Mince Internal Documentation
Table of Contents

Chapter 1

Program Logic Manual

1.1 Generalities

This is the program logic manual for Mince. It disucusses a variety of topics: First, it reviews the basic terminology used to discuss the editor and identifies the parts of Mince. Second, it explains the conventions and structure in the implementation. Third, it discusses the file MINCE.GBL and reviews the use for each of the variables found there. Finally, it warns about some potential pitfalls in modifying the existing code. These topics are not discussed in any particular order. Throughout the manual comments on customizing Mince will be made. These comments will be indented, as in:

> First customization note. When you are writing a new command, it is a good idea to make a copy of an existing function which does a related task and modify that copy. Writing new commands from scratch is much harder.

This chapter cannot be read by itself. In order to get a full understanding of the structure of Mince and how to modify it, the rest of the manual is needed. First and foremost is the Mince User's Guide. It can answer questions about general concepts: for example, why you want to use GetArg as opposed to Ask. It explains what Mince looks like to the user.

Next is the Complete Command List. That document gives the definition of each command in words. You can then look at the C source code and see how that command is actually implemented. That chapter is also useful in reverse. If you are looking at some code and can't figure it out, looking at the English description can be of great help. Both the Mince User's Guide and the Complete Command List appear in the Mince User's Manual.

The third chapter of use is "Theory and Practice of Text

Editing." This chapter describes the structure of a text editor and spends a lot of time discussing text buffers and the redisplay, two major parts of Mince for which source code is not supplied. In spite of the fact that the chapter was written only last May (1980) and Mince was written in October (1980), the underlying software technology has changed substantially. However, the chapter does provide a solid base upon which to understand Mince and the general approaches presented are still valid. Note that the buffer interface described in that chapter are not the same as those actually used in Mince.

The fourth item of interest is the Entry Point Documentation. This chapter documents each entry point in Mince for which source code is not supplied. It gives enough information for each entry point to allow you to use that entry point.

The fifth item of interest is the Source Documentation. It explains in great detail some of the more complicated routines for which source code is supplied. This item also describes the interface for each routine in SUPPORT.C. Understanding what these routines do is often the key to understanding a routine that they are used in. An extreme example is DoReplace. The entire Query Replace command (MQryRplc in COMM2.C) merely calls DoReplace(TRUE).

The final useful item for understanding Mince is the source code itself. In it, you see a concrete implementation of what all this documentation can only talk about in abstract terms.

1.1.1 Notes on Data Abstractions

Data abstractions are programming tools. Like any tool, they can either be appropriate to a given situation or not and they can be both correctly and incorrectly applied. We used them in Mince because they were appropriate and aided us in the development.

A data abstraction is a collection of subroutines and data. Only those subroutines are allowed to access that data, and that data serves as the sum knowledge that the subroutines have about the "world." An imaginary line can be drawn around these subroutines and data. All knowledge about the internal representation of the data is contained within this line.

The data abstraction is manipulated by calling the subroutines and passing them arguments. A subroutine is defined for every operation that can be done with the data.

The data abstraction is defined by its interface to the outside. Any internal arrangement of data and procedures that implements the interface is acceptable and DIFFERENT IMPLEMENTATIIONS ARE COMPLETELY INTERCHANGABLE! This last property is of critical importance. It allows substantial revisions of programs while confining these changes to only those procedures that are directly affected. For example, while developing Mince, we rewrote the buffer abstraction so as to provide virtual memory. No command routines had to be modified at all. Without the high degree of information isolation provided by the buffer abstraction, this change would have proven much more difficult.

In summary, a data abstraction is a way to contain information. It does this by hiding all of the internal representations and only showing the "outside world" a clean, simple interface.

1.1.2 Quick Review of Mince

Mince is a multi-buffer in-memory text editor. This means that it edits a copy of a file by reading it into a text buffer. This buffer is usually in main memory (i.e. RAM). Having more than one buffer allows you to edit several files at once by copying each of them into a different buffer, then switching back and forth from buffer to buffer. Since the available main memory space (RAM) may not be large enough to store all of the text which can be read into buffers, the existence of additional main memory is simulated by a virtual memory system, which copies parts of buffers from main memory to disk (i.e., secondary storage) and vice versa as necessary. (The buffer abstraction, whose entry points are described elsewhere, handles both of these functions.)

Each buffer has a Point associated with it. All editing changes take place at the Point. There are also a number of marks which can be placed anywhere in any buffer. A mark will stay in the same place with respect to the surrounding text no matter what changes go on around it. Each buffer always has a distinguished mark associated with it. It is this mark that is referred to as "the Mark" in the user documentation.

A buffer also has a mode list. Modes are ways of tailoring the command set on a per-buffer basis. Modes are coded in C and represented in Mince as the differences between the default command set and the desired one.

Each keystroke that is typed invokes a function to implement the meaning associated with (or "bound to") that key.

> Customization Note. When devising modes, it helps a lot if they don't rebind an existing command or, if they do, that the same command key is not also rebound by another mode. Page mode and Fill mode have this problem: if you invoke Fill mode first, after Page mode is added it destroys the definition of Auto Fill Space which was bound to the Space key. Such things are annoying at best.

Mince is implemented as an editor within an editor. The outer editor interfaces to the user and reformats his or her desires so as to be executable by the inner editor, known as the buffer abstraction. It might help to think of its interface as a user's manual for the inner editor.

The buffer abstraction sub-editor takes all memory left over after the code, globals, and operating system have taken what they need and divides it into 1K pages. These pages are used to store the contents of the buffer. They are swapped between memory and the swap file on an LRU (least recently used) basis.

> In any virtual memory scheme, a page must be swapped out of main memory to make room for the desired page. An LRU scheme is one where the page that was least recently accessed is the one that is chosen to be swapped out. Swapping out a modifed page (the copy in memory is different from the copy on disk) requires that the page be written to disk. Swapping out an unmodified page doesn't require any activity. Thus, the LRU scheme that Mince uses has been modified to try to swap out unmodified pages first as doing so takes less time. That is also why Mince swaps out modified pages (making them unmodifed) when it is otherwise idle; when you start editing again, it has less work to do.

BFlush is used to implement the delayed write-through. If the system is idle, pages are written through one by one to the swap file until everything out there is current. Note that there are two types of modified pages. One type is what the user sees: a page (buffer) has had an insert or delete performed upon it. The other type does not imply insertion or deletion. Rather, it means that the page is different from the copy of it on the swap file. It therefore must be written to the swap file before its memory can be reused. The latter case happens, for example, when reading a file in for the first time. Although the buffer is unmodifed, the pages have to be swapped through to disk.

Redisplay is the process of updating the user's terminal's screen to reflect the current contents of the buffer. Mince command routines need know nothing about this process as it is handled automatically by the sub-editor. The redisplay process is invoked each edit command cycle, just before the editor asks for a command. If a command has been entered, or if one is entered as redisplay occurs, the redisplay is aborted and the command is executed immediately.

The redisplay operates by scanning the buffer and comparing its current contents with the redisplay's internal model of what is on the screen. The internal model consists of an array of special screen marks, one for each screen line. Each screen mark has a modified flag associated with it. Whenever an insert or delete operation takes place, the buffer abstraction automatically sets the modified flag on the associated screen mark. The redisplay process thus can determine which parts of the screen could have been affected.

1.1.3 Code Structure

The basic edit loop is as is disussed in "Theory and Practice of Text Editors," chapter Five. Function Main calls Setup and then calls Edit. Setup does a lot of initialization, but the important thing is that it calls SetModes.

SetModes is a critically important function. It is called at initialization, when switching buffers, and when adding or deleting modes. It is responsible for initializing the command bindings to their new values. It fulfills this task by calling, in turn, finit1, finit2, and finit3 which set up the default bindings for Control, Meta, and Control-X commands, respectively. It then modifies these defaults by going down the mode list for that buffer and performing the tailoring specific to each mode.

> Customization Note. When adding or removing a mode from the code, the change must occur in two places. First is in SetModes. The other place that the change must take place is in CheckModes in SUPPORT.C.

After SetModes has been called, Setup returns and then Edit is called. Edit performs an IncrDsp (incremental redisplay), waits for a character, and dispatches to the command routine that the character is bound to. (A "command routine" is any procedure which can be called directly from a dispatch table.) Arg is set to 1 and Argp is set to FALSE.

Any further calls to SetModes will be from commands (e.g., Switch Buffers).

At this point, each command can assume the following environment:

Arg is set to 1
Argp is set to FALSE
Lfunct points to the function bound to the command that was executed before this one (the "last function" executed)
Cmnd is set to the character that was typed by the user to generate this command

Note that "this point" does not always exist. If a C-U (Universal Argument, MArg in COMM1.C) is typed, it will eventually dispatch again to the commands. In that case, Arg will in general not be 1, Argp will be TRUE, and Cmnd will still be the character that was bound to you. Similar changes happen with the Meta and Control-X dispatch functions. Note that 128 <= Cmnd <= 255 indicates a Meta command and 256 <= Cmnd <= 383 indicates a Control-X command.

If Arg is greater than one when you return back to edit, edit will decrement it and call you again.

1.2 Specifics

1.2.1 Supplied Files

There are several source files supplied with Mince. They are:

BINDINGS.C	Source code for the key binding functions finit1, finit2, finit3, and SetModes.
COMM1.C	Source for Control commands. COMM1, COMM2, and COMM3 have the routines listed in the same order as appears in the full command list, i.e. in the same order as the ASCII collating sequence for the keys to which they are bound.
COMM2.C	Source for Meta commands.
COMM3.C	Source for Control-X commands.

SUPPORT.C	Source for support routines used by the commands. They are listed in alphabetical order.

1.2.2 Coding and Documentation Conventions

The Entry Point Documentation, the Source Code Documentation, and the Terminal Abstraction Documentation follow certain conventions. First, the name and type of each argument is given; the return argument is only given if the routine specifically returns a meaningful value. (In C, everything returns a value.) Second, each global variable that the routine accesses is listed in those routines for which the source code is not given. Here is a guide to interpreting the ways that globals are used:

Exports means that that routine sets the global for other routines' use.
Imports means that that routine reads the value of that global.
Private means that no other routines should look at or set that variable.
Updates means that that routine both reads and writes the value of that global.
Uses means that that routine bashes any existing value and leave a garbage value in that global.

Note that there is no listing of which globals any of the command routines access. In general, they utilize globals heavily and a quick check of the source code can tell you which of them a particular routine uses.

The upper/lower/mixed caseness of names also has meaning.

UPPERCASE names are constants. They are initialized with #defines and are only initialized in MINCE.GBL.
MixedCase names are procedures. A capital letter usually indicates the start of a new word (in lieu of a space or underscore).
lowercase names are variables or procedures. When procedures, they usually name a procedure used only locally.

Within written English text, a lowercase name will often be Capitalized in order to facilitate its recognition as a variable or procedure name.

The first letter of a procedure name usually has meaning as well. Note that the routines in SUPPORT.C ignore this convention completely.

letter	indicates that the procedure is part of the...
A	dynamic memory Allocator
B	Buffer abstraction
C	top level buffer abstraction that makes the buffer abstraction Compatible with the Mince user-visible view of buffers
M	Mince command set
Q	FIFO Queue maintainer
T	Terminal abstraction

1.2.3 Constants and Globals

This section discusses the contents of the file MINCE.GBL. It briefly covers the meaning of each of the constants and globals.

