer or a gnirt. In fact, since all character processing is done through special, fast coroutines, any instruction which accepts a string as an input argument will take any sort of text object. Therefore it is convenient to use the handiest or most efficient representation when working with text.

In support of the display orientation of the editor a data type exists, called a screen, which contains data needed to keep the image of a buffer up to date. Each screen corresponds to a window on the terminal display and is used to keep that window "correct". The display mechanism supports multiple windows on the display and each screen (and window) has a buffer associated with it. The screen contains information about how many lines long the window is, where it starts on the display, where to position the current line relative to the top of the window, and one mark for every line of the window. These marks point to the beginning of every visible line and are used to keep track of where each line starts and whether it should be redisplayed. The redisplay routine also makes use of the start and end modification marks that are maintained for each buffer. After each redisplay these marks are reset to show no modification.

```
declare
 1 screen based,
   2 next pointer,       /* not used */
   2 buffer pointer,
   2 line_on_display fixed binary(15),
        /* where the window starts on the display */
   2 number_of_lines_in_window fixed binary(15),
        /* this is really the number of lines times 4 minus 4 */
   2 first_line_containing_modifications fixed binary(15),
   2 displayed fixed binary(15),
        /* flag so this screen is only updated once */
   2 current_line fixed binary(15),
        /* line containing the current location */
   2 pad bit(16),       /* not used */
   2 force_display bit(32),
        /* one bit per line in screen: if the bit is set the
           line must be updated */
   2 empty bit(32),
        /* if the line is empty this bit is set */
   2 last_top fixed binary(31),
        /* address of first character in top line on screen */
   2 line_marks (24) pointer;
        /* only number_of_lines_in_screen of these
           are used but all 24 are allocated when the screen is
           created */
```

The editor maintains an array of pointers, one pointer for each line on the physical terminal display. Each pointer indicates the screen which includes that line. Thus if a screen contains the first five lines on the display then the first five pointers in this line array would point to that screen. If another screen contains lines three through ten, then the shared lines are part of both screens but must be in one window at a time. The line array determines which, by pointing to the screen "responsible" for each line. An instruction is available which sets the screen pointer for each line and adjusts the size and position of the screen.

This scheme has the difficulty that the point around which the window is centered is the current location in the buffer. This is not useful if two windows are to display different sections of the same buffer, since there is only one point (current position) in each buffer. For this and other reasons an alternate proposal has been suggested to increase both the flexibility and simplicity of the screen management system. This proposal is to upgrade the line array to a table containing all of the information that is needed. This would eliminate the need for screen objects altogether. The table would contain a structure for each line on the display. This structure would have a bit specifying whether it is the top line of a window or not. If it is, then it would have a pointer to a mark which would indicate the centering point for the screen, as well as specify some additional information needed to compute the new screen. Subsequent lines in a window would contain only a mark into the buffer where that line starts and pointers to the next and previous visible lines (table entries) of the window. This organization would reduce the number of marks necessary and would allow a screen to be discontinuous on the physical display.

Most of the rest of the data types are not specially oriented towards string processing but are needed to provide the desired LISP-like environment. The most important of these is the variable. This corresponds quite closely with the LISP atom. It i a global entity with a name, a value cell and a next_variable pointer. All the variables in the environment are chained together so that every module which is loaded into the environment will share variables of the same name. These variables are the primary means of communication between various modules. The "linel" variable lets every routine know what the line length is. The "current_buffer" variable allows any routine to modify the buffer being edited, without requiring every routine which deals with the buffer (almost all of the functions in the editor do) to take the current buffer as an argument which would be very expensive.

```
declare
 1 variable based,
   2 next_variable pointer,
   2 name pointer,
   2 value pointer;
```

To allow all functions to efficiently use global variables when their addresses are not known at compile time is obviously necessary. The names of the variables are compiled into the object segment of the module. There is only one instance of each variable per module even though there might be many routines contained in the module which use the variable. When the module is loaded into the Sine environment the variables are looked up in the environment to obtain their addresses and to create them if they did not previously exist. These addresses are then inserted into a table through which the references to variables are made. The address of this table is associated with every function; all functions that were compiled together share the same table. The pointer to this table is one of the four index registers used in operand address calculation mentioned above. Though the address calculations for variables are conceptually more complex than those which just reference stack data the difference amounts to only about three instructions in the decode routine.

The necessity of copying strings from the pure code buffers into string space, mentioned previously, is taken care of by including those strings which must be copied with the variables, with a flag set to indicate that no variable is to be created. The string is then copied into string space and a pointer to it is left in the cell in the variable table where variables normally are. The type bits on the pointers are used to differentiate the two when an operand referencing them is decoded.

A difficulty with this use of variables is that in order to have a piece of data be static it must also be global. This leads to naming and efficiency problems. Tne efficiency problems are caused by the fact that a global variable must have a name and be chained with the other variables so that other routines can access the variable. In essence all that is needed is a single cell. No name or other complication is needed at run time. All of this extra overhead for a global variable makes the string space, a per-user impure area, bigger than necessary. The variable table has a single header word which gives the count of the number of variables in the table. This is mostly used by the garbage collector. A second count could be added to specify the number of words in the table which are not global variables or strings but are value cells. These could be addressed as any of the cells on the stack are, and a single comparison should suffice to separate the local static variables from the global ones. The data I have found from metering the garbage collector has indicated that more that half of the storage in the string space is taken up with strings and my guess is that a fair fraction of those could be made into local variables, saving a lot of space.

Then, of course, there is the cons. This is the mainstay of most data objects in a LISP and while diminished somewhat in importance by the string data types in Sine it is still very important.

```
declare
 1 cons based,
   2 car pointer,
   2 cdr pointer;
```

The array is also an important data structure primitive, so Sine supports single dimension arrays of bits and pointers (objects). The header for the array contains a count of the number of elements and a indicator of the number of bits in each element. In principle, array elements of any bit length could be kept in such arrays but in practice byte arrays are better handled by strings and halfwords may be useful eventually, although I have not added the support for them as yet. Any other size is probably much more work than it could possibly be worth.

```
declare
 1 array based,
   2 element_size fixed binary(31), /* in bits */
   2 number_of_elements fixed binary(31),
   2 elements (number_of_elements) bit(element_size);
```

The last data type implemented by the Sine Machine is the function. This data type contains the information needed to call a function. This includes a buffer which contains the code, a pointer into that buffer where the function starts, and a context pointer to the variable table associated with this function. These three things are all set by the program which loads a code segment into the Sine environment when it encounters function definitions. The names associated with a function are not actually connected to the function object but come from the variable containing the function. The decision to name the variables and not the functions was not easy. This method has several advantages. There is only one sort of named object in the Sine environment, which certainly simplifies naming issues and, if nothing else, saves the space of having all those extra names around. But more interesting is the fact that variables can be set and bound and hence functions can be changed and redefined. This fact has some rather serious and perhaps dangerous implications but I decided that the extra flexibility and power provided by this capability would be of more benefit than harm - a statement not true of every scheme to increase flexibility and power.

```
declare
 1 function based,
   2 pc_buffer pointer,
   2 pc_offset fixed binary(31),
   2 variable_table_ptr pointer,

 1 variable_table based (variable_table_ptr),
   2 number_of_entries fixed binary(31),
   2 entries (number_of_entries) pointer;
```

The Garbage Collector

A very important aspect of the Sine Machine is the Garbage collector. As in most object oriented systems the environment gradually gets filled up with forgotten objects (called, for obvious reasons, garbage). A garbage collector is needed to peruse the string space and the stack and to recompact all objects in the space, thus reclaiming the space used by unwanted objects. The garbage collection scheme I chose uses a temporary segment into which all the referenced objects in the Sine environment are copied and the references to them updated. Then the new string space is copied back into the old one and the segment shrunk to the correct size. This subroutine is called whenever the allocator has allocated a certain amount of storage.