Basic constants and variables:

```
TRUE       (-1)
FALSE      0

NULL       0     -- the null pointer

HOME       0,0   -- TSetPoint(HOME) puts you at the upper
                       left corner

FORWARD    (-1) -- altername names for TRUE and FALSE to
BACKWARD   0          enhance readability in some places

SWAPFNAM   "mince.swp"    -- two places where it looks
SWAP1FNAM  "a:mince.swp"       for the swap file

INPUT      0     -- mode for file input
OUTPUT     1     -- and output
UPDATE     2     -- and update

FILMAX     15    -- maximum length of a filename
STRMAX     40    -- maximum length of a search string
MODEMAX    20    -- maximum length of the mode name string
MAXMODES   4     -- maximum number of modes (per buffer)
BUFNAMMAX  9     -- maximum length of a buffername
BUFFSMAX   7     -- maximum number of buffers
```

mark	-- user settable mark in the current buffer
arg	-- the numeric argument to a command
argp	-- TRUE if an explicit argument was entered, FALSE otherwise
lcol	-- column that Next Line and Previous Line try to leave you in
psstart	-- a mark placed one character before the start of the screen
sstart	-- a mark placed at the start of the screen
send	-- a mark that tries to be placed at the end of the screen (not valid if redisplay was interrupted)
cnt, tmp	-- scratch variables for local use by commands or support routines. Watch out for calling and called routines that use the same variable!
(*functs[3*128])()	-- the key bindings table
(*lfunct)()	-- the previous command invoked
	for commands that deal with margins...
fillwidth	-- the first column text can't be in
indentcol	-- the first column text can be in
strarg[STRMAX]	-- previous search string
mode[MODEMAX]	-- current mode name sting
namearg[BUFNAMMAX]	-- previous buffer name
pnt_row	-- screen line the Point was on in the last redisplay
stat_col	-- column that the statistics (-%-, etc) begin in
abort	-- set it to TRUE if you want to exit the editor (abort out of the command loop)
cmnd	-- current command character (128 <= cmnd <=255 is a Meta command, 256 <= cmnd <= 383 is a Control-X command)
cbuff	-- index of the current buffer in the "buffs" structure
del_buff	-- buffer descriptor of the delete

```
                      buffer for use with the buffer
        abstraction (e.g., BSwitchTo)

    tmark               -- scratch variable used to hold a mark

                        user visible buffer structure
    struct cbuffer {
         bbuff                    -- buffer descriptor for use
                                  with the buffer abstraction
         bmark                    -- the user settable mark
         bname[BUFNAMMAX]         -- the buffer name
         fname[FILMAX]            -- the associated filename
         bmodes[MAXMODES]         -- the list of mode ids for
                                  this buffer
         } buffs[BUFFSMAX]
```

Terminal Abstraction constants and variables:

```
    ROWMAX     60   -- maximum configurable # of rows
    COLMAX     132  -- maximum configurable # of columns

    NUL  '\0'       -- ASCII NULL character (0 decimal, ^)
    BELL '\7'       -- ASCII BELL character (7 decimal, ^G)
    BS   '\10'      -- ASCII BACK SPACE character
                         (8 decimal, ^H)
    TAB  '\11'      -- ASCII HORIZONTAL TAB character
                         (9 decimal, ^I)
    LF   '\12'      -- ASCII LINE FEED character
                         (10 decimal, ^J)
    CR   '\15'      -- ASCII CARRIAGE RETURN character
                         (13 decimal, ^M)
    ESC  '\33'      -- ASCII ESCAPE character
                         (27 decimal, ^[)
    DEL  '\177'     -- ASCII DELETE character
                         (127 decimal, ^?)
    NL   '\212'     -- character that Mince uses to mean
                         Newline (138 decimal, ~^J)

    KBBUFMAX  80    -- maximum number of typeahead
                         characters

    prow            -- position of the screen point
    pcol
    srow            -- position of the cursor
    scol

    tabincr         -- number of columns between tab stops
```

```
tlrow           -- row number of the saved screen row

clrcol[ROWMAX]  -- for each row, the number of the
                     column after the last non-blank
                     character

tline[COLMAX]   -- the saved screen line (for redisplay)

lindex          -- temporary pointer into tline

struct {        -- keyboard input queue structure
     head
     tail
     bottom
     top
     space[KBBUFMAX]
     } kbdq
```

Terminal Abstraction variables that are read from the swap file header. The swap file header was written to disk by Config.

```
NOPAD      0    -- for "padp", if Padp==NOPAD, no
CHARPAD    1         padding is necessary, if Padp==
DELAYPAD   2         CHARPAD, pad with characters, if
                     Papd==DELAYPAD, pad with wait loop

FIRST      255  -- possible arguments to put_coord.
SECOND     0         Tells whether to put out the first
                     or the second coordinate this time.

                     structure describing the terminal
struct termdesc {
     ctrlz      -- ^Z to tell the world to stop reading
                     the file, as it will contain what
                     looks like garbage
     nrows      -- number of rows on the terminal
     ncols      -- number of columns on the terminal
     rowbias    -- bias to add to the desired row and
     colbias    -- column when doing cursor positioning
     rowfirstp  -- TRUE if the row should be sent first
     compp      -- TRUE if the row and column should be
                     bitwise complemented after biasing
                     and before sending
     binaryp    -- TRUE if the row and column should be
                     sent in binary, FALSE if they
                     should be sent in ASCII
     padp       -- tells how to pad commands
     padchar    -- what character to pad with for
```

```
                       padp==CHARPAD
      nhclpad  -- amount of padding to do for home and
                       clear screen
          ncleolpad -- amount of padding to do for clear to
                       end of line
          ncpospad  -- amount of padding to do for cursor
                       positioning
          ncleowpad -- amount of padding to do for clear to
                       end of screen
```

For each of the operations that follow, this is the index and length of the character string to send:

```
          struct str {
               idx  -- index
               len  -- length
               } hcl      -- home and clear screen
                 cleol    -- clear to end of linen
                 cleow    -- clear to end of window (screen)
                 cpos1    -- prefix string for cursor
                               positioning
                 cpos2    -- string to separate the two
                               coordinates
                 cpos3    -- postfix string for cursor
                               positioning
                 bell     -- ring the terinal bell
                 init     -- intialize the terminal
                 deinit   -- leave the terminal in a
                               reasonable state

          strspc[73]      -- the space that the above
                               operations index into.  It
                               must be 73, as it fills
                               out the disk block.
          } terminal
```

I/O port descriptions:

```
     struct portdesc {
          biosp     -- TRUE if the bios is used.  If FALSE,
                         the rest is relevant:
          dataport  -- number of the data port
          statport  -- number of the status port
          datamask  -- ANDed with incoming data
          readymask -- ANDed with status to deterine whether
                         a character is waiting or the port
                         is ready for output
          polarity  -- polarity of the relevant bit.  TRUE
```

} inport outport	if "1" bit means the port is ready --for both input and output ports

Random parameters:

prefrow	-- preferred cursor row
fillinit	-- initial fill column
tabinit	-- initial indent column
indentinit	-- initial tab increment
mhz	-- processor speed in tenths of megahertz
delaycnt	-- delay constant for echoing "Meta: ", etc. as well as wait time before swapping starts
npages	-- number of pages in swap file. Must be a multiple of 8
swapbase	-- base of the actual swap area in sectors

Spare area for patches:

spare[10]	*-- ten integers' worth*

Note that you cannot alter these declarations in any way. However, the Spare variables are available for use by any routines that you write.

1.2.4 Conditional Compilation Flags

Mince has conditional compilation flags scattered throughout the source code. These flags are used to tailor the Mince source code for a variety of machines and operating systems. The flags are:

UNIX RSX CPM	-- indicates the operating system. only one of these can actually be on
SUSER	-- single user system
LARGE	-- extra command memory available
TYPEAHEAD	-- typeahead is detectable

Note that for CP/M systems, the only flag that can be altered is the LARGE flag.

1.3 Extending and Modifying Mince

1.3.1 An Example

Let us uncover some of the potential pitfalls and see how all of this hangs together by writing an example function. This function is a sort of poor man's detabify. It's called MDeTab and it will find the next Tab character and replace it with the number of spaces that it represents.

A noticable amount of implicit knowledge was used in the above paragraph. First, the knowledge of how routines are named indicated that the function name should be prfixed with "M". ("M" stands for "Mince.") Second, the knowledge that the general function (detabification) is useful is implicit in selecting this particular example. Third, the knowledge of how this function could fit in with the "Mince philosophy" to serve as the foundation for a "Detabify Region" command helped to shape the definition. The "Mince philosophy" is something that is gradually acquired by writing a (possibly large) number of commands and trying to fit them in with the existing structure.

The first step is to establish the algorithm. This step is not as forbidding as it sounds. All that it implies is that we rewrite the above description in a more formal way:

```
find the next Tab
figure out how wide it is
replace the Tab with that many spaces
```

Or, yet more formally:

```
search forward through the buffer for a Tab
figure out how wide it is
delete the Tab
insert the correct number of spaces
```

In C/Buffer Abstraction "Language" this would be:

```
BCSearch(Tab)
width=TWidth(column,Tab)
BDelete(1)
call BInsert(" ") WIDTH times
```

Note that this is NOT a finished function and, as it stands, it will not work. It is left at this stage to point out different ways of making two coding decisions. First, the call to TWidth might have been coded as a BGetCol, a move backwards one character, and a second call to BGetCol. The difference between the returned values in the column positions is the answer (and is the same number that TWidth will return). Second, the insertion of the WIDTH spaces can be by an explicit for- or while- loop or it can be via a call to Indent. This example illustrated that there is probably a function in Mince that directly does what you want; your job is to ferret it out.

The finished function looks like this:

```
MDeTab()              /* change a Tab to blanks */
{
     if (!BCSearch(TAB)) return;
     BMove(-1);
     SIndent(TWidth(BGetCol(),TAB));
     BDelete(1);
     }
```

The if statement puts in a very important check: if there is no Tab, we don't do anything. Checks of this sort are very important in finished code. However, they are not relevant to the definition and so were left out of the earlier discussion. We then go backwards over the Tab. Thus, the BGetCol will return the desired column (the one you are in just before the Tab is "printed"). TWidth takes this column and returns the width of the Tab. SIndent puts in that many spaces. The BDelete then deletes the Tab.

If we were to write a Detabify Region command, this function would serve as a good base. It does need some touching up, however. First, it should be passed a mark which was placed at the end of the region. MDeTab would then not do anything if the BCSearch left you after the mark. Second, MDeTab would probably return either TRUE or FALSE, depending upon whether it did anything. The Detabify Region command could check this flag to determine whether to continue on in the loop.

1.3.2 On Changing Mince

As the previous example indicated, there is a lot to know before you change or extend Mince. The best way to acquire some of this knowledge is to first become an expert in using Mince. After all, it is wasteful to write a Center Line command if one already exists and you merely didn't know about it. That knowledge will help you to figure out how existing code works and it will also help you have your modifications fit in with the "Mince philosophy." Unless you are reworking the entire command set and user interface, users will be much happier if any changes are in line with the existing philosophy of the program. It is easier to learn and be happy with an undesirable philosophy that consistently implemented than with the same philosophy that has been changed here and there so as to be "less undesirable." Of course, we feel that our philosophy is not undesirable...

1.3.3 Compiling and Linking Mince

There are a number of points to consider when compiling and linking a version of Mince. First, the BDS C compiler (Version 1.42 or higher) must be used for compiling Mince.

The object code files (.CRL) are distributed in two forms. Both forms were compiled with the -e option. (We estimate a 25% increase in size without this option.) The object code files for which source code is not supplied are:

MINCE.CRL	LMINCE.CRL
UTIL.CRL	LUTIL.CRL
VBUFF1.CRL	LVBUFF1.CRL
VBUFF2.CRL	LVBUFF2.CRL

The normal-named versions use -e7900 and the versions starting with "L" use -e8100. The extra 2K of space allows room for adding functions to Mince. Note that there is essentially no extra space in the -e7900 versions so if you add code there, you must take out something else. If more space is required, contact us and we will generate a special version.

Note: when compiling SUPPORT.C, the -r option must be used to prevent symbol table overflow (use -r10).

When linking, use the linker supplied by Mark of the Unicorn (called "L2") and NOT the linker supplied by BDS. (Among other reasons, our linker saves 10% of code space.) This linker is experimental and is not guaranteed to link any arbitrary C program. It will, however, link any software supplied by us.

The following command line will successfully link a Mince (it can be found in ML.CMD):

```
l2 mince bindings comm1 comm2 comm3 vbuff1 vbuff2 -l
```

```
support term util
```

Almost nothing else will. About all that you can do is to split or include new command files, in which case be sure that they are before the "-l". If you would like to know more about the linker or have any problems, contact us.

1.3.4 Debugging Code

So, you have written some new functions and would like to debug them. Debugging a display editor is not quite the same as debugging anything else. For one thing, the screen has a tendency to rearrange itself on you...

The best method to use is a modification of the basic tracing techniques that every programmer has used: put in a print statement. The function Debug is the print statement and you simply have to call it every place that you would like to print a value or message. Debug does the following:

it prints a message
it prints the value of an integer
it does an redisplay (so you can see what the buffer
  looks like)
it waits for you to type a character, and returns
  TRUE if the entered character was a Delete or
  BackSpace, FALSE otherwise

It is a good idea to sprinkle a lot of calls to Debug in any suspicious code, especially in infinite loops. Each message should be different (to allow you to see what part of the loop you are actually in. In this manner, you can see the "intermediate results" of your function. You can also get out of the function to try something else by doing:

```
if (Debug("I am here",cnt)) {
     arg=0;
     return;
     }
```

Don't even think of trying to use a conventional debugger with Mince. You haven't got a chance.

Good Luck!

Chapter 2

Entry Points

2.1 Top Level and Redisplay Routines

```
main(argc,argv)
     int argc
     char *argv[]
```

This is the command line entry point.

```
setup(iargc,iargv)
     int iargc
     char *iargv[]
```

Exports the globals fillwidth, del_buff, indentcol, namearg, stringarg, and tabincr.

Initialize the editor. It is called immeidately by main and may not be reinvoked. Fillwidth is initialized to the default fill width and indicates the first column that characters may not appear in while filling text. Indentcol is initialized to the default indent column and indicates the first column that characters may appear in while filling text. Tabincr is initialized to the default tab increment and indicates the number of columns from one tab stop to the next. Namearg is initialized to the null string and retains the default used for switching buffers. Stringarg is initialized to the null string and retains the last search string. Del_buff is a buffer descriptor and defines the kill buffer.

The last act performed is to call the routine UInit to perform any initialization desired by the user.

edit()

Exports the globals abort, arg, argp, cmnd, and lfunct. Imports the global functs.

The basic editor loop. It reads a command, dispatches it, and invokes redisplay. It may be called recursively and setting the global abort to TRUE will exit the current invocation. Argp indicated the presence (TRUE) or absence (FALSE) of an argument to a command and is initialized to FALSE. Arg is the actual argument value and is initialized to 1. Cmnd contains the key that was typed combined with any prefix codes. Lfunct is the address of the procedure invoked by the previous command.

NewDsp()

Clears the screen, does a ScrnRange, and forces redisplay.

IncrDsp()

Imports the globals psstart, send, sendp, and sstart.

Performs a redisplay. Updates the screen, one line at a time, to reflect the current state of the buffer. If a character is typed during redisplay, it aborts after finishing a line. Upon successful completion, Send is placed at the actual end of the window. It also calls ModeFlags.

```
int
InnerDsp(from,to,pmark)
    int from, to, pmark
```

Exports the globals send, sendp, and tline.
Imports the globals cur_cptr and terminal.
Private globals lindex, pnt_row and tline.
Updates the global pcol.

Redisplays a single window for IncrDsp. The window runs from screen lines From to To. If Pmark is non Null, returns the row that the mark is on. This routine should not be called outside IncrDsp.

ScrnRange()

Exports the globals psstart, sstart, and send.

Centers the window so that the Point is on PREFLINE. Sstart is a mark which is placed at the start of the window. Send is a mark which approximates the end of the window. Psstart is a mark which is placed one character before the start of the window.

int
WHeight()

Imports the globals divide and topp.

Returns the height of the current window in lines.

int
PrefLine()

Imports the global prefrow.

Returns the line within the current window that the user prefers the Point to be on. This value is linearly dependent upon the position within the window of the value of the "preferred row" parameter given in Config.

ModeLine()

Exports the global stat_col.
Imports the global buffs, cbuff, and mode.

Displays the mode line. Stat_col indicates where to start displaying the mode flags.

2.2 User Level Buffer Description

The data structure is:

```
     struct cbuffer {
          int bbuff, bmark;
          char bname[BUFNAMMAX], fname[FILMAX], bmodes[MAXMODES];
          } buffs[BUFFSMAX];
```

```
int
CMakeBuff(buffername,filename)
     char *buffername, *filename
```

Creates a buffer named Buffername with an associated filename Filename. It also sets the modes to no modes, creates a mark, and sets bbuff to be a buffer descriptor of a new buffer. Returns the index of the newly created buffer, or -1 if it failed. Buffername must be unique. If it is not, the results are undefined.

```
CSwitchTo(bufferindex)
     int bufferindex
```

Exports the globals cbuff and mark.

Makes the buffer selected by Bufferindex the current buffer, calls SetModes, and makes the mark associated with that bufferthe global mark. Cbuff is a bufferindex and is the index of the current buffer.

```
int
CFindBuff(buffername)
     char *buffername
```

Returns the index of the buffer whose name is Buffername or -1 if there is no such buffer.

2.3 Memory Allocation Abstraction

The following routines implement a dynamic storage allocator and are used internally by the buffer abstraction. They cannot be used outside of the buffer abstraction.

```
     AAlloc         ACoalesce      AFindNext
     AFree          AInit          ALen
     APrtWrld       ASpace
```

They require the following globals:

```
     int *Abegin, Asize, *Aend, *Acap
```

2.4 Queue Abstraction

The following routines implement a FIFO character queue. They all use the following definition of a queue.

```
     struct queue {
          char *qhead, *qtail, *qtop, *qbottom, qspace[size]
          }
```

```
QInit(queue_pointer,size)
     struct queue *queue_pointer
     int size
```

Format a queue structure that you have allocated and pass to QInit. Size tells the initializer how big the queue should be. You must allocate the space yourself.

This space includes both the space for the queue and the space for the queue header. Size here and the size in the structure declaration are the same.

```
char
QGrab(queue_pointer)
     struct queue *queue_pointer
```

Returns the next character on the queue. Results are undefined if the queue is empty.

```
QShove(char,queue_pointer)
     char char
     struct queue *queue_pointer
```

Places Char onto the queue. Results are undefined if the queue is full.

```
FLAG
QEmpty(queue_pointer)
     struct queue *queue_pointer
```

Returns TRUE if the queue is empty; FALSE otherwise.

```
FLAG
QFull(queue_pointer)
     struct queue *queue_pointer
```

Returns TRUE if the queue is full; FALSE otherwise.

2.5 Buffer Abstraction

2.5.1 Initialization and Buffer Manipulation

```
BInit(swap_file_descriptor)
     int swap_file_descriptor
```

Initializes the buffer abstraction and tells it to use the indicated file as the swap file.

```
int
BCreate()
```

Creates a new buffer and returns its buffer_descriptor. Returns NULL if no more buffers are available or there is no more memory.

```
BSwitchTo(buffer_descriptor)
     int buffer_descriptor
```

Makes the buffer indicated by Buffer_descriptor the current buffer.

```
BDelBuff(buffer_descriptor)
     int buffer_descriptor
```

Deletes the buffer indicated by Buffer_descriptor. You cannot delete the current buffer (indicated by the global Cbuff), and you will get an error message if you try.

2.5.2 Inserting and Deleting Text

```
BInsert(character)
     char character
```

Inserts Character at the Point. It gives an error message and does nothing if there is no more available (virtual) memory.

```
BDelete(quantity)
     unsigned quantity
```

Deletes Quantity characters forward from the Point.

```
BDelToMrk(mark)
     int mark
```

Deletes all text between the Point and the passed mark. The passed mark must be in the current buffer, and you will get an error message if it is not.

```
BCopyRgn(mark,buffer_descriptor)
     int mark
     int buffer_descriptor
```

Copies the text between the Point and the passed mark into the indicated buffer, inserting the text at that buffer's Point. The passed mark must be in the current buffer, and you will get an error message if it is not. You cannot copy into the same buffer that you are copying from, and will get an error message if you try. Leaves the destination buffer's Point after the inserted text. If it runs out of available memory, it will abort after having copied as much as it can and print the message "Swap file full".

2.5.3 Beginning of Buffer, End of Buffer, and Basic Motion

```
BToStart()
```

Moves the Point to the beginning of the buffer.

```
BToEnd()
```

Moves the Point to the end of the buffer.

BShoveIt()

Places the buffer in a repeatable, invalid state. It is used internally by redisplay. Results are undefined if it is invoked outside of redisplay.

FLAG
BIsStart()

Returns TRUE if the Point is at the beginning of the buffer, FALSE otherwise.

FLAG
BIsEnd()

Returns TRUE if the Point is at the end of the buffer, FALSE otherwise.

BMove(dist)
int dist

Moves the Point Dist characters. Dist may be either positive or negative. It will move up to but not past either the beginning or the end of the buffer.

2.5.4 Status and Complex Movement

UNSIGNED
BGetCol()

Returns the display width of the text between the beginning of the buffer or the latest Newline and the Point. Normally, this will be the column that the cursor is displayed in.

BMakeCol(column)

```
     int column
```

Moves the Point so that the display width of the text between the beginning of the buffer or the latest Newline and the Point is as near Column as possible. The Point is placed at the end of the current line if Column is beyond the end. If Column falls in the middle of a multi-column character, the Point is placed after the character.

```
UNSIGNED
BLocation()
```

Returns the Position of the Point in the buffer in units of characters from the beginning of the buffer. This value is not correct for locations above 65,535.

```
UNSIGNED
BLength(buffer_descriptor)
     int buffer_descriptor
```

Returns the length of the buffer indicated by Buffer_descriptor in characters. This value is not correct for buffers of more than 65,535 characters.

```
FLAG
BCSearch(what)
     char what
```

Search forward through the buffer starting at the Point for character What. Returns TRUE if the character was found, FALSE otherwise. The Point is left after the character that was searched for if it was found or at the the end of the buffer if it was not.

```
FLAG
BCRSearch(what)
     char what
```

Search backward through the buffer starting at the Point for character What. Returns TRUE if the character was found, FALSE otherwise. The Point is left before the character that was searched for if it was found or at the beginning of the buffer if it was not.

```
FLAG
BModp(buffer_descriptor)
     int buffer_descriptor
```

Returns TRUE if the buffer indicated by Buffer_descriptor has been modified since it was created or had a file operation (e.g. read or write) performed upon it, FALSE otherwise.

```
char
Buff()
```

Returns the character after the Point. Results are undefined if the Point is at the end of the buffer.

2.5.5 Mark Manipulation

```
int
BCreMrk()
```

Creates and returns a mark and places it at the Point. An error message is given and NULL is returned if there are no more available marks.

```
BKillMrk(amark)
     int amark
```

Removes the mark Amark.

```
BMrkToPnt(amark)
     int amark
```

Places the mark Amark at the Point.

```
BPntToMrk(amark)
     int amark
```

Places the Point at the mark Amark. The passed mark must be in the current buffer and an error message is given if it is not.

```
BSwapPnt(amark)
     int amark
```

Interchanges the positions of the Point and the mark Amark. The passed mark must be in the current buffer and and error message is given if it is not.

```
FLAG
BIsAtMrk(amark)
     int amark
```

Returns TRUE if the positions of the Point and the mark Amark are the same, FALSE otherwise. The passed mark must be in the current buffer and an error message is given if it is not.

```
FLAGB
IsBeforeMrk(amark)
     int amark
```

Returns TRUE if the position of the Point is before the position of the mark Amark, FALSE otherwise. The passed mark must be in the current buffer and a value of FALSE is returned and an error message is given if it is not.

```
FLAG
BIsAfterMrk(amark)
     int amark
```

Returns TRUE if the position of the Point is after the position of the mark Amark, FALSE otherwise. The passed mark must be in the current buffer and a value of FALSE is returned and an error message is given if it is not.

Screen marks are special marks used by redisplay to improve its performance. Each mark has a flag associated with it. Redisplay clears all of these flags and the buffer modification routines (e.g., BInsert and BDelete) set the flag on the the last mark located before the modification. As the redisplay places the marks on the beginning of each screen line, the flag serves to indicate whether that line has changed since the last redisplay.

```
int
BScrnMrk(indx)
     int indx
```

Returns the screen mark corresponding to the Index'th row of the screen.

```
FLAG
BTstMrk(smark)
     int smark
```

Returns the state of the mark Smark and resets the state to FALSE. It is used internally by redisplay. Results are undefined if it is invoked.

```
BSetMod(smark)
     int smark
```

Sets the state of the mark Smark to TRUE. This will result in the corresponding line being redisplayed the next time IncrDsp is invoked.

2.5.6 Reading and Writing Files

```
FLAG
```

```
BReadFile(filename)
 char *filename
```

Read the contents of the file Filename into the buffer. The current contents of the buffer are lost. All marks and the Point are placed at the beginning of the buffer. Succeeded is TRUE if the read was successful, FALSE otherwise. The buffer's modified flag is cleared.

```
BWriteFile(filename)
     char *filename
```

Write the contents of the buffer into the file Flename. The buffer's modified flag is cleared.

```
BFlush()
```

If there are any modifed pages, one is written to the swap file, otherwise, nothing happens.

2.5.7 Private Routines

The following are routines private to the buffer abstraction. They may not be invoked from outside the buffer abstraction.

DskWarn	DskUnWarn	free_page
get_memp	GetGap	into_mem
make_cur	make_offset	new_page
page_split	SetMod	SubSet

Chapter 3

Source Code

Note that not all routines will be documented here; many of the commands are left out. Understanding these more simple routines is tantamount to understanding what the command is supposed to do.

3.1 Control Commands: File COMM1.C

MArg()

Does the C-U (Universal Argument) command.

Tmp accumulates the numeric argument as typed in (e.g. '5' then '3' becomes the value fifty-three). Tmp does NOT accumulate the multiplications by four. Cflag tells whether or not a digit or digits was entered. Eflag tells whether the argument has been echoed and thus needs clearing.

As the routine is entered, any previous argument is multiplied by four. A (possibly null) digit sequence is picked up and accumulated. If the digit sequence was non-null, the old argument is thrown out and those digits become the argument. You can thus end an argument with any number of C-U's. The non-digit which ends the digit sequence then gets dispatched as a command. If a C-U was picked up, this routine is invoked recursively.

3.2 Meta Commands: File COMM2.C

MDelELine

Does the M-C-K (Kill Entire Line) command.