It does not have a fixed amount of memory to fill up as in most garbage collecting situations but instead it strives to keep the size of string space as small as possible without burdening the Sine user with overly many garbage collections.

Code Segment Format

The code segment format has been referred to briefly in other contexts. The first halfword of a code segment is the offset in the segment of the beginning of the strings which define the functions, strings, and variables. The first halfword of the variable section gives the count of the number of strings in this module. Then follow string descriptors, each of which contains two halfwords and a string. The second halfword gives the length of the string, while the first is the offset of the function definition if the string is the name of a function, zero if the string is a variable and minus one if it is just a string to be copied into string space. The loader interprets these halfwords and creates a string, it also creates a variable if the halfword is not negative and a function too if it is positive. Between the string area and the first halfword of the segment is the code consisting of any number of function definitions. The Sine code for each function is preceded by a halfword which specifies how many automatics are to be created by the call instruction when it builds the stack frame for that function. After the code for a function the strings that can safely be referenced by PC-indexed operands may appear.

```
declare
 1 code_segment based(function.pc_buffer -> buffer.text_base);
   2 offset_of_string_section fixed binary(15), /* in bytes */
   2 code (offset_of_string_section/2 - 1) bit(16),
        /* all the code and constants for the functions compiled
           in this module are here */
   2 number_of_strings fixed binary(15),
   2 string_descriptors (number_of_strings),
     3 string_type fixed binary(15),
     3 string_length fixed binary(15),
     3 text character(string_length);
        /* pad this out to nearest halfword */
```

The Sine Language

The set of Sine instructions is a little over one hundred strong. All of these can be used through the compiler, but just producing code for Sine instructions is not the major task of the Sine compiler. Its major task is to make instructions and user functions appear indistinguishable. It uses LISP-like syntax which is made easy since its written in LISP itself. It also manages variables and their binding, the definition of stack automatics, stack discipline and the expansion of macros.

The Workings of the Compiler

A form is a parenthesized list of tokens; the first of these is the operator and the rest are operands. All top level forms in a Sine programs must be one of three types. Either the operator is the token "variable" in which case the operands are declared to be global variables throughout this module, the operator is "defun" and the rest of the form is the definition of a function, or the token is "documentation" in which case the second operand in the form is a string which is stored in the object segment; the get_documentation function retrieves this string when given a function. Since declaring a function is useless unless the function is bound to a variable, the defun form also has the effect of declaring a global variable.

The form of a "defun" is as follows, where the items in square brackets are optional.

```
(defun <function> ([<arg1> <arg2> ...] [&aux <vbl1> <vbl2> ...])
        ... the body is like a prog body in lisp ...)
```

The second token in the form is the name of the function being defined. The next is a form itself which specifies the arguments and automatics for the function. The rest of the forms in the function definition are either single tokens in which case they are defined as labels or else they are compiled to yield code. In LISP terms, the functions are implicit progs so that labels, gotos and returns will work.

The idea for the "&aux" construction is taken from the Lisp Machine dialect of LISP [MOON]. The argument list has two parts, either or both of which may be missing, separated by the token "&aux". Those tokens appearing in the argument list before the "&aux" are argument names which are referenced as negative offsets from the stack frame pointer. If an argument has been declared as a global variable the compiler will generate a "bind" instruction to save the old value and a "store" instruction to assign the argument value which is below the stack frame pointer to the global variable. References to those arguments are to the global variable not the the actual argument.

Those tokens appearing after the "&aux" are auxiliary variables. If an "aux" variable is declared as a global variable then it will be bound like an argument, but no assignment will be made. Those "aux" variables can be modified in that function and the old value will be restored when the function returns. Tokens which are not globally known will have stack automatics generated for them which will be referenced as positive offsets from the stack frame pointer.

Once the argument list has been parsed the compiler proceeds to generate code for the rest of the function in a traditional two pass process. The first pass does most of the work. First, the locations of all of the labels must be determined. More interesting though is that the nesting of function calls is unwound on the first pass. That is, when a function as its argument another function, that other function must be evaluated first and its returned value pushed onto the stack. Then the reference to the argument of the first function becomes a stack reference. The depth of that reference below the top of the stack depends on how many arguments to that function were functions themselves.

Here is an example of an expansion of a nested set of function calls. The result is expressed in an assembly language type format. The index register "sp" is the top-of-stack pointer. "read_in_string" is a user defined function. The minus before some opcodes means that the returned value will not be pushed (the compiler sets a bit in the opcode). Aspects of this example will be explained in detail below.

```
        (set_loc (car buffer_list) 0)
loop    (ifnil (looking_atp (read_in_string) (car buffer_list) 0)
                loop)


        +car            buffer_list
        -set_loc        -4(sp),0
loop:   +call           read_in_string,0
        +car            buffer_list
        +looking_atp    -8(sp),-4(sp),0
        -ifnil          -4(sp),loop
```

The function which evaluates the forms also checks for macros. Before analyzing a form it makes two checks on the operator of the form. If it is an atom with a macro property then the compiler calls LISP eval on that form. The definition of the Sine macro puts the property on the atom and defines a function which recursively calls the first pass form evaluator to produce the effects that are wanted. The form evaluator does nothing else with the macro form assuming the macro function has done everything that is necessary. The form evaluator next checks to see if the operator is a known instruction opcode. If it is, the form is processed in the normal way: opcode followed by all the operands in the form. Otherwise the operator is assumed to be an external function. These should be declared to be global variables but the compiler will assume it is a global if it is not known. For a function, though, things are different. All arguments to a function must be pushed on the stack since functions always find their arguments just below their stack frame. Instructions can address their arguments where ever they are and so do not need their arguments push onto the stack for them. The actual code for the function is also different. Instead of an opcode followed by operands, a call instruction with one argument is generated. The argument to the call instruction is the number of arguments passed to the function.

The second pass then evaluates all symbols into an indexing type and an offset, decides whether to have the function, or instruction, push its result, and emits the appropriate code. After emitting code for all functions in the module the compiler sets the first halfword to the ending PC and then emits the string section described above.

The Language

Having to keep track of all the values that are at various offsets on the stack, offsets which change as things get pushed and popped, would make programming in Sine much too hard. The use of the LISP syntax for function calling eliminates the need to keep track of the stack at all. The programmer can almost forget that there is a stack. The compiler manages the stack for you; this is as it should be.

Another important simplifying factor is that there are only five kinds of control transfers not including the call and return instructions. The first of these is the unconditional branch. This takes an address in the current code buffer and transfers to that location. There is a branch on true and a branch on false instruction which take an address and a boolean and branch on whether that is true or false. The last two are a pair which branch on error or no error. These test a internal flag which is set by instructions like search and set_loc when they encounter a some error condition.