The tricky thing in this routine is that it does the delete in two separate operations. It first deletes the text from the Point to the beginning of the line and puts it at the START of the delete buffer. It then deletes the text to the end of the line (including the Newline) and puts it at the END of the delete buffer. If the command is given an argument, the Point is left at the beginning of the following line and reinvoked. All of the text from that and following lines will be put at the end of the delete buffer.

MCapWord

Does the M-C (Capitalize Word) command.

The tricky thing here is that if the word that is being capitalized is only one character long, you don't want to call the lowercase word routine.

MFillPara

Does the M-Q (Fill Paragraph) command.

It begins by resetting the fillwidth if there was an argument. It then creates a different mark so that it can return there when it is done.

The rest of the initialization involves putting the point and a mark in the right places. A BMove(-1) is done so that it will fill "this" paragraph if the Point is just after the last non-white character (to wit, the final period of the final sentence). It then goes to the end of the paragraph and backward a character yet again. If you are at the end of the buffer (BIsEnd is TRUE), you must have a null buffer and so return. If that character is whitespace, you have only whitespace in that direction in your buffer (Forward Paragraph always leaves you just after non-whitespace, if possible), therefore, you return. Otherwise, you set a mark there and go back to the beginning of the paragraph to start filling.

The basic flow of the rest of the command is to bounce through the paragraph, stopping at each Newline and before each word and doing some processing, finally finishing when you reach that mark at the end of the paragraph. Note that breaks are only made on whitespace (e.g., it won't split "a-b") and whitespace can be deleted or inserted by the command.

There are two cases. First, a word can end after Fillwidth. Second, a Newline can occur before Fillwidth. In the first case, you jump backwards to the previous whitespace, delete it, insert a Newline and any indentation, and jump forward again. You never have to back up past whitespace more than once (unlike MFillChk, the auto fill space routine) because you were just there and it wasn't past Fillwidth. You jump forward to whitespace as you finish for efficiency and to keep from hanging in an infinite loop if you have a single word longer than Fillwidth. In the second case, you delete the Newline and any indentation and insert a space.

In the center of the text of the routine and separating the two cases is the mechanism for switching between the two cases. No matter which case you just processed, you skip over any blanks or Tabs and stop when you get to a Newline or a non-whitespace character. A call to IsNL is made to see which case it was that you encountered and the result of that call selects the case to handle.

The routine finishes by returning the Point to the original mark and cleaning up.

Note that in this scheme all Newlines will be deleted and re-inserted, even though no actual motion of text is needed. *sigh*.

MCntrLine

Does the M-S (Center Line) command.

This begins by getting Fillwidth and Arg to be the same value. Arg performs double duty here. It comes in as the argument, of course, but it goes on to become the value Fillwidth - Indentcol.

The routine itself moves to the beginning of the line, deletes any whitespace (for example, the stuff it put in last time if the line is being re-centered), goes to the end of the line, gets its position, goes back to the beginning, inserts the right number of spaces, and leaves you at the end.

MFillChk

Does the Fill Mode Space (Auto Fill Space) command.

It begins by returning if it doesn't have to do anything (the Point is before Fillwidth). If the Point is at or after Fillwidth, it has to split the line. It jumps back to whitespace until it gets before Fillwidth (it can't assume that the immediately previous whitespace is before Fillwidth). It then deletes that whitespace.

Now things get hairy. As you might have noticed, it placed a mark before it did anything. If it is still at that mark, that means there was lots of trailing whitespace on the line and it is the whitespace that put you over Fillwidth. It sets a flag indicating what the situation was. It then inserts the Newline and does the indenting. If that flag indicates the this is a reasonable situation (it was a word that put you over), it puts you back at the mark (which should be after the word that was wrapped) and inserts a space. If it was the spaces that put you over, it doesn't have to insert the space as the Newline and indentation already is the space.

3.3 Control-X Commands: File COMM3.C

MLstBuffs

Does the C-X C-B (List Buffers) command.

Note how it uses BSetMod and BScrnMrks to tell the redisplay what lines it bashed. Note also that it waits for the next character to be typed in before it allows the screen to be cleared.

MFindFile

Does the C-X C-F (Find File) command.

Yes, there is a call to BReadFile lurking in there. This command is shorthand for three different things and it shows, both in the complexity of the documentation and in the complexity of the code.

MDelMode

Does the C-X C-M (Delete Mode) command.

This routine, as with Add Mode, assumes that the mode list is kept packed and stored from the top down (large indices to small indices) in Buffs.Bmodes. Thus, [0 0 ModeA ModeB] is legal, while [0 ModeA 0 ModeB] and [ModeA ModeB 0 0] are not. These examples are number the indices from zero to three, from left to right.

Given that convention, Delete Mode's job is easy. It scans down the mode list until it finds a match, then packs the rest of the list.

3.4 Support Routines: File SUPPORT.C

```
FLAG
ArgEcho(targ)
     int targ
```

Uses the global cnt.

Waits an amount of time proportional to DELAY. If a character has been entered, returns FALSE. If not, prints "Arg: " and the argument Targ, and returns.

```
FLAG
Ask(mesg)
     char *mesg
```

Prints the message in the echo line. If the user responds with "Y", "y", or " ", returns TRUE. If the user responds with "N", "n", DEL (^?), or BS (^H) returns FALSE. Otherwise, it rings the bell and checks another character.

```
BlockMove(from,to)
     int from, to
```

Moves the (possibly zero length) block of characters between the Point and the mark From to the mark To. To is left after the moved characters. The Point is left at mark From. The Point is assumed to be before the mark From. If it is after, no characters are moved.

```
change(old,new)
     char *old, *new
```

Assumes that the Point is at the end of Old. It deletes Old and inserts New.

```
int
CheckMode(tmname)
     char *tmname
```

Returns the modeid of the indicated mode if tmname is a mode name, otherwise FALSE. The mode identifier is typically the first character of the mode name, e.g. "f" for Fill mode.

```
ClrEcho()
```

Clears the echo line.

```
CopyToMrk(tmark,forwdp)
     int tmark, forwdp
```

Imports the globals del_buff and lfunct.

Copies the region (the text between the Point and Tmark) into the delete buffer. If Forwdp is TRUE, it appends the text to the end of the delete buffer. Otherwise, it puts it at the beginning. It calls DelCmnd and if TRUE is returned, it saves what is in the delete buffer, otherwise the region replaces what is already there.

```
FLAG
Debug(message,value)
     char *message
     int value
```

Prints Message and Value in the echo area, calls redisplay to allow you to see what the buffer looks like, and waits for a character. Returns

TRUE if the character is DEL (^?) or BS (^H), FALSE otherwise.

```
FLAG
DelayPrompt(mesg)
     char *mesg
```

Uses the global cnt.

Waits for DELAY. If a character has been entered, returns FALSE. If not, prints the message in the echo line and returns TRUE. This routine is used to accomplish the delayed echoing of "Meta:", etc.

```
FLAG
DelCmnd(lfunct)
     int (*lfunct)()
```

Imports the global lfunct.

Returns TRUE if Lfunct points to a command that saves deleted text in the kill buffer, FALSE otherwise.

```
DoReplace(query)
     int query
```

Uses the global tmark.

Does the Replace String and Query Replace commands. Query is a flag that tells it whether or not to ask each time is locates an occurrence.

The routine is straightforward except in two cases. First, all characters not specifically checked for (the default) act as "No" answers. This is implemented in the code by not executing the remainder of the while loop because it executed the continue statement.

The other sticky point is in handling the ',' (Replace And Confirm) command. First, if it is a query replace, the routine sets the Tchar variable to ',' in order to cause the "Replacing" message to be printed. Each time ',' command is entered, you will be asked to confirm the replace and this message will have been bashed. Therefore, it is refreshed the next time through the loop and that same mechanism is used to get the whole mess going. The ',' command does the replace, so in that respect it is treated just like "Yes." However, after the change has been made, it must check to see if the command was ',' and, if so, ask for confirmation.

Note that the ',' option does not save the original string that was being replaced. Thus, if the "put it back" option is selected, the upper/lower caseness of the original string is lost.

```
Echo(mesg)
     char *mesg
```

Prints the message in the echo line.

```
error(mesg)
     char *mesg
```

Displays an error message. An error message appears off to the right in the echo line. Waits for a character to be typed before returning.

```
ForceCol(col,forwardp)
     int col, forwardp
```

This routine will force the Point to be in column Col. If necessary, it will insert spaces to put you there. Forwardp tells whether to round up or down on multi-column characters. (Due to multi-column characters, you are not always going to be in column Col. ForceCol merely deals with the interesting cases of Newline and Tab characters preventing you from being in the desired column.)

First, don't touch this routine. If you change anything, it probably won't work. This routine also relies heavily on the behavior of BMakeCol. BMakeCol leaves you in the proper column, if possible. If the proper column falls in the middle of a multi-column character (e.g., Tab or ^A), it leaves you after the character. If the line isn't long enough, it leaves you at the Newline.

ForceCol begins by doing a BMakeCol. That ensures that you are at least close to the desired position. It then checks to see if the desired column is less than or equal to zero. If it is less than, return. If it is equal to zero, we know that BMakeCol succeeded in leaving us in the correct column.

There are now three cases. First, Col can be after the Newline. It is detected by BGetCol < Col. In this case we want to insert some spaces. The second case is for Col to have fallen in the middle of what a Tab is tabbing over. It is detected by BGetCol > Col and the character before the Point is a Tab. In this case, it moves back one character and inserts some spaces. The third case is where Col is in the middle of a multi-column character (not Tab, though). In this case, it leaves you before or after the character depending upon the sense of Forwardp. There is an implicit fourth case, that in which you are already at the correct column. In this case, nothing must be done.

The purpose of the big hairy if statement on the third line is to leave you before the Tab if there is one (the second case above). The Indent on the following line then can handle both the first and second cases at one fell swoop. Note that if Indent is given a negative or zero argument, it does nothing. Thus, if you are already at the correct column or you are beyond it (third case), it does nothing.

The last if statement handles the third case. All other cases will have BGetCol == Col by now. All it does is leave you on the indicated side of a multi-column character.

```
FLAG
GetArg(mesg,term,str,len)
     char *mesg, term, *str
     int len
```

This routine does all of the work for accumulating a string argument (e.g., a search string or file name). It provides the full echoing and line editing facilities needed.

It prints the message and accumulates the response into Str. Str can have up to Len characters. When Term is typed, the routine returns. It returns TRUE if everything went all right, FALSE if C-G (Abort/Cancel Prefix) was entered.

```
int
GetModeId(msg)
     char *msg
```

Calls GetArg with Msg as the prompt and returns a Modeid of a valid mode or prints an error message and returns FALSE if a valid mode was not entered.

```
index(tstr,tchar)
     char *tstr, tchar
```

Returns the index of the first occurrence of Tchar in Tstr, or -1 if there is no occurrence.

```
FLAG
IsClose()
```

Returns TRUE if the character after the Point is a "closing" character, FALSE otherwise. A closing character is one of),], {, ", or '. Used by the sentence movement commands.

FLAG
IsGray()

Returns TRUE if the character after the Point is a "gray" character, FALSE otherwise. A gray character is a Space, a Tab, or a Newline. Note that IsGray includes Newlines, while IsWhite does not.

FLAG
IsNL()

Returns TRUE if the character after the Point is a Newline, FALSE otherwise.

FLAG
IsNLPunct()

Returns TRUE if the character after the Point is a Newline or punctuation, FALSE otherwise. Punctuation is either ".", "?", or "!".

FLAG
IsParaEnd()

Returns TRUE if the character after the Point is a Newline, Tab, "", or ".", FALSE otherwise. It is intended to be used to determine whether the Point is at the end of a paragraph. Paragraphs are delimited by Newline Newline, Newline Tab, Newline "" (Scribble commands), or Newline "." (most other text formatter commands). This routine assumes that you are between the two delimiters and it only checks the second one.

FLAG
IsSentEnd()

This one is a monster. You are assumed to be just after a candidate for an end of sentence (to wit, ".", "!", or "?"). This routine moves you over an arbitrary number of)}]'" characters and stops at the first character that isn't one of those. If that character is a grayspace character, it returns TRUE, otherwise it returns FALSE. Note that this will leave you at the grayspace or the other character.

FLAG
IsToken()

Returns TRUE if the character after the Point is a token character, FALSE otherwise. Token characters are alphabetics and digits.

FLAG
IsWhite()

Returns TRUE if the character after the Point is a whitespace character, FALSE otherwise. A whitespace character is either a Tab or a Space. Note that IsWhite does not include Newlines, but IsGray does.

itot(n)
unsigned n

Prints N on the terminal.

KbWait()

Waits for a character to be typed. It writes out modified pages (by calling BFlush) so long as no character has been typed.

KillToMrk(tmark,forwdp)
int tmark, forwdp

Deletes the region (between the Point and Tmark) and saves the deleted text in the delete buffer. Forwdp tells whether to put the deleted text at the beginning or the end of the delete buffer.

```
lowcase(str)
     char *str
```

Converts the string to lower case.

```
ModeFlags()
```

Uses the globals *buffs*, *cbuff*, and *lfunct*.

Prints the percentage, modified, and append next delete flags on the mode line.

```
MovePast(pred,forwdp)
     int (*pred)(), forwdp
```

Moves through the buffer, invoking Pred at each character. It stops when Pred returns FALSE, leaving the Point on the near side of the character which caused Pred to return FALSE. Forwdp tells whether to move forward or backward. Pred is a pointer to a function of no arguments.

```
MoveTo(pred,forwdp)
     int (*pred)(), forwdp
```

Moves through the buffer, invoking Pred at each character. It stops when Pred returns TRUE, leaving the Point on the near side of the character which caused Pred to return TRUE. Forwdp tells whether to move forward or backward. Pred is a pointer to a function of no arguments.

```
NLPrnt(str)
     char *str
```

Print Str at the terminal. Newline characters are printed as "<NL>".

```
FLAG
NLSrch()
```

Puts you after the next Newline or at the end of the buffer if there isn't one in that direction. Returns TRUE if it found one, FALSE otherwise.

```
Rebind(from,to)
     int (*from)(), (*to)()
```

Updates the global functs.

Changes all occurrences of the the function From in the bindings tables to To.

```
FLAG
RNLSrch()
```

Puts you before the previous Newline or at the beginning of the buffer if there isn't one in that direction. Returns TRUE if it found one. FALSE otherwise.

```
RubOut(ostr,str,tcol)
     char *str, *ostr, tcol
```

Performs DEL hackery for GetArg. It handles deleting multi-column characters from the screen.

```
SIndent(arg)
     int arg
```

Uses the global cnt.

Inserts Arg spaces. Does nothing if Arg is negative or zero.

```
strip(to,from)
     char *to, *from
```

From is a file name. This routine strips off the device ("a:") if it exists and the extension (".doc") if it exists and returns what's left in To. This routine does not change the case of anything.

```
FLAG
StrSrch(str,forwardp)
     char *str
     int forwardp
```

Uses the global cnt.

Does a string search in the direction indicated by Forwardp. Returns TRUE if it found the string, FALSE otherwise. Leaves you after the string or at the end of the buffer if it is a forward search, before the string or at the beginning of the buffer if it is a backward search.

```
ToBegLine()
```

Moves you to the beginning of the current line.

```
ToEndLine()
```

Moves you to the end of the current line.

```
TIndent(arg)
     int arg
```

Indents Arg columns with Tabs and spaces. It assumes that the Point was in column zero (or at least at a Tab stop).

```
ToNotWhite(forwardp)
     int forwardp
```

Moves you to the first non-whitespace in the direction indicated by Forwardp. You are not moved if you are already at non-whitespace.

```
ToSentEnd(forwardp)
     int forwardp
```

Moves you to the end of a sentence in the direction indicated by Forwardp. This routine does the work for Backward Sentence and Forward Sentence.

It first finds a potential sentence end (a punctuation mark or Newline). If the candidate is a Newline, it sees whether or not it is the end of a paragraph. If the candidate is a punctuation mark, it places a mark there, calls IsSentEnd (which moves you somewhere), and restores your position. It then checks to see if you are done.

```
ToWhite(forwardp)
     int forwardp
```

Moves you to the first whitespace in the direction indicated by Forwardp. You are not moved if you are already at whitespace.

```
ToWord()
```

Moves you forward to the beginning of a token (word). You are not moved if you are already at a word.

```
upcase(str)
     char *str
```

Chapter 4

The Terminal Abstraction

The terminal data abstraction is responsible for handling all of the interaction with the user's console. It provides a uniform interface to any terminal that Mince would ever see. This interface standardizes the calls for performing operations such as cursor positioning and clearing parts of the screen. It also provides for displaying characters in a uniform manner across all terminals.

The terminal abstraction performs these tasks by defining a "virtual terminal." By virtual terminal, we mean that the interface will always perform the desired function; it is up to the abstraction to make up for any missing or unusual features. It is the responsiblity of the terminal abstraction to ensure that the characteristics of this virtual terminal are faithfully reproduced on any physical terminal.

The virtual terminal has a screen that is a two dimensional array of characters. They are numbered with (0,0) in the upper left corner and (TMaxCol()-1,TMaxRow()-1) in the lower right corner of the screen. There are both a screen point and a cursor. The screen point is the (x,y) coordinate where the next modification to the screen will take place. The cursor is the (x,y) location where the visual marker is displayed. Note that these are not necessarily the same place: the screen point can be moved quite a bit while the cursor stays in the same place. (On most terminals, modifications must take place at the cursor. Thus, when an actual change is being made to the screen, the cursor must first be moved to that place.) The virtual terminal does not know about its own right edge. Thus, strange things can happen if, for example, a multi-column character is displayed so that it might wrap. (Mince's redisplay avoids this case.)

The keyboard for this virtual terminal has a buffer, typically eighty characters or so long. The terminal will only remember these characters (i.e., implement typeahead) if it is checked often enough. Thus, those calls to TKbChk which are liberally interspersed throughout Mince (and should be included in any changes that you make) perform a vital task.

The terminal abstraction is tailored for a specific terminal by running Config. Config stores the description in the swap file, which is in turn read by Mince as Mince comes up. A description of the place this information is read into and the format of the file can be found in Chapter One.

4.1 Initialization and Termination Routines

These globals are used throughout the terminal abstraction.

Private globals prow, pcol, srow, scol, clrcol, and kbdq.

The following routines initialize and terminate the terminal abstraction.

TInit()

Initialize the terminal abstraction and the terminal.

The routine first initializes the keyboard queue (buffer). It then sends the initialization string (as defined in the configuration program), sets the Clrcol array to the last column, and clears the screen (which sets them again).

TFini()

Restore the terminal to its original state.

Forces the cursor to be displayed where it "belongs" (at the bottom of the screen) and deinitializes the terminal by sending the string defined in the configuration program.

4.2 Cursor Positioning

```
int
TGetRow()
```

Returns the row that the screen point is on.

```
int
TGetCol()
```

Returns the column that the screen point is on.

```
int
TMaxRow()
```

Returns the number of rows on the screen. Row-1 is the number of the last row on the screen.

```
int
TMaxCol()
```

Returns the number of columns on the screen. Column-1 is the number of the last column on the screen.

```
TSetPoint(irow,icol)
     int irow, icol
```

Sets the screen point to (irow,icol).

TForce()

Forces the visual cursor to be displayed at the screen point.

Does nothing if the cursor is already there. If not, it uses the cursor positioning sequence to put it there.

4.3 Display Routines

TBell()

Rings the terminal bell or performs other alarm indications by sending the bell string.

TCLEOL()

Clears from the screen point to the end of the line.

Sends the clear to end of line string if there is one, otherwise it sends the correct number of blanks. It pays attention to and sets clrcol.

TClrLine()

Clears the line that the screen point is on. The screen point is left at the beginning of the line.

TCLEOW()

Clears from the screen point to the end of the screen.

Sends the clear to end of window string if there is one, otherwise it calls TCLEOL, then repeats going to the next row, setting the screen point column to zero, and calling TCLEOL until it gets to the end of the screen. It then restores the screen point to where it was.

TClrWind()

Clears the entire screen (window). The screen point is left at the beginning (home position) of the screen.

Sets the screen point to (0,0). Sends the clear window string if one exists, otherwise calls TCLEOW.

4.4 Printing Text

TPrntChar(ichar)
char ichar

Imports the global tabincr.

Prints the character at the screen point and updates the screen point by the display length of the character. Ordinary characters are printed normally. Control characters are printed as "^" followed by the character Ichar XOR 64 (e.g., C-A prints as ^A). Meta characters are printed as "~" followed by the character Ichar AND 127 (which may be a control charcter). Tabs (^I) are printed as the number of spaces remaining before the next tab stop (determined by Tabincr).

On ordinary characters, it puts the cursor at the screen point, sends the character, updates the cursor and screen point columns, and updates Clrcol.

On other characters, it prints them "in pieces" by calling itself recursively or, in the case of Newline, implements the meaning of the character by logically printing a Carriage Return and a Line Feed.

```
TPrntStr(string)
     char *string
```

Prints the string as if by repeated calls to TPrntChar.

```
TDisStr(row,col,string)
     int row, col
     char *string
```

Sets the screen point to (row,col) and prints the string as if by TPrntStr.

```
int
TWidth(colcnt,tchar)
     int colcnt
     char tchar
```

Imports the global tabincr.

Returns the display width of the character as it would be printed by TPrntChar with the screen point in column colcnt. If the character should be wrapped to the next line, the extra spaces needed to display the wrapping are included in Width. Note that the display width of a Newline is the negative of colcnt.

4.5 Low Level Output and Keyboard Drivers

```
TPutChar(ochar)
 char ochar
```

Sends a character to the terminal without any interpretation.

It actually does the output. It uses the bios if possible (based on the global outport.biosp). If it can't, it does the output manually, waiting for the port to be ready and then sending the character.

```
TKbChk()
```

Checks to see if there is a character available from the keyboard. If there is, it reads the character in and places the character in a FIFO queue. This routine should be called frequently in order to maintain proper typeahead buffering. If the queue is full, it rings the terminal bell.

It actually does the input. It checks the status, using the bios if possible. If there is a character ready, it reads it in, again using the bios if possible.

```
FLAG
TKbRdy()
```

Returns TRUE if there is a character avaiable, FALSE otherwise.

It calls TKbChk, then checks the queue to see if there is a character ready.

```
char
TGetKb()
```

Returns the next available character. It will wait if necessary for a character to become available.

Waits in a loop, calling TKbChk until there is a character ready, then grabs it.

4.6 Internal Routines

```
put_string(sdef)
     struct str *sdef
```

Puts a command string (e.g., the clear to end of line string) which is represented as a structure in the terminal abstraction internal format.

```
put_coord(firstp)
     int firstp
```

Sends a cursor coordinate to the terminal. The argument, Firstp, indicates WHICH coordinate to send; the coordinates themselves are stored as globals (Prow and Pcol). If Firstp is the same sense as Terminal.Rowfirstp (i.e., they are both TRUE or both FALSE), the row is sent, otherwise, the column is sent. In either case, the coordinate is biased as necessary (Terminal.Rowbias or Terminal.Colbias). If the coordinate is sent in binary, it will optionally complement it (Terminal.Compp).

```
put_num(num)
     unsigned num
```

Sends a number to the terminal as ASCII digits.

If the number is greater than ten, it calls itself recursively to print out the first n-1 digits. It then prints the nth digit. Note that it calls TPutChar while itot (in SUPPORT.C) which does a similar thing calls TPrntChr. More often than not, this routine is called by TPrntChr in the course of doing a cursor positioning sequence.

About This Chapter

This chapter was originally written and submitted as a B.S. thesis at the Massachusetts Institute of Technology. Its goal was to provide a discussion of considerations of implementing a text editor.

The preceding chapters of the Mince Internal Documentation discuss the details of a specific implementation; this chapter will help provide perspective about what considerations are general and what ones are implementation specific.

Acknowledgements

I would like to thank Owen Ted Anderson for teaching me a lot of what I know about editors as well as writing one of the most readable programs around.

I would like to thank Bernard S. Greenberg, who supplied some of the algorithms which are presented here.

I would like to thank Richard M. Stallman and the rest of the M.I.T. Artificial Intelligence Laboratory for creating the original Emacs and for doing most of the development of ITS EMACS.

1. Introduction

This thesis is intended to answer the question, "What are the important considerations in designing a text editor?" In answering this question, it will provide a reference document for would-be implementors of text editors.

There is a modest amount of literature available which discusses topics related to text editing. Most of the papers are "reference manual"-like because they explain the user interface only. A few of the rest cover the details of a specific implementation of an editor. This thesis will generalize the latter into a document which considers the problems relevant for all text editors.

The primary goal of a text editor is to allow the user to edit text. There are two secondary goals. First is to perform this editing without wasting resources. Second is to give the user a pleasant environment to edit in. The latter requires a good command set, feedback to the users, and quick response to commands.

Achieving these goals is hard. One way to make it easier is to break the design of the editor into three parts. The memory management part performs efficient editing of the text. It is essentially a very simple editor in itself. The incremental redisplay part provides feedback to the user. The command set (loop) part translates the user's input into commands to the memory management part. Each part of the structure contributes in its own way towards providing quick response. It is this structure that will be discussed in this thesis. Each chapter of the thesis covers a different part.

The second chapter is memory management (you are reading the first chapter). The basic problem that is addressed is: given that you have a possibly large buffer, how do you structure the storage for it so that trivial operations (e.g., inserting a character and moving around in the buffer) do not require excessive amounts of work? Other problems are: what should the interface to the buffer look like from a program? How do these considerations change when you have multiple buffers and/or virtual memory? In a nutshell, this chapter discusses the cpu time - memory - disk channel tradeoff. This topic is interrelated with the next one.