These simple transfer instructions are then used by Sine macros to produce the rather more complicated control structures used in LISP. A variety of predicates exist which return T or NIL and which can be passed on to the conditional branches. In this way "cond", "do_while", "do_until", "repeat" and others are implemented. Also the "iferror" function is really not a function but a macro which executes a form then generates a conditional branch which tests the error flag. This is used primarily to handle special cases involving failing searches of various sorts. The "errset" macro is used to set up a dynamic handler for a Sine condition. These conditions are produced either by the signal instruction or by serious errors in the Sine Machine such as an illegal type being passed to an instruction.

Since the macros are written in LISP and thus can deal with lists and lists of lists very easily it is simple to produce code to correspond to very complicated control structures with only very simple instructions. The usual sequence is to generate a conditional by consing up the appropriate list incorporating a predicate from the argument list and calling the form evaluator, then mapcaring that evaluator down a list of forms and generating a label at the end. This is what the "do_while" macro does.

A very interesting and useful Sine program would be a Sine Compiler. Since the data types are very LISP-like this should not be too difficult but Sine has not been blessed (or burdened) with a reader. This means that the parsing of the input text does not come for free as it does in LISP. Since Sine is a string processing language, though, it should be particularly suitable for this task. The major difficulty is that macros would have to be re-done significantly. They can not really be done without and since, in the current implementation of the compiler, they rely on the existence of the subr "eval" they must be dealt with differently. The major use of a Sine compiler written in Sine is that it would make writing simple little Sine programs much easier, since one would not have to switch to LISP all the time.

The Editor: What It's All About

Basic Commands

A real-time display-oriented editor has three basic categories of commands. All the rest of the commands can be divided into two or three other groups. The basic categories are: commands which modify the buffer, commands which manipulate the position of the cursor on the screen, and commands that deal with files. Each of these broad classifications can be further subdivided by their scope of action or various other means.

The first group of commands are those to modify the buffer. All the printing characters are in this group. They each insert themselves into the buffer. The rubout or delete key removes the previous character from the buffer. Control-D (^D) Deletes the next character in the buffer.

To understand how this really works it is necessary to understand the working environment of a display-oriented editor a bit better. What is seen by the user is a terminal screen that is initially blank save a line or two at the bottom describing the state of the editor. This blank workspace is very much like a sheet of paper on which you can type. Associated with this window is a point, a current position in the text buffer, which is represented by the terminal as a cursor. The exact nature of the cursor varies from terminal to terminal but is often a lighted square or horizontal underscore. This cursor selects the next character. The character to the left of the cursor is the last character. After typing a character the cursor is moved over the new character and it becomes the last character.

Rubout is good for removing characters that have been typed. If you notice a mistake several lines back, however, it would be a real shame to hit rubout a couple hundred times to fix the error. To help in these situations the control-P (^P) command moves you up to the beginning of the Previous line. Then ^F will move you forward one character at a time until you arrive at the offending spot. Control-D will delete the character under the cursor. Using ^Ds ^Fs and typing new characters the typo can be corrected.

This set of commands is enough to edit really anything but it would be very clumsy. To fill out the basic repertoire and to summarize here is a list of basic editing and positioning commands:

letters, numbers, and punctuation

  The character is inserted between the last character (to the left of the cursor) and the next character (under the cursor). The new character becomes the last character and the next character remains the same.

^F Forward over one character unless at end of buffer.

^B	Backward over one character unless at beginning of buffer.
^N	Goes to beginning of Next line (down one line).
^P	Goes to beginning of Previous line (up one line).
^E	Goes to the End of the current line.
^A	Goes to the beginning of the current line.
^D	Delete next character.
rubout	Remove last character.
^K	Kill from point to end of line. If line is blank delete the line. ^K at the beginning of the line will make the line blank. Two ^Ks will delete the line entirely.

The use of these commands can be made even more convenient when a command must be repeated many times. The ^U command reads a number and passes it to the next command as a repeat count. Thus ^U15^N will move you down 15 lines in the current buffer. If no number is supplied the argument is four times the old argument, where the default argument is one. Thus ^U^N is four lines, ^U7^N is seven lines ^U^U^N is 16 (1*4*4) lines, and ^U7^U^N is 28 lines. .

In order to make any of this editing worth while you must be able to save the buffer in a file. All of the file commands are sub-commands of the ^X command. ^X^W (type control-X immediately followed by control-W) will ask for a file name to write the buffer out to in a special area at the bottom of the screen called the echo area. Type the filename followed by a carriage return (CR) and the file will be saved. If no filename is typed (just ^X^WCR) then the buffer will be written out with the default filename. The ^X^S (Save) command just writes the buffer out to the current default filename without asking for a new one. When you start a session the buffer is initially empty and to edit an old file you need to read it in. The ^X^R command is analogous to ^X^W but instead replaces the contents of the buffer with the contents of the specified file.

With this set of commands you can edit quite reasonably. All the other fancy commands (and there are a lot of other commands) just make the editing easier and faster. This is also enough of an introduction to the editor to allow an explanation of how the editor is constructed in the Sine environment described above. At this point it is suggested that the reader glance over Appendix 1 enough to get an idea of what kinds of instructions exist and what sorts of arguments they take.

The Structure of the Editor

By structure I mean two things, what kind of data objects does it deal with and how are subroutines layered: who calls who. I will describe the structure of the editor outlined above. The editor naturally includes more complexity that would be needed to implement the above described subset.

When the editor is called the first time it must do a certain amount of work to set up the initial environment. Later entries return to the Sine environment with all information intact. The state of the buffer, the current position, the default filename, everything but the exact positioning of the cursor in the current window since the screen is refreshed when you reenter. The function that is called when the editor is called for the first time is called "top_level". Later invocations enter at the label specified by the restart_at instruction.

After initialization, "top_level" calls "invoke_editor". This function deals with setting up the mode information which will be described later. Eventually control enters the program called "reader". This is the function which reads characters from the keyboard and calls the appropriate function. It also does a number of bookkeeping actions which must be handled before or after every command. To save redisplay time the function calls "tyis" and only if no characters are in the input buffer does it do a redisplay. This means that while you are typing fast it will only attempt to refresh the screen every few characters. The "reader" also resets the argument to its default value of one and updates the buffer modified flag by calling "modifiedp" on the current buffer. The basic function of this routine is to read a character with "tyi", look up the function (in a variable) corresponding to that character in the dispatch array, and call that function with no arguments the number of times specified by the global variable "argument". The character read is put into the global variable "char" and the function called is put in to the global variable "function_to_call".

The selected function does whatever it wants then returns to the reader which reads another character and calls another function. Explaining the function of several typical functions will be useful. The most commonly called function is called "self_insert". This executes one instruction: "insert char current_buffer". The ^F command calls "forward_char" which is defined as follows:

```
(defun forward_char ()  ;no arguments to this function
       (add_to_loc current_buffer argument)    ;increment "point"
       (store 0 argument))  ;so it doesn't get called again
```

Note that the global variable "argument" is not an argument in the strictest sense of the word. Thus this function doesn't take any arguments but it does get information from its caller via "argument". This function doesn't want to get called repeatedly if you type ^U32431^F, instead, the function takes care of the requested repetition itself, then sets "argument" to zero so "reader" drops out of its loop. The method that is used to set the argument to other than one is quite simple. The function that build up the numeric argument is called when ^U is typed.

It reads characters until it gets a non-numeric character, then calls the appropriate function and resets "function_to_call" so the reader will repeat the correct function. The ^E function is a little more complicated:

```
;;; Set the location to just before the next CR or to the end
;;; of the buffer if there are no more CRs.

(defun end_of_line ()
       (set_loc current_buffer
                (iferror (sub (search current_buffer 13) 1)
                         (length current_buffer))))
```

Another level of routines are provided in the editor's environment which are not commands. They are various utility routines called from several of the command functions.

The data structures used by the editor are of three types: variables, arrays and lists. See Appendix 2 for a list of global variables used by the editor and a description of what each is used for. There are six object arrays used by the editor. One is the dispatch table for all the commands which can be accessed by one character. It is 128 entries long and is indexed by the ascii value of the character typed. Two other arrays are for the ^X sub-commands and the Meta sub-commands. The other three are copies which are used to restore the command characters to their defaults when reading in the echo area while in another mode; more about modes later. These arrays contain variables since functions by themselves are not named. They are loaded initially by using the "fill_vbl_array" instruction.

The lists used by the editor are mostly for holding information about tvbufs. The tvbuf management basically allows the editor to keep track of several editing workspaces at once. Only one is used for editing at a time but each has a separate file in it and has its own modes associated with it. Multiple tvbufs are very convenient for examining one file while editing another or for moving text from one to another or just for keeping any sort of information easily at hand. The global variable "buffer_list" contains a list of tvbufs used in the editor. Each time a new tvbuf is entered it is added to the list and so can be returned to very easily. Each element of the tvbuf list is as follows:

```
(tvbuf_name buffer screen mark filename modified_flag
      (mode_names . mode_defs))
```

The name of the tvbuf is identified by the first element; the buffer object actually used for inserting into is the second. The third is the screen object needed to display the buffer on the terminal. The mark is an alternate place in the buffer which some commands refer to. The fifth element is the default filename. The modified_flag is "t" if the buffer has been modified since last written out and "nil" otherwise. The last element is a cons containing two lists. This first is a list of mode names and the second is a list of functions which are to be called to activate this mode. When a tvbuf is entered the "goto_buffer" routine is called. This routine sets up several global variables such as "current_buffer" and "current_mode" which make it easier to use the information about this tvbuf.

Another important structure is the kill_ring. This is conceptually a circular ring of the last ten strings deleted from the buffer. A ^K, for instance, copies the line onto the kill_ring before deleting it. All of the commands that delete more than a single character copy the stuff they are going to delete onto the kill_ring. Consecutive deletions are concatenated, thus ten ^Ks will save the next five lines on the kill_ring. Several commands then exist for inserting the last thing killed and looking at previous killed strings. The kill_ring is the usual way of duplicating and moving text; by deleting the desired text and bringing it back.

The implementation of the kill_ring is rather complex. The text is stored in a special buffer (no screen, or filename, or anything - just a Sine buffer). Then a circular doubly threaded list contains marks into that buffer which separate the various elements of the kill_ring. Two flags called "save_delete_flag" and "old_save_delete_flag" are used to keep track of consecutive deletions. The new killed text is concatenated either to the beginning or the end of the old area so that the old order of text is preserved.

One of the very useful features of the editor as it is written is that the reader is recursively invokable. The reader function takes two arguments, one is a buffer to read the text into and the other is the screen associated with that buffer. The reader is defined with the arguments current_buffer and current_screen. These are the same variables that are set by the "goto_buffer" function and are used throughout the editor. When the reader is entered these variables are bound because they are global variables and the values passed to "reader" become the global values. Thus the read_line routine which reads text in the echo area can call the reader with "echo_buffer" and "echo_screen" and all references to current_buffer by functions like "self_insert" and "forward_char" will use "echo_buffer" automatically. To cause a carriage return to end input, as with the ^X^W and ^X^R commands, the slot in the dispatch table corresponding to CR is rebound, using the bind_array_cell instruction, to a function which returns to the caller. Then any text in the "echo_buffer" is the text that was typed in.

Another extremely important feature of the editor, due to the nature of the Sine environment, is that new functions can be loaded at any time. This means that all the functions that may be needed while editing do not have to be present initially. Users can load their own functions to replace standard ones or add new functions or packages of functions of their own to the environment. When the editor is first started it looks for a file of initial commands to run which the user can specify. This allows each person to specially tailor his environment to his preference. Also some functions in the dispatch table are not functions at all but are strings. If the reader tries to call a string, that string is interpreted as a filename containing the function that is to be called. This auto-loading feature is very useful in loading large, infrequently used packages into the editor when and if they are needed.

Modes

Two issues have not been addressed very well so far. The first is that of command character allocation and the other is that of specialization. Both are extremely important in the construction of a large editing system. The decision of what keys should execute which commands might seem unimportant but human nature being what it is the debate on the subject often blooms into raging controversy. There are about three problems fueling the argument. The first is that there are not enough keys to go around. There are only 32 control characters and some of those such as ^M (CR) and ^I (tab) are taken. This is not nearly enough characters especially if some attempt is made to make the commands mnemonic (consider even ^A: beginning of line?). The use of the escape key and the ^X key as prefix commands helps but the double characters are harder to type. Another problem is that with so many commands, most bound to non-mnemonic keys, and different implementations with different command sets there is no standard set of editing command keys. The really basic reason is that different applications require different actions when dealing with the text. The elementary commands described above do not really illustrate this problem but many commands exist to specialize the editor for the editing task at hand. A good example of this are commands for commenting code. Different languages have comments that work differently. But to have a separate command for each type of commenting style is a ridiculous waste of commands. The objections to all these arguments is that people do not like to have editing commands change. Most editing functions become habitual sequences of key strokes and it is very hard to remember to use a different set of commands.

Though the command allocation deals in part with specialization, the question is more complex than that of deciding which kind of commenting to do. Specialized applications involve changing the definition of a word, removing or adding a function or class of functions, or even changing something as fundamental as the self_insert command. A general mechanism for changing everything and then changing it back is needed. The modes facility I implemented in the editor is a good attempt at this.

Attached to each tvbuf is a list of modes that are in effect when that is the current tvbuf. There is usually a major mode which is a function of the sort of text you are editing and a list of minor modes that can modify any (almost any) major mode. Examples of major modes are PL/1_mode, TECO_mode, LISP_mode, and text_mode. Minor modes can be such things as Auto_Save_mode (which writes your tvbuf out every so often so that if the system crashes little work will be lost) and Auto_Fill_mode (when the line you are typing gets too long it will automatically insert a CR in the right place).

The mechanism that I used to accomplish this is to attach to each tvbuf a list of mode names (so a person can find out what modes he is in) and a list of functions which implement those modes. Whenever a tvbuf is changed or a mode is added to a tvbuf a flag is set and the reader is told to return to its caller. When the caller, "invoke_editor", notices this flag set it immediately calls the reader again. But "invoke_editor" does not call the reader directly, but through a chain of mode functions. It sets a global variable, "tunnel_path", to the list of mode functions to call then calls "keep_on_tunneling". This function takes the first function on the list and saves the rest of the list in "tunnel_path", then calls that first function. Each mode function then calls "keep_on_tunneling" when it is done setting everything up. When the list of functions is empty the reader is called.

The advantages of this scheme are that the binding and unbinding mechanism can be used to effect the changes required by each mode. Variables can be changes by making them &aux variables in the mode function. Functions can be changed similarly. Key bindings, the most common change, can be done by the "bind_array_cell" instruction. All of these actions are automatically undone upon return, planned or unplanned. If an error occurs and the handler unwinds past all of these levels of mode functions all their changes will be undone.