The third chapter is incremental redisplay. The basic problem here is: given that the user has a reasonable video terminal which you can communicate with over a limited bandwidth channel, how do you change what is displayed on his screen to match the current contents of the buffer? Other problems are: what are reasonable terminals to use? What extra information can you retain to speed up the updating process? This chapter discusses the cpu time - I/O channel usage tradeoff.

The fourth chapter is a discussion of the command loop. What is the basic edit cycle? What sort of errors do you have to recover from? How and why do you dynamically change the editor itself? What are some criteria to use when selecting an implementation language?

The fifth chapter considers user interface hardware. What are desirable ways for the user to interact with the editor? This area includes such things as desirable features in keyboards and how to take advantage of graphical input.

The sixth chapter mentions some other uses for text editors.

Note that MIT Emacs will be used in this thesis whenever a reference to a specific editor is required (for example, when discussing command syntax). This class of editors will be referred to as

"Emacs-type." A specific editor was selected (as opposed to creating another one) specifically to avoid the work of reinventing the wheel. MIT Emacs was selected because of the author's familiarity with it and because several implementations of it have been made, thus providing a wealth of experience with it in different environments.

2. Memory Management

A copy of the text that is being edited is stored in a buffer. The text appears to the user as a sequence of characters. All editing operations are specified relative to a place in the buffer. This place is called the point and it is always located between two characters (thinking this way eliminated the possibility of some fencepost errors). It is the responsibility of the memory management software to support buffers cleanly and efficiently.

It is assumed that the user will be presented with some sort of status display. This display will tell the user such things as the name of the buffer that he is editing, the name of the file that is being edited, and what modes the buffer is in (see section 4.5, page 37 for a discussion of what modes are). The interface to the memory management software includes operations to maintain this auxilliary information.

It is assumed that buffers are stored in the equivalent of main memory while the editing is being done. This means that the buffer is either in main memory (for very small machines), in the address space of the editor (for large address space virtual memory machines), or it can be mapped into the address space (for small address space virtual memory machines). Any of these cases will be assumed to be memory in this thesis. There are two commonly used techniques to manage memory in order perform the editing efficiently. These techniques are known as buffer gap (store the text as an array of characters) and linked line (store the text as a linked list of lines) and will be discussed in following sections. Their discussion follows the more theoretical sections which cover the definition of the interface between the main editor and the memory management routines. Further discussion shows how the two schemes perform in a virtual memory environment and when multiple buffers are manipulated. Some closing remarks will be made about scratchpad memory and methods of reclaiming storage.

2.1 Data Structures

This section discusses the data needs of an editor. With two exceptions, all of the state of the editor is defined here. Thus, if this information is retained across invocations, you will have the ability to resume editing where you left off. Thus, the amount of work involved with editing can be reduced.

The other place where state information is kept is in the the screen manager. The screen manager to retains a knowledge of how buffers were displayed. Retaining this information allows the screen to reappear as the user left it. If the information is not retained, the screen manager will have to recalculate the display and this can be somewhat confusing. However, the editor will not lose any functionality if this state is left out.

The World contains the buffers in use by the editor. It is a circular list of BufferDescriptors and an indication of which buffer is the current one. In a PL/1-ish syntax:

```
declare 1 World,
          2 CurrentBuffer      pointer,
          2 BufferChain pointer;
```

Each buffer descriptor has several types of internal information.

```
declare 1 BufferDescriptor,
          2 NextChainEntry          pointer,
          2 BufferName              char(big) varying,

          2 Point                   location,
          2 Length                  fixed,
          2 Modifiedp               bit(1),

          2 FileName                char(big) varying,
          2 ModeName                char(big) varying,

          2 MarkList                pointer,
          2 ModeList                pointer,

          2 StorageData             pointer,
          2 ScreenData              pointer;
```

NextChainEntry is a mechanism for implementing the circular list of buffers. The list is circular because there is no preferred buffer and it should be possible to get to any buffer with equal ease. BufferName is a way for the user to be able to refer to the buffer.

Point is the current location where editing operations are taking place. It is of the data type location. The representation for this data type is implementation specific. For buffer gap editors, it is an integer, but for linked line editors, it is a (line pointer, offset) pair. Length indicates how long the buffer is in some reasonable unit (usually characters). Modifiedp is a flag which indicates whether the buffer has been modified since it was last written out or read in.

FileName is the name of the file system object which is currently associated with the contents of the buffer. ModeName is way to tell the user what modes are in effect. Typically, each mode will insert its name there as it is invoked. This information is not really implicit in the ModeList because there can be invisible modes (for example, autoloaded commands) which use the mode mechanism for invocation but the user does not want to be made explicitly aware of them.

MarkList is simply a list of marks.

```
declare 1 Mark,
          2 NextMark     pointer,
          2 Name         {anything convenient, try small
                          integers},
          2 WhereItIs    location;
```

Each mark has a pointer to the next one, a name, and a location within the buffer. Note that this list is not circular and it would probably help to keep it sorted by increasing location. The name is a way of distinguishing this mark from any other one associated with this buffer. This name is generated by the Create Mark routine and returned. It can thus be any convenient data type. Small integers will work quite well.

ModeList is a list of procedures to be invoked when this buffer is selected. See section 4.5, page 37 for a more complete discussion.

StorageData is a descriptor block which defines how the contents of the buffer are stored in memory. The nature of this block is dependent upon the memory management algorithm used. ScreenData is a descriptor block which defines how the buffer appears on the user's screen. Its definition will become apparent in the discussion in the next chapter.

2.2 Marks

A mark is a named fixed point within a given buffer. A mark always points between the same two characters no matter what has been inserted or deleted around it. Marks are used for several different reasons.

- They remember a specific location for future reference. For example, a command might paginate a file. In this case, a mark would remember where the point was so that the command could return with the location of the point unchanged.

- They delimit a portion of text in conjunction with other marks or, more commonly, the point. This portion of text is called the region. This case would be used, for example, in a DeleteRegion command.

- They serve as bounds for iteration. Because they remain invariant when changes are made to the buffer, they can serve as a constant position to "head towards." An example could be the FillParagraph command. This command iterates through the buffer deleting and inserting whitespace (in the process, making each line as long as it can be without going past the right margin) until it reaches the end of the paragraph. A mark is used to remember where the end of the paragraph is. This usage is a variation on the region-delimiting usage, but it is worth noting in itself.

When an insertion is made at a mark there is a question about what to do with the mark (i.e. on which side of the inserted character it should end up). For the most part, the mark should move (i.e. be after the inserted character). However, there are good reasons for having it work the other way and so there are fixed marks, which remain before an inserted character.

An example of using fixed marks is to delimit an insertion. A routine could create both a mark and a fixed mark at the same location. Any inserted text would push the marks apart and end up between them. Thus, it is possible to keep track of what has been inserted.

2.3 Interface Procedures

This section defines the interface between the main part of the editor and the memory management routines. They will be described in terms of their logical function only, leaving out specific implementation details. An example of such a detail is a code variable which is returned and which indicates whether the operation succeeded. Also note that any data types mentioned (e.g. string) are intended to be canonical and no specific implementations are assumed. A <r> after a parameter means that it is returned by the procedure.

There is an important question as to exactly who allocates the data (the buffer descriptors and the buffers themselves). This issue is more language specific in the sense that certain languages specify an answer which must be used whether or not it is the right one. The procedures will be defined as if they own the data. If it is decided that they do not, it is relatively easy to include an extra argument on each procedure call which identifies a descriptor of the object that the procedures are to manipulate.

```
InitWorld
SaveWorld(FileName)
LoadWorld(FileName)
```

InitWorld is the basic set-up-housekeeping call. It is called once, upon editor invocation. SaveWorld and LoadWorld implement the state-saving across edit sessions. SaveWorld is used to save the state of the editing session for later resumption. This operation might be quite expensive if it requires explicitly writing out all of the buffers to a large file or it might be very cheap in a virtual memory environment, where all that might be required is to set an external static variable to indicate that the environment is consistent. The possibility of multiple saved environments is interesting, but has not been implemented to my knowledge. It seems to be a nice way to work on several of tasks (not in the process management sense) at once.

If you are creating a "stripped down" editor then the save and load world routines will not do anything. They can be put in as stubs if there is a reasonable possibility that the editor will be embellished later.

```
CreateBuffer(BufferName)
DeleteBuffer(BufferName)
SetCurrentBuffer(BufferName)
SetCurrentBufferNext(BufferName <r>)
```

CreateBuffer is given a name and it returns after having created a buffer of that name. If the buffer already exists, it probably should signal an error of some sort to keep from bashing existing information. DeleteBuffer deletes a buffer. If the current buffer is deleted, the default buffer becomes the current one. (The editor is created with one default buffer called "Main" or something like that. It must always exist.)

Depending on the implementation language, we may be able to chosse in the procedures that are being defined between including a buffer as an explicit parameter or having it implicit by setting an own variable that indicates which buffer is the current buffer. If buffers are changed often, it is worthwhile to include the buffer with each call. If, on the other hand, buffer switching is done infrequently, the overhead involved with setting a current buffer is more than acceptable. My experience has been that buffer switching occurs only rarely and so the extra call is worth it.

SetCurrentBuffer makes buffer BufferName the current one. SetCurrentBufferNext makes the next buffer in the circular list the current one and RETURNS its name in BufferName. This mechanism allows for iterating through all buffers looking for one which meets an arbitrary test.

Note that most of the above calls are really useful only if you have a multiple buffer implementation of the editor. In a single buffer editor, they are relatively useless and should be used only if there is a reasonable chance of expanding to a multiple buffer editor in the future.

```
SetModified(Flag)
GetModified(Flag <r>)
SetPointA(Location)
SetPointR(Count)
```

```
GetPoint(Location <r>)
GetLength(Size <r>)
```

These routines deal with several variables. They allow setting and asking for the point, the current buffer length and the state of the modified flag.

The modified flag provides an indication of whether the buffer has changed. It is set implicitly by any buffer change operation (principally insertion and deletion) and cleared automatically by writing the contents of the buffer to a file. The procedures to set or clear it explicitly are provided as this ability will ordinarily be used by the redisplay code (see section 3.6.2, page 27 for the discussion of what it is used for).

Note that there are two flavors of the SetPoint routine, designated "A" and "R". They both do the same logical operation, but the "A" version interprets its argument as an absolute position within the buffer and the "R" version interprets its argument as an offset relative to the current position of the point. (Negative values indicate a backward offset.) Due to the definition of the location type, the "A" version does not take an integer value as the location and so one usage is not readily simulatable in terms of the other.

```
Insert(String)
Delete(Count)
GetStringA(Location,Length,String <r>)
GetStringR(Count,Length,String <r>)
```

These routines manipulate and examine the buffer. Insert inserts a string into the buffer at the point. The point is left at the end of the inserted string. Delete removes abs(Count) characters from the buffer. (Negative counts delete before the point).

GetString returns the string starting at the specified location and Length characters long. There are both absolute and relative versions of this routine.

```
Search(String,Location <r>,Flag <r>)            F | B    A | R
FindFirstIn(String,Location <r>,Flag <r>)       F | B    A | R
FindFirstNotIn(String,Location <r>,Flag <r>)F   | B      A | R
LookingAtP(String,Location,Flag <r>)                     A | R
```

There are a total of fourteen routines in this section, but they have been listed in an abbreviated form for convenience. These are search routines and each form can head either forward (F) or backward (B) and with the returned location either absolute (A) or relative (R) to the point.

Note that these routines are not necessary as their action can be readily simulated by using the other defined routines. However, they have been included in the discussion because they are useful and because they are often implemented in the same level as the other memory management routines for having lower level access to the buffer will speed up their execution.

Search looks for the first occurrence of the string in the buffer, starting at the current point and heading in the specified direction. It returns a flag saying whether the string was found and an indication of the string's location. The location returned is the location of the end of the found string (either absolute or relative) in the direction of the search. (For a backward search, the location is the beginning of the actual string.) This definition implies that repeated searches will stop at successive instances of the string.

FindFirstIn searches for the first occurrence of any of the characters in the string. For example, FindFirstIn("0123456789",...) would return the location of the first digit encountered. Like Search, it returns a flag as to whether a match was made and the location of the match. Unlike Search, the location returned is at the beginning of the match and not the end. Successive applications will thus return the same position. The difference in behavior between Search and FindFirstIn is a function of their different uses. FindFirstIn is used to parse through text slowly, and zero and one characters strings must be handled properly. These definititons facilitate that handling. FindFirstNotIn works in exactly the same manner as FindFirstIn except that it matches on any character not in the string. For example, the following code fragment implements a forward word operation.

```
alphabet="abcdefghijklmnopqrstuvwxyz";
FindFirstInFA(alphabet,location,flag);
SetPointA(location);
FindFirstNotInFA(alphabet,location,flag);
```

The first Find operation will skip over any non-word characters to the beginning of a word. The next one will skip to the end of the word. (Note that the alphabet variable should also have the digits, uppercase, and several special characters in order to work as one would intuitively expect. Note also that the selection of special characters will in general be language-specific. Further, no checks were made for string-not-found, etc. Thus, it is not an example of finished code.).