Major and minor modes are implemented in exactly the same way, which casts suspicion on the distinction made by ITS Emacs and others. The idea itself are very good. The power of an editor system can not be effectively used if the user must remember and explicitly type a thousand different commands. Despite or because of its promise this area of editor design is very poorly understood. Consequently, this aspect of editor design will probably see the most work in the future.

Results, Observations and Conclusions

In this last section I will review the interesting ideas and problems that have come out of this project. In addition to these thoughts I will mention a few ideas fcr future applications of this system or a system of similar design.

A problem I had not bargained with when I first designed this system was the amount of impure overhead that would be necessary to support all these data types and complicated data structures. On the small machine this system was implemented on the size of this area is a real problem. The biggest contributor to this problem is the names of all the variables and strings of other sorts. These are theoretically constant strings but because of problems with limited address space size I had to copy the names and strings into the string space. In a system where the address space swapping is not necessary the strings can be referenced where they are without copying. This could help a lot.

Considering the interpreted nature of the Sine Machine the speed of even quite complex programs is surprisingly good. This is partly due to the fact that the whole interpreter and most of the instructions are written in assembly language. The operand decoding routine is also very carefully optimized for speed. I would guess that obtaining additional speed would require using the scheme outlined above for compiling directly into machine code. If speed is not of absolute importance, and in editing it rarely is, then this pseudo-machine seems to be a good speed/space compromise.

The issue of personalization versus compatibility is very interesting. Editors of this kind allow the user to completely redefine how the editor looks. This means that trying to arrange a compatible set of commands is almost impossible if next to no one has the same command environment as anyone else, let alone each installation with their own version of an editor. But perhaps indicates that it is not worth the trouble. The cost of everyone having their own different environment is little more expensive than having everyone's the same, so perhaps it is not worth worrying about a standard.

Several times during the design of the system naming issues came up. Most of the time I used LISP as a guide in solving the problems but in the case of functions I diverged rather radically. A function is one of several facets of the named object in LISP (the atom). In my scheme, the only object attached to a name was a value. This value could be a list, a number, a function or anything. The question had to do with what to do with functions without names. Can it be called? Can it be referenced? Several places in the editor I assign a function from one variable to another, which serves to rename the function. The fact that functions could change names in midstream was a little disconcerting. But no serious problems have come of it, though a great deal of code has not been written in Sine to date.

The most interesting and fundamental problem that this project attempted to address is the tradeoff between flexibility and complexity. The question is: How does one provide as much flexibility as possible to the sophisticated user without removing all flexibility from the realm of the relatively naive user? ITS TECO is so complex and subtle that almost no one can use it effectively. Though it is very powerful, its complexity effectively prevents its widespread use. My goal in this respect was to have a system easy enough to use that the inexperienced user could take good advantage of it.

Basically almost any task that involves either human interface or text processing is a possible application for a system like this. A command processor is one program I have always though would be useful. A documentation system to peruse and edit documentation would be very handy. A text justifier is an often-mentioned project but that task is a bit too processor-intensive a task to be adequately handled in Sine, though as an interactive text justifier it is already useful. My editor is missing a macro mode. This is some way to be able to quickly type in a simple Sine program to do something that would require several keystrokes, then to quickly compile or interpret this and perhaps attach it to a key. This would greatly help in debugging Sine programs and in all sorts of editing tasks.

The Sine system is written in a very modular fashion. The pseudo-machine that it is written in can be conceived of as a package of utility subroutines which provide perhaps the ultimate in modular design. Every operation is performed through this very carefully defined, relatively simple interface. The Sine system would probably be ideal for transporting to other sites with different computers. The reimplementation of the Sine Machine on the new system immediately permits the whole editor to run. The system is flexible and powerful enough so that there is still a great deal of room for growth even with this well defined interface.

References

[ECC] Ciccarelli, Eugene. An Introduction to the Emacs Editor. AI Memo #447, January 1978, MIT.

[BSG1] Greenberg, Bernard S. Real-Time Editing on Multics. Multics Technical Bulletin # 373, Honeywell Information Systems, Inc. April 1978, Cambridge.

[BSG2] Greenberg, Bernard S. Online Multics Emacs Documentation. (>udd>m>emacs>editor.info) MIT-Multics, Cambridge, Mass.

[M6] MagicSix Subroutine Manual. MIT Architecture Machine Group. 1978.

[MLK] Michael Kazar, Dynamic Linking in a Small Address Space, Undergraduate Thesis, EECS Department MIT, May 1978

[RMS1] Richard Stallman, ITS Teco Order, Online Documentation. (AI:.TECO.;TECORD >). MIT Artificial Intelligence Laboratory.

[RMS2] Richard Stallman, ITS Emacs Order, Online Documentation. (AI:INFO;EMACS >). MIT Artificial Intelligence Laboratory.

[MOON] Dan Weinreb and David Moon. Lisp Machine Manual. November 1978, MIT Artificial Intelligence Laboratory.

[DLW] Dan Weinreb, A Real-Time Display-Oriented Editor for the LISP Machine, Undergraduate Thesis, EECS Department MIT, Jan. 1979

Appendix 1 - Table of Data Types Used by the Sine Machine

This table defines the opcodes and the data types that each takes as well as a definition of what each instruction does. The names, type numbers and a short description of each type is given below. See the text for more information about each of these data types. In the functional definition op1 will refer to the first operand, op2 the second etc. No instruction takes more that four operands except fill_vbl_array.

Numbers:	(0)	28 bit signed two's complement integers
Booleans:	(0)	t ("0b02fff1"b4) or nil ("0b02fff0"b4) are just numbers.
Labels:	(0)	positions in a program. These are 24 bit number from the base of the code buffer segment.
Characters:	(0)	A single character represented by its ascii value.
Buffers:	(1)	Large scale text objects. Can search and insert and delete text anywhere.
Strings:	(2)	Unmodifiable text objects.

Windows:	(3)	Used to delimit part of a buffer.
Conses:	(4)	LISP-like conses with car's and cdr's
Variables:	(5)	Global variables with names and value cells
Functions:	(6)	Procedure which can be invoked with arguments.
Marks:	(7)	Pointer at text in a buffer which float with the text.
Screens:	(8)	Description of display windows needed to update the display of a buffer.
Arrays:	(9)	One dimensional arrays of objects or of bits
Gnirts:	(10)	Strings object. Can insert and delete only at the end.

Compound Objects

Text:	A character, buffer, string, window or gnirt
Growable:	A buffer or gnirt.
long_text:	buffer, string, window, gnirt (text minus

	character).
regional:	buffer, string, gnirt, character. (text minus window)
char_map:	buffer, string, window, gnirt, character, array. (text plus array)
logical:	number and boolean
Anything:	Any of the above data types.

Notes

Some operands are fetched as addresses only. The store instruction is prime example of this. There are two methods of getting the address of something if it is a variable. Either the address of the variable's value cell or the address of the variable itself may be needed. (VA) will indicate Variable Address and (CA) will indicate Cell Address. If neither are present then the object itself is passed to the instruction.

A (NR) after the opcode name will indicate that the instruction does Not Return a value which may be pushed onto the stack. A bit in the opcode can cause a returned value to be discarded instead of pushed onto the stack.

Storage Control:

store anything anything(CA)

return (op1)

The value op1 is stored into op2. These arguments are the reverse of the order of arguments to the LISP function setq.

Numeric operators:

add number number

return (op1+op2)

sub number number

return (op1-op2)

mul number number

return (op1*op2)

div number number

return (op1/op2)

this is integer division so the result will be truncated to the nearest integer.

mod number number

return (remainder of op1 divided by op2)

min	number	number

return (the smallest of op1 and op2)

max	number	number

return (the largest of op1 and op2)

Logical operators:

and	logical	logical

return (bitwise: op1 & op2)

or	logical	logical

return (bitwise: op1 ¦ op2)

xor	logical	logical

return (bitwise eXclusive OR of op1 and op2)

List Operations

For more information about list structures see any introductory guide to LISP. These functions exactly parallel standard LISP functions of the same name.

cons	anything	anything

```
        return (a cons whose car is op1 and whose cdr is op2)


car                    cons
        return (the car of op1)


cdr                    cons
        return (the cdr of op1)


caar                   cons
cadr                   cons
cdar                   cons
cddr                   cons
```

For your convenience caar, cadr, cdar, and cddr are defined to take the car of the car, the car of the cdr, the cdr of the car, and the cdr of the cdr respectively.

```
rplaca                 cons              anything
        return (op1)
        This function replaces the car of op1 with op2.


rplacd                 cons              anything
        return (op1)
        This function replaces the cdr of op1 with op2.
```

Variable operators:

intern text array

return (new variable with a name of op1)

This instruction makes a new variable with a specified name and hashes it into op2 which is an obarray. If a variable of the specified name already exists on the obarray that variable is return, a new one is not created and the error flag is set. These variables will never conflict with other variables in the Sine environment of the same name.

make_variable text

return (a variable with name op1)

This looks in the Sine environment for a variable of name op1 and returns it if it can. Otherwise it creates a new one and puts it into the Sine environment. If an old variable is returned the error flag is set.

get_pname variable

return (the string which is the name of op1)

Stack manipulation operators:

These are all completely random instructions which modify the stack pointer and are of no interest to the casual user with the possible exception of push which just prior to a return instruction returns the argument to the caller.

push anything

return (op1)

This is just so that objects in variables can be put on the stack. This is used by the compiler when preparing do a call to a function since all args to a function must be on the stack.

pop (NR) anything(CA)

pops the top of the stack and stores it into op1.

squish (NR) number

This throws away op1 stack locations from just below the top item on the stack. The top item on the stack is still top but is op1 cells closer to the base.

discard (NR) number

throws away the top op1 things on the stack.

bind (NR) variable(VA)

bind takes a variable (like call does) and saves its value in a binding stack. All explicit bindings (the assembler does some) must occur before anything gets pushed on the stack in the function.

```
bind_vbl (NR)          variable
        the same as bind but it takes a variable like call_vbl
        does.

bind_array_cell (NR)  array            number          anything
        binds a specific element of an array to a value.
```

Control operators:

```
call                   variable(VA)   number
        return (the top value on the stack when this function is
                returned from)
        This calls the function bound to the variable op1 with
        the top op2 values on the stack as arguments.  This
        instruction is normally generated from implicit context
        by the assembler and can be forgotten about.  If the
        value cell of the variable does not contain a function
        the error "call^fun" will be signaled.

call_vbl               variable       number
        This is the same as call but fetches its argument
        differently.  An example of the difference between the
        two is as follows:
        (call foo 0)
                This is generated by (foo) in a Sine program.
        (call_vbl (ar function_dispatch 12) 0)
```

This code will expect to find a variable in element 12 of the array "function_dispatch" and will call it with no arguments.

return

Returns from a function. This instruction is automatically generated by the assembler when it reaches the end of a defun.

restart_at (NR)   number

This is a instruction which gives the interpreter a label to branch to if the interpreter is reentered. This allows the editor to recover gracefully if an error occurs and the user is dumped out to command level. He can release and reenter the editor, which can then take appropriate recovery action.

eval   variable

return (op1)

The normal argument fetching for op1 is performed. This returns the value cell of a variable. Compare quote below.

quote   variable(VA)

return (op1)

This returns a variable object not the value cell.

```
goto (NR)               label

        branches to the label specified


ift (NR)                anything       label

        branches to the label if the boolean is not nil.


ifnil (NR)              anything       label

        branches to the label if the boolean is nil


berr (NR)               label

        Branches to the label if the error flag is set and resets
        the error flag.


bnoerr (NR)             label

        Branches to the label if the error flag is not set.


handle (NR)             text      label

        Causes a branch to the label if the condition with the
        name specified by op1 is signalled.


signal (NR)             text

        Signals the condition named by op1.


revert (NR)

        Reverts the most recent handler.
```

Predicate operators:

```
t

        return (t)

        This is the number "0b02fff1"b4.


nil

        return (nil)

        This is the number "0b02fff0"b4.


eq                         anything        anything

        return (t or nil)

        If op1 and op2 are text objects then eq returns t if the
        strings are equal, otherwise it returns nil.  If the
        objects are not text they are checked to see if op1 and
        op2 are the same object.


not                        anything

        return (if op1=nil then t else nil)

        This instruction returns nil if it is passed any object
        except nil.  It returns t if passed nil.


gp                         anything        anything

        return (if op1>op2 then t else nil)
```

```
gep                   anything       anything
        return (if op1>=op2 then t else nil)

lp                    anything       anything
        return (if op1<op2 then t else nil)

lep                   anything       anything
        return (if op1<=op2 then t else nil)

looking_atp           text           buffer         number
        return (if op1=<text in buffer, op3 chars from location
                of op2> then t else nil)

functionp             anything
        return (if op1 is of type function then t else nil)

stringp               anything
        return (if op1 is of type string then t else nil)

variablep             anything
        return if op1 is of type variable then t else nil)
```

Search operations:

All these search operations set the error flag if the search fails. The value returned by these instructions in the case of a failure is not clearly defined so the iferror macro should always be used if there is a possibility of failure. It is used as follows:

```
        (iferror <form in which an error may occur>
                 <form to be executed if an error occurs>)
```

To go to the next line or the end of the buffer if on the last line the following code could be used:

```
        (set_loc buffer (iferror (search buffer 13)
                                 (length buffer)))
```

```
search                  buffer          text
        return (A number which is the position in the buffer of
                the end of first occurrence of the string after
                the current location)

rsearch                 buffer          text
        return (A number which is the position in the buffer of
                the start of the first occurrence of the string
                before the current location)

searchr                 buffer          text
        return (A number which is the distance forward from the
                current location in the buffer at which the first
```

occurrence of the string after the current location ends)

```
rsearchr                 buffer          text
          return (A number which is the distance backward from the
                  current location in the buffer at which the first
                  occurrence of the string before the current
                  location)
```

The next set of functions search for a list of characters specified by op2. They all return the position just before the character found. The last two characters of the instruction name specify the direction of search and the point relative to which the resulting position is returned. The first character is "f" for forward or "b" for backward from the current position. The second characters is "a" if the absolute position is returned and "r" if the number of characters from the current position is returned. The suffix "_br" will always return a negative number.

The "find_first_in" sequence is equivalant to the PL/1 search builtin; it finds the first character in the buffer which is specified in op2. The "find_first_not_in" is analogous to the PL/1 verify builtin; it finds the first character in the buffer which is not specified by op2.