An alternative way of defining these operations is to have them automatically set the point instead of returning a location. (The flag must still be returned.) If the string was not found, the point would not be moved. The choice of a method of implementing these routines is a matter of taste.

LookingAtP has a much more simple defintion. It compares String against the sequence of characters in the buffer starting at Location, returning the true/false answer in Flag.

```
GetHpos(Column <r>)
SetHpos(Column)
```

(These routines, like the previous set, are not necessary but useful.) GetHpos returns the column that the point is in, after taking into account tab stops, etc. It does not take into account the screen width as it should not make any difference to the edi r how big the terminal is. SetHpos moves the point to the desired column, stopping at the end of a line if it is not long enough. If there is no character at the desired column (due to tab stops), it uses the next higher available column position.

```
SetFileName(FileName)
GetFileName(FileName <r>)
WriteBuffer
ReadBuffer
```

These routines interface between the buffer and the file system. The FileName routines set and return the file object (in general a string--the file name). At the user interface, the editor might implement an intelligent "default and guess" interpretation of the file name so as to make life easier for the user, but doing so does not affect this level of code. This general area is one where system-specific conventions become significant.

WriteBuffer writes the contents of the buffer out to the file name associated with the buffer. Any conversions between internal format and what the file system requires will be done at this time. Also, the buffer modified flag (ModifiedP) will be cleared.

ReadBuffer reads the file into the current buffer. There are two choices about how to do the read operation. Both directions will be discussed along with their ramifications for the other parts of the editor. They are not both implemented because it is desirable to keep the number of primitives to a minimum.

First, it can replace the contents of the buffer with the contents of the file. If it does so, the buffer modified flag will be cleared automatically. The editor will want to check on what it is replacing. If the previous contents of the buffer have been modified, the user should be asked what to do (e.g., whether he is making a mistake).

Second, it can insert the contents of the file into the contents of the buffer at the point. In this case, the first method can be simulated by explicitly deleting all of the buffer and then reading. The buffer modified flag will have to be manually cleared. The same policy of asking the user what to do with modified buffers should be followed. The advantage behind this method is that it allows the easy implementation of the insert file command. The first method requires the allocation of additional space and then the copying of the data; a luxury that may not be available on smaller systems. This second method is thus preferred.

```
CreateMark(MarkID)
CreateFixedMark(MarkID)
DeleteMark(MarkID)
SetMark(MarkID,Location)
GetMark(MarkID,Location <r>)
CompareLocation(Location1,Location2,Result <r>)
```

These routines manage marks. They allow for creating both ordinary and fixed marks, deleting marks, and setting and evaluating them. Note that except for creating them, there is no difference in usage with these routines between ordinary and fixed marks (although their behavior will, of course, differ).

SetMark merely sets the location of the mark to Location. (A relative version of this routine can be supplied, if desired.) GetMark returns the current location of the mark. It should be used directly and not assigned into a variable as its value can change across some buffer operations. These operations are Insert, Delete and ReadBuffer.

CompareLocation allows the comparison of any two marks or the point and the mark to be done without being aware of the specific scheme chosen. It takes two Locations as arguments and returns the sign (+1, 0, -1) of the result of Location1 - Location2.

```
SetModeName(ModeName)
GetModeName(ModeName <r>)
AppendModeList(Procedure)
DeleteModeList(Procedure)
InvokeModeList
```

These routines manage the multiple mode capability. The ModeName is a string which can be displayed to remind the user what is going on. It does not affect anything else.

Append, delete, and invoke operations are all supplied. It is generally bad form to define the modes so that it matters in which order the procedures are called, but there do arise such occasions. Therefore, the procedures should be called in the order that they are appended onto the list. Checks should be made to insure that a procedure is not put on the list more than once. Again, see section 4.5, page 37 for a complete discussion of modes.

2.4 Buffer Gap

This section discusses the implementation of one of the two ways of implementing the memory management functions.

A buffer gap system stores the text as two contiguous sequences of characters with a (possibly null) gap between them. It thus uses memory efficiently as the gap can be kept small and so a very high percentage of memory can be devoted to actually storing text. Changes are made to the buffer by first moving the gap to the location to be changed and then inserting or deleting characters by changing pointers.

In more detail, here is an example buffer which contains the word "Massachusetts".

```
0   1   2   3   4   5   6   7   8               9  10  11  12  13
| M | a | s | s | a | c | h | u |   |   |   | s | e | t | t | s |
-----------------------------------------------------------------
0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
 30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45
                    P          GS          GE
```

There is a lot of information here which needs explaining. First, the buffer is 13 characters long and it contains no spaces. The blanks between the "u" and the "s" show where the gap is and do not indicate that the memory has spaces stored in it. The point is between the "a" and the "c" at location 5 and is labeled with a "P" in the bottom line (legal values for the point are the numbers from zero to the length of the buffer). There are also three different sets of numbers (coordinate systems) for referring to the contents of the buffer.

First is the user coordinate system. It is displayed above the buffer. The values for it run from 0 to the length of the buffer. As you will note, the gap is "invisible" in this system. The coordinates label the positions between the characters and not the characters themselves. Thought of in this way, the arithmetic is easy. Thought of as labeling the characters, the arithmetic becomes fraught with special cases and ripe for fencepost errors.

Second is the gap coordinate system. It is displayed immediately under the dashed line. The values for it run from 0 to the amount of storage that is available and it, too, labels the positions between the characters (or rather, storage cells). The internal arithmetic of the buffer manager is done in this coordinate system. The start of the gap (labeled "GS" in the bottom line) is at position 8 and the end of the gap (labeled "GE") is at position 11.

Conversion from the user coordinate system to the gap coordinate system is quite easy. If the location (in the user coordinate system) is before the start of the gap, the values are the same. If the location is after the start of the gap (NOT the end of the gap!), the location in the gap coordinate system is (GapEnd - GapStart) + the location in the user coordinate system. It is a good idea to isolate this calculation either in a macro or a subroutine in order to enhance readability. Most routines (e.g. Search) will then use the user coordinate system even though they are essentially internal.

The third coordinate system is the storage coordinate system. It is the bottom row of numbers in the diagram. It is the means whereby the underlying storage cells are referenced. It is labeled from X to X + the amount of storage that is available. The origin (the value of X) 30 was chosen to be 30 here to help distinguish between the various coordinate systems. Its absolute value makes no difference. Note that it labels the cells themselves and so caution must be taken to avoid fencepost errors.

A buffer gap system has a very low overhead for examining the buffer. The reference (GetChar) comes in in the user coordinate system and the location is converted to the gap coordinate system. The cell is the looked up and the contents returned. Essentially, one compare and a few additions are required. The purpose of the conversions is to make the gap invisible. Note that in no case is any motion of the buffer necessary.

There is more of an overhead associated with inserting or deleting a character. In this case, the gap must be moved so as to be at the point. There are three cases:

1. The gap is at the point already. No motion is necessary.

2. The gap is before the point. The gap must be moved to the point. The characters after the gap but before the point must be moved. Thus, ConvertUserToGap(Point) - GapEnd characters must be moved. This quantity is numerically point - GapStart.

3. The gap is after the point. The gap must be moved to the point. The characters after the point but before the gap must be moved. Thus, GapStart - ConvertUserToGap(Point) characters must be moved. This quantity is numerically GapStart - point.

After the gap has been moved to the point, insertions or deletions can be effected by moving the GapStart pointer (or the GapEnd pointer--it makes no difference). A deletion is a decrementing of the GapStart pointer. An insertion is an incrementing of the GapStart pointer followed by placing the inserted character in the storage cell.

Note that after the first insertion or deletion, further such operations can take place with no motion of the gap (it is already in the right place). Further, the point can be moved away and back again with no motion of the gap taking place. Thus, the gap is only moved when an insertion or deletion is about to take place and the last modification was at a different buffer location.

This scheme has a penalty associated with it. The gap does not move very often, but potentially very large amounts of text may have to be shuffled. If a modification is made at the end of a buffer and then one is made at the beginning, the entire contents of the buffer must be moved. (Note, on the other hand, that if a modification is made at the end of a buffer, the beginning is examined, and another modification is made at the end, no motion takes place.) The key question that must be asked when considering this scheme is, when a modification is about to be made, how far has the point moved since the last modification?

> Sidenote Calculation. How far can the point be moved before the shuffling delay becomes noticeable? Assume 1/10 sec. is noticeable and that it is a dedicated system. Assume 1usec, 8-bit wide memory. Assume 10 memory cycles per byte moved (load, store, eight overhead cycles for instructions). Then, 10,000 bytes can be moved with a just noticeable delay.
>
> Because of the locality principle, it seems reasonable to conclude that for almost any rational buffer size the average distance moved will be less than 10K bytes and so the shuffling delay will not be noticeable.

2.4.1 Gap Size

Note that the size of the gap does not affect how long the shuffling will take and so it should be as large as it can be. Typically, it is all of the otherwise unused memory. In that case, when the gap size goes to zero, there is no more room to store text and the buffer is full.

2.4.2 Multiple Gaps and Why They Don't Work

Assume that we were still uncomfortable with the shuffling delay and a possible fix was put forth. This fix would be to have, say, ten different gaps spread throughout the buffer. What would the effects be? The idea behind this discussion is to help in understanding the buffer gap system by seeing how it fails.

First, the conversion from the user to the gap coordinate system would be more complicated and take longer. Thus, some ground has been lost. However, this is a small loss on every reference in order to smooth out some large bumps, so it might still be a reasonable thing to do.

Second, the average amount of shuffling will go down, but not by anywhere near a factor of ten. Because of the locality principle, a high percentage of the shuffling is of only a short distance and so cutting out the "long shots" will not have a large effect.

Third, unless the writer is very careful, the gaps will tend to lump together into a fewer number of "larger" gaps. In other words, two or more gaps will meet with the GapEnd pointer for one matching the GapStart pointer for another. There is just as much overhead in referencing them, but the average amount of shuffling will increase.

On the whole, the extra complexity does not seem to return proportional benefits and so this scheme is not used.

2.4.3 The Hidden Second Gap

On two-dimensional memory systems such as Multics, a second gap at the end of the buffer is provided with almost no extra overhead. The key to this gain is that the buffer is not stored in a fixed-size place. Rather, the size of the memory that is holding the buffer can also increase.

The extra overhead is a check to see whether a modification is taking place at the end of the buffer. If so, the modification is made directly with the EndOfAvailableStorage (the buffer runs from X to X + EndOfAvailableStorage) variable serving to note that the change has taken place.

This change has more of an effect that might at first be apparent because a disproportionately high percentage of modifications take place at the end of the buffer. This distortion is due to the fact that most documents, programs, etc. are written from beginning to end and so the new text is inserted at the end of the buffer.

The overhead for this change is low because the check for the end of the buffer was already there. There is no problem of the gaps coalescing because one of them is pegged into place. The gains are not all that great, but neither are the costs and so it is used. This technique is also usable with some implementations of multiple buffers.

2.5 Linked Line

The other method of memory management that we will discuss is called linked line. It stores the buffer as a doubly linked list of lines. This method is especially useful with languages such as Lisp which provide memory management facilities integral with the language.

Each line in the linked list has several pieces of information in its header. Not all of these pieces are required, but they can help greatly in managing the buffer. The pieces of information are:

```
NextLine        pointer         /*32 bits*/
PreviousLine    pointer         /*32 bits*/
Length          fixed           /*16 bits*/
/*presumably no SINGLE line will be >64K characters*/
Line            char            /*the line itself*/
```

optional fields:

```
AllocatedLength fixed           /*16 bits*/
Version         fixed           /*32 bits*/
Marks           pointer         /*32 bits*/
TextPtr         pointer         /*32 bits*/
```

The NextLine and PreviousLine fields implement the doubly linked list. The length field is, clearly, the number of characters in the line. These, along with the line itself, are all that are required in order to implement the linked line scheme. The other fields are a help in making the scheme efficient and some of them are very valuable to include.

The AllocatedLength field indicates how much memory is allocated to storing the line itself. Thus, an allocate/free combination are not required each time a character is inserted or deleted. For example, a memory allocation block size of 16 bytes has been used in some implementations of this scheme. AllocatedLength will then be either 0, 16, 32, 48, 64, etc. The allocate/free combination is only required every time the line crosses a 16 byte boundary, a considerable savings in overhead.

Allocating memory in 16 byte chunks cuts down significantly on fragmentation. It will almost certainly be possible to run without a compactifying garbage collector. See the discussion of scratchpad memory (section 2.9, page 17) for further information.