In all of these, op2 can be a bit array; if the element of the array addressed by the ascii value of a character is a one then that character is specified by the bit array. Op2 can also be a text object in which case all the characters in the text string are specified. The bit array form is more efficient and should be used for repeated calls to these routines. A bit array can be easily fill in from a text string by the fill_char_array instruction. See below for its usage.

```
find_first_in_fa      buffer        char_map
find_first_in_fr      buffer        char_map
find_first_in_ba      buffer        char_map
find_first_in_br      buffer        char_map

find_first_not_in_fa  buffer        char_map
find_first_not_in_fr  buffer        char_map
find_first_not_in_ba  buffer        char_map
find_first_not_in_br  buffer        char_map
```

Buffer and string operators:

```
make_buffer
        return (new empty buffer)

make_string           text
        return (string whose value is the same as op1)
        This is essentially a copying operation.
```

```
make_window             buffer
        return (window on to the buffer op1)
        The region specified by the window is specified by the
        define_window instruction; see below.

define_window           window          number          number
        return (op1)
        The low bound of the window is set to op2 in the
        buffer addressed by the window and the high bound is set
        to op3.  If either op2 or op3 is illegal the error flag
        is set.

make_gnirt
        return (new empty gnirt)

insert                  text            insertable
        return (op2)
        This inserts op1 into op2 at the current position if op2
        is a buffer and at the end if op2 is a gnirt.  If any
        of these insert operations are attempted on a read-only
        buffer the error "bash ROB" (read only buffer) will be
        signaled

insert_region       regional    number      number      insertable
        return (op4)
```

This inserts the text of op1 from position op2 to position op3 into op4.

```
insert_ioa                 long_text      number         insertable
```

return (op3)
calls a MagicSix system routine, called ioa$ioars_nnl, [M6] with op1 as the control string and op2 as an argument and inserts the string returned from the formatting routine into op3. This system subroutine is primarily used to convert numbers to character strings but it also provides very good control over the exact nature of the conversion.

```
nth                        long_text      number
```

return (the character object at the location op2 in op1)
The position is number of a character is defined as the number of characters between the beginning of the text string and the addressed character. Thus (nth "foo" 0) returns the ascii value of "f". If op2 is negative nth returns the first character and sets the error flag. If op2 is greater the the length of op1 then the error flag is set and nth returns a -1.

```
nthr                       buffer         number
```

return (character in op1 op2 characters from the current position of op1)

This is equivalant to:

```
(defun nthr (buffer offset)
        (nth buffer (add (location buffer) offset)))
```

location   buffer

return (current position location of op1)

set_loc   buffer   number

return (op2)

This makes the current location of op1 be op2. If the position goes negative it is set to zero and the error flag is set. If the new position is greater that the length of op1 it is set to the length and the error flag is set.

add_to_loc   buffer   number

return (op2 + (location op1))

This increments the current position of op1. The same error actions as in set_loc take place.

modifiedp   buffer

```
return (if <buffer modified since last redisplayed>
        then t else nil)
```

This instruction examines the marks which point to the first and last points modified in the buffer if the first point is less than or equal to the last point then it returns t else it returns nil. These marks are used by the redisplay routine to identify which parts of the buffer need to be refreshed on the screen. They are reset to the unmodified position after every redisplay operation on a screen containing the buffer.

get_hpos   text   number

return (the horizontal position of character at position op2 in op1)

It does this by searching backwards through op1 from position op2 for a carriage return or the beginning of the text, then calculating the horizontal position forward from there. Very long lines will not wrap around but return hpos's as though linel is infinite.

length   text

return (number of characters in op1)

delete   insertable   number

return (op1)

deletes op2 characters from op2. If op1 is a gnirt the last abs(op2) characters are deleted. If it is a buffer a positive number deletes characters after the current location and a negative number before. (delete gnirt (length gnirt)) empties a gnirt. If op2 is larger than the amount of text that can be deleted in the specified direction then the error flag is set and as much is deleted as possible. If op1 is a read_only_buffer then this instruction will signal the error "bash ROB".

```
make_read_only          buffer
```

return (op1)

This sets a bit in the buffer which causes the error "bash ROB" to be signaled whenever an attempt is made to modify it.

Mark operators:

```
make_mark
```

return (new mark attached to no particular buffer)

```
set_mark                mark          buffer         number
```

return (op1)

The mark is added to the mark_chain for the buffer op2 and the mark is set to position op3 in the buffer. If the position is illegal for that buffer the error flag is set and the mark is set to the end of the buffer.

```
unset_mark              mark          buffer
```

return (op1)

The mark is removed from the mark_chain for the buffer op2. If the mark was not on the mark_chain the error "unk mark" is signaled.

eval_mark	mark	buffer	

return (location in op2 specified by op1)

Array operators:

make_array	number	number	

return (new array of the specified type and size)
An array is created with op1 bits per entry and op2 entrys. A request for an array with other than 1 or 32 bits per entry will cause the error "array sz" to be signaled. All elements of the array are zeroed.

ar	array	number	

return (op1[op2])

as	anything	array	number

return (op1)
The op3-th element of op2 is set to op1.

fill_vbl_array	array	number	

return (op1)
The instruction really takes an arbitrary number of arguments and therefore causes the compiler no end of

trouble. op2 is the number of additional arguments, which must be variables or strings, which are stored into consecutive elements of op1. The variables are stored into the array not the value cells of the variables. Each operation is equivalant to:

(as (quote vbl-n) op1 n)

```
fill_char_array        array            text
```

return (op1)

If the array is not a bit array then the error "arry^bit" will be signaled and if the array is not 128 bits long then "^chrarry" will be signaled. Each character in op2 is used as an index into op1 and that element is set to one.

I/O operators:

tyi

return (ascii value of the next character typed on the terminal)

tyis

return (the number of unread characters in the input buffer)

```
tyo                    character
```

return (op1)

This prints op1 on the terminal at the current position. In general, it should not be used, use print instead.

```
print                text          number        number        number
```

```
return (if <more was flushed>
                then <ascii value of flushing character>
                else 0)
```

This prints op1 at the (op2,op3) position on the screen. If op4 is odd more processing will be performed. The redisplay subroutine is informed of the damage produced by the print instruction to the screen and next time the display instruction is called the text over written by print will be restored. Note that tyo does not inform the redisplay code what areas of the screen it is overwriting.

```
print_clearing       text          number        number        number
```

Same as print except that a clear to end of line is done before every line is output. This instruction should go away. To replace it the next bit of op4 should specify whether clearing is to be done.

```
read_file                  buffer          text
```

return (system error code for last file system operation performed)

Reads the file specified by op2 into the buffer. All marks associated with the buffer are zeroed. If the operation fails the returned value will be negative and the error flag will be set.

write_file text text

return (last system error code recieved)

Writes the text specified by op1 into the file named by op2. If the returned value is negative the error flag is set.

load text

return (last system error code recieved)

Loads all of the functions in the file specified by op1 into the current Sine environment.

Miscellaneous MagicSix operators:

command_args number

return (op1-th argument to the Sine Machine Interpreter)

This allows the program which calls the interpreter to pass infomation into the Sine program.

cline (NR) long_text

Calls the command processor on op1.

call_af text

return (a gnirt containing the text returned by op1 called as an active function)

get_documentation variable

return (the string specifying documentation for this function)

When a Sine module is compiled it is possible to specify a string describing the documentation for the module. Usually this is the filename of a file containing the documentation.