The version field is for use by the redisplay code and is an optimization to make it run faster. It will be discussed with the rest of the redisplay process. It serves the purpose of specifying a unique id for the line.

Using integer-valued buffer positions is hard with the linked line scheme. Instead, a (line pointer, offset) pair are used. Marks are then always associated with a line and can thus be merely strung in a list associated with the line that they are on. With this implementation, less time is required to update the marks because only those that are on the line can possibly be changed. Note that there should still be a central listing of all marks in order to facilitate finding any given one and that mark ids should be unique within a buffer.

Finally, instead of storing the text of the line with the header, it can be separately allocated. The TextPtr field is then used to remember where the text is. This ability is especially useful when several places point to the header and properly updating them whenever the line is reallocated is difficult.

In summary, the most useful fields are NextLine, PreviousLine, length, AllocatedLength, and either version or the mark list. These fields can fit within one 16 byte allocation block.

The operation of a linked line scheme is quite straightforward. New lines, when created, are simply spliced into the list at the appropriate place. (Note that no characters are stored to indicate line breaks). If the new line is in the middle of an existing line, some movement of the text on the end of the old line to the new line is all that is required.

The line itself is stored as a packed array of characters. Inserting or deleting text is done by scrolling the line after the point of modification. Clearly, this scheme is very inefficient with large line lengths.

The reason why the length fields were 16 bits long is not obvious. After all, only rarely will a document have even 256 character lines. But people occasionally edit rather strange things, including object files. One cannot rely on encountering new line characters at reasonable intervals in such files. Thus, the extra size.

2.5.1 Storage Comparison

Storage requirements for a linked line scheme are somewhat higher than for buffer gap. A buffer gap scheme requires one or two new line characters per line, and a small amount of fixed storage (GapStart, GapEnd, etc.).

Linked line requires, in a reasonable implementation, one 16 byte block plus an average of 8 bytes lost due to fragmentation for each line. On the other hand, large amounts of text will never have to be moved.

2.5.2 Error Recovery Comparison

Recovering from errors (an unexpected program termination, for example) is relatively easy and fail soft in a buffer gap. In general, the start and end of the buffer are findable if a marker is left around the buffer (say, a string of sixteen strange (value 255) bytes) and the buffer is everything between them. The gap can be recovered and manually deleted by the user or, if it, too, is filled with a special marker, it can be automatically deleted.

Linked line management is harder to recover. Recovery is greatly aided by erasing freed memory. Basically, you pick a block at random and examine it. If it can be parsed into a header (i.e., the pointer values, etc., are reasonable), continue (a careful selection of header formats will help). Otherwise, pick a different block. You can then follow the next and previous pointers and parse them. If this works three or four times in a row, you can be confident that you have a handle on the contents. If a header doesn't parse, it is because it is either a part of a line (either pick again at random or go back one chunk and try again) or a header that was being modified (in which case you are blocked from continuing down that end of the chain). In the latter case, go in the other direction as far as possible. You now have one half of the buffer. Repeat the random guess, but don't pick from memory you already know about. You should get the other half of the buffer. Leave it to the user to put them together again. If the freed blocks are not erased, the chance of finding a valid-looking header that points to erroneous data is very high.

2.6 Multiple Buffers

How do buffer gap and linked line schemes implement multiple buffers? There is a variety of choices:

```
intertwining (linked line only)
separate storage for each buffer
        large address space (therefore paged)
                structured
                non structured
        small address space
                special cases
```

Intertwining is an option that is only open to linked line. In this case, all allocation is done out of a common pool and so, over time, the buffers tend to "intertwine" (i.e., the lines of one buffer are mixed in with the lines from other buffers in physical memory). Such an approach tends to maximize the density of text and thus make the most efficient use of memory. It also assumes that a large address space is available. (See also the discussion in the next section about paged environments.)

Separate buffer space means that each buffer is allocated out of its own area and that all of a buffer's area is contiguous. Thus, the address space is cut up into separate sections for each buffer.

If a large address space is available, the cutting up can be done one of two ways. If the address space is structured (as in Multics), the operating system takes care of managing such things automatically. If the address space is not structured (as in Vax/VMS), the memory management scheme can reserve fixed regions of the address space for separate buffers, each more than large enough for any reasonable file.

If the address space available is too small to reserve effectively, the memory management scheme will have to keep track of all of the buffers and map them into and out of the available address space as needed. Caution must be taken to avoid requiring that only one buffer be in the address space, as a multiple window editor must be able to scan multiple buffers. In addition, auxilliary buffers will be needed from time to time (e.g., for copying tet from one to another).

Managing multiple buffers is relatively easy. They are treated as a set of buffers, only one of which can be accessed at a time. See the earlier section on buffer data structures (section 2.1, page 3).

2.7 Paged Virtual Memory

How well do the two schemes perform in a paged virtual memory environment?

The buffer gap scheme works very well in general. Its highly compact format allows for accessing large parts of the buffer with only a few pages in memory. Its sequential organization also implies that it has a very good locality of reference and so the nearby pages are heavily referenced and likely to be around.

Its major problem is the large amount of shuffling that must be done in some cases. A move of the whole buffer implies that the whole buffer must be swapped in and--most likely--swapped out again. (A search of the whole buffer also requires this swapping, but the user asked for it and no management scheme can search linearly through memory that is on disk. Therefore, the user should expect lesser response.) If the memory manager is built into the operating system, some interesting hackery can be done with the page table to "move" all of a chunk of memory by one or more pages by moving page table entries. The existence of this example implies that such a function might well be desirable to include in a future set of operating system calls ("insert n pages after page x and scroll through page y"--delete n pages is implicit in this and it only affects part of the address space).

In a tight memory situation, the buffer gap scheme does as well as can be expected. The nearby sections of the buffer will be around because of locality of reference, but anything far away can take a while to get to.

A linked line scheme does not perform as well overall. First, if an intertwining multiple buffer scheme is used, one may as well forget performance in a tight memory situation. The intertwining can use different parts of each page for storing different buffers. Thus, when considering any given buffer, the page size is effectively reduced.

Even in a separated buffer scheme, the data is not as tightly packed overall (the headers and fragmentation) and so some performance is lost. Also, the linked lines can be anywhere in a large portion of memory and so the density of nearby lines can range from good to very low. Finally, even if a desired line happens to be on an in-memory page, in order to get there (via the links), you will probably have to swap in several additional pages and, in the process, may even swap the desired page out!

The primary advantage that linked line has is that it never requires moving large sections of the buffer. Thus, if memory is not tight, the entire buffer can fit in memory and performance will be very good.

2.8 Editing Extremely Large Files

Extremely large files come in two flavors. First are files that are so large that reasonable assumptions break down. Such things tend to start happening about 64M bytes or so. At that point, even simple things (e.g. string search) tend to take several minutes to run on a fast processor with the whole file in memory.

Although there are one or two interesting hacks to stay alive, life is simply not bearable when trying to edit such a large unstructured file. The alternative (which large data base people have known about for years) is to structure the file. This alternative is not that unpalatable because an unstructured editor can still be used to edit the subpieces of structure. The other reason why this is not that much of a problem is that there aren't all that many gigantic files to edit. The vast majority of files are much smaller. Gigantic files call for special tools for manipulating them.

The other flavor is more applicable to microprocessors where an extremely large file might be 100K bytes. The reason why it is considered so large is that the disk to store it on might be only 50K bytes, or there might only be magnetic tape for permanent storage. Thus, a 100K file would tax the hardware resources severely.

The basic way of dealing with such files is to break them up into chunks and edit the chunks separately (the TECO yank command is an example of this). In general, you can only proceed forward through the file in any given edit session because of the problems involved with the file size changing as the edit progresses. Either a marker byte (↑L is commonly used) or a character count (not as polite to the user) can be used to determine where the file breaks are to occur. This method requires an input and output file to both be available and open at the same time. A crash preserves the input file and some of the output file. Thus, editing a 100K file requires up to 200K of storage. This is the only method that works on magnetic tape.

The next method allows full access to the file without breaking it up in any way. It requires three files (input, output, and backup) to be open simultaneously. As you proceed through the file, it edits from the input to the output file. However, when you reverse direction, it reads fromthe output file onto the third, backup file (it does not modify the input file, thus ensuring its integrity in the case of a system crash). Note that the data is stored in the backup file in reverse order! Preferably, file i/o is done in blocks and only the order of the blocks needs to be reversed, not the contents of the blocks themselves. When you switch to going forward once again, the backup file is read until it is exhausted and then use of the input file.is resumed. This method allows for simulating a very large buffer as the file management can be done invisibly. Thus, the user can edit a 100K file with much less physical memory. Note that the swapping can be slow!

The final method that is available is to simulate demand paging by breaking a buffer gap scheme up so that there are many small buffers. Each buffer is then paged to disk independantly. If a buffer should fill up, it can be split up into two buffers and insertions can continue. No large motion of text is ever required, but memory is lost.

In none of these systems is linked line acceptable. Memory is assumed to be very tight and the overhead of the extra headers is not acceptable.

2.9 Scratchpad Memory

Scratchpad memory contains the temporary variables allocated by the editor. Because of the transient nature of these variables, it is allocated and freed often. It is used to hold the buffer descriptors, string variables, and--in the linked line scheme--the buffers themselves. The scratchpad memory management required for text editing is relatively simple, but there are some general considerations that are worth mentioning. There aren't too many buffer descriptors and they are of a well known size so they are easy to manage. The string variables can range from being null to being entire buffers. Thus, they can cause fragmentation quite easily. The linked line formats have already been discussed.

In a large address space system, two buffers worth of address space should be devoted to scratchpad storage (to allow for putting an entire buffer there, which takes one buffer worth, and because space is allocated in integer buffers worth). In a small address space system, large operations are typically done character at a time because memory itself is usually at a premium. Therefore, the amount of scratchpad storage needed can be quite small. In any system where the editor can be dynamically extended (see the Command Loop chapter), scratchpad storage needs can vary dramatically and are not generally predictable in advance.

Allocating memory in chunks helps prevent fragmentation, therefore not usually requiring a compactifying garbage collector. If memory becomes badly fragmented, a compaction is requried. In a linked line scheme, compaction eliminates the possibility of using the line pointers as unique ids (they change). Such unique ids are used by the redisplay algorithm.

3. Incremental Redisplay

The most visible part of a screen-oriented text editor is the redisplay process. This is the section of code that keeps the current contents of the buffer accurately displayed on the user's terminal. It has the additional goal of performing this function in such a way that a minimum or near minimum amount of clock time is required in order to fulfill this purpose. Clock time is a combination of transmission time, cpu time, and disk access time which is perceived by the user as the delay from when he enters the command to when the redisplay is finished.

In general, the contents of the buffer will change only a small amount during the basic read command - evaluate it - do redisplay loop. The screen will then only have to be changed by a small amount in order to reflect the changed buffer contents. Hence, the algorithms concentrate on incrementally redisplaying the buffer and the entire process is referred to as incremental redisplay. Fortunately, it turns out that in cases where the buffer is changed drastically, the increment-oriented approach to redisplay works quite well and so there is no need for multiple algorithms.

Our discussion of the incremental redisplay process assumes a model of the system where the editing is done on a main processor which communicates with a terminal. If the main processor is the same as the terminal, the bandwidth of the communication channel can be though of as being very high. The incremental redisplay process is an optimization between cpu time and I/O channel time, with a few memory considerations thrown in. The primary constraint is the speed of the I/O channel. Typical speeds that are currently available are 30 characters/second, 120 cps, and 960 cps. There are also memory mapped terminals which run at essentially bus speeds. Equivalent speeds can be derived and run in the 100 to 50,000 cps range.

A typical video terminal has a 24 x 80 character screen. At 30 cps, it will thus take three seconds to print a line and over a minute to refresh the whole screen. At 120 cps, less than one second is required to print a line and about twenty to refresh the screen. At 960 cps, it will take only one or two seconds to refresh the screen. The speed of the communication greatly affects the amount of optimization that is desired. At 30 cps, even one extra transmitted character is painful to the user, while at 960 cps reprinting entire lines does not take an appreciable amount of time. One dimension of the optimization is thus clear: the importance of optimizing the number of characters sent increases in proportion to the slowness of the communication line.

A user interface issue arises at this point. While it is acceptable from a clock time point of view to reprint entire lines, users do not like to see text which has not changed in the buffer "change" by being reprinted. The flickering that is generated by the reprinting process attracts the user's attention to that text, which is undesirable (the text has not, after all, changed). Thus, avoiding extraneous flickering and movement of text is good. Even with infinitely fast communications and computation, incremental redisplay will still be a desirable feature.

Cpu time must be spent in order to perform these optimizations. If the cpu time that is spent exceeds some small amount of clock time, response will be annoyingly sluggish (and that is not good). It