Display operators:

make_screen buffer

return (new screen good for displaying op1)

force_display screen number

return (this returns garbage and shouldn't return anything)

If op2 is positive ~~then op2~~ is the line on the screen where "point" should be positioned. If it is negative the screen will be recentered around "point". In either case the specified screen will be refreshed.

display_screen (NR) screen number number

This instruction causes op1 to be displayed at physical line op2 and extending op3 lines.

display (NR)

Redisplays the console display, this requires it to step through each line in the console display and determine what screens are currently visible. It will reset the default display mode of each visible screen and the modified marks of each buffer.

Appendix 2 - Global Variables Used in TVmacs

C-X_dispatch

The dispatch array for the control_X commands (Control-X prefix).

M_dispatch

The dispatch array for the Meta commands. (altmode prefix)

abort_flag

This value is used to pass information from various commands to the reader loop. It has several possible values.

0 - Usual thing: read another character

1 - The user wants to abort: unwind

2 - Just return from the reader

3 - Return from the reader to change modes

There are several occurrences which could cause the reader to do something besides reading more characters. If anything unusual happens and a command wants to abort, the command signals "abort" and the reader catches this with an errset and goes on reading commands. If the abort function is invoked, it means that the user has gotten to a place he doesn't want to be, so return. This command is normally bound to ^G and is used primarily for getting out of the read_line routine for inputting a line. This function sets "abort_flag" to 1 and returns. The reader notices this and signals "abort" so that the reader calling it will notice an error. The reader must signal "abort" because if the "abort" function tried to do it the first reader would handle it. The reader must signal "abort" outside its errset so that control will unwind to the previous level. The "top_level" function has a handler for "abort" and just calls the reader again if it is ever signaled. This popping-up scheme is complicated but really works. The value 2 is used to exit the reader in the normal way; it is used to quit the editor. The 3 indicates that there has been some change in the modes for the current buffer and that "invoke_editor" should call the reader again with the new modes in effect.

alphanumerics

A bit array with bits set for all letters and numbers for use with the find_first_... instructions.

argument

The value of the command argument. It is set by the functions "get_multiplier" and "get_number" which are called by ^U.

argumentp

T or NIL depending on whether an argument was given to the command. It is used to distinguish between the user giving an argument of 1 and the user not giving an argument which sets "argument" to 1 by default.

buffer_list

The list containing all buffers known about by TVmacs. See also current_tvbuf.

char

The character last typed and read by the reader. It is primarily used by "self_insert" so that it can insert the correct character.

clear_modified

This is set by the functions which read files into the buffer so that the reader will clear the modified flag for the current buffer. It can not clear it itself because the action of reading in the file will make the buffer modified. It will remain modified until the reader calls the display function.

current_buffer

The buffer which contains the text of the current tvbuf. The "goto_buffer" function sets it along with the next several variables.

current_filename

The default filename of the current tvbuf.

current_mark

The mark associated with the current tvbuf.

current_mode

The cons containing the list of mode names and functions for the current tvbuf.

current_modifiedp

The modified bit for the current tvbuf.

current_screen

The screen for the current tvbuf.

current_tvbuf

The list containing all of the above state variables for the tvbuf. The "buffer_list" is a list of such lists.

default_cxd

The default C-X_dispatch table. See also C-X_dispatch and default_d.

default_d

The default dispatch table which is copied into the "dispatch" array by the "invoke_editor" function to allow each invocation of the editor to start with the default bindings, not the ones of the modes it was in when it was last called. It is also the dispatch table used by the read_line function. See also default_cxd and default_md.

default_md

The default dispatch array for the meta commands. See also default_d and M_dispatch.

default_mode

The string indicating the initial mode for any tvbuf.

dispatch

The dispatch array for all single character commands.

display_mode_line

The flag indicating that some item displayed in the mode line has changed and that it should be updated.
Every time the reader calls "display" it also calls "redo_mode_line" which just returns if display_mode_line is 0. If it is -1 the whole mode line is reprinted and if it is 1 then only the message part of the mode line is output.

echo_buf

The buffer (not tvbuf) used to read text into by the "read_line" function. It is used as an argument to the reader function when called from "read_line".

echo_mark

The mark used by "read_line" to separate the prompting string and the text typed by the user. Both go into the echo_buf.

echo_screen

The screen used to update the echo_buf on the screen. It is defined to occupy one line at the bottom of the screen by default.

echo_window

The window used by "read_line" to return its results. It does not copy the typed string but returns it in place by means of this window. Thus the value returned by "read_line" is not valid after a second call to "read_line".

editor_name

The string typed out as the first thing in the mode line. It is normally "TVmacs".

first_time

This is T or NIL depending on whether this it the first time the "startup" function has been called in this editor environment.

function_to_call

The variable containing the function called by the character last read by the reader. The control-U command resets this to be the command to be repeated.

hold

This is 1 if the next call to "display" is to be inhibited. This is used to prevent text printed with "print" or "print_clearing" from being erased immediately.

kill_buffer

This is the buffer in which saved kills are stored.

kill_ring

A circular ring of ten marks pointing into kill_buffer to separate the ten different elements of the kill stack.

last_buffer_in

The name of the last buffer which was the current_tvbuf. This is set by "goto_buffer" to the name of the current tvbuf before it goes to the new buffer. This is the buffer selected by "select_buffer" if the buffer name read in is zero length.

ibrary_dir

The string which indicates the directory from which auto loaded keys are loaded. It is ">sl1>sinemacs" by default.

inel

The number of characters per line on the user's terminal.

oaded_modes

The list of mode names whose associated packages have been loaded into the Sine environment. This is to prevent multiple loadings of the same mode package.

essage

The message printed by "redo_mode_line". "Search failed" and "n chars written" are typical messages. They are automatically cleared after one command is typed.

ld_save_delete_flag

This flag is copied directly from "save_delete_flag" before every command to allow the state to be preserved for one command.

page_overlap

This is the number of lines less than a full screen that the "next_page" and "previous_page" functions move. The default is 11 lines.

pagel

The number of lines on the terminal.

read_line_tg

The temporary gnirt bound in the "read_line" function to "tg". See also tg.

recursive_read_line

The flag that indicates that the echo area is in use and that calls to "read_line" should abort.

save_delete_flag

This flag is T if the last command was a command which saved some text on the kill_stack. This information allows consecutive deletions to be concatenated into a single element on the kill_stack. See also old_save_delete_flag.

search_string

The string which was last searched for. If the "string_search" function prompts for a search string and receives a zero length string from "read_line" it will search for this string again.

tg

This hold a temporary gnirt which is used by many command functions when they need a gnirt. Be sure when using this that you were not called by anyone who is using it.

token_chars

The bit array defining all the characters that are part of tokens. This is used by the word commands: "forward_word", "delete_word" etc. This is different from "alphanumerics" because it has some of the punctuation marks that want to be considered parts of word, such as underscore in PL/1_mode.

token_hackers

The list of functions that are to be called when ever a break character is self_inserted. In fact, all the printing characters other then letters and numbers call "self_insert_break" instead of "self_insert". "self_insert_break" calls all the functions on this list before inserting the character in "char". This feature is used by "fill_mode" to check the current line to see if it is too long whenever a break character is typed.

tunnel_path

The list of mode functions still to be called by "keep_on_tunneling" before it calls the reader.

white_space

The bit array defining all the white space characters: space, tab and carriage return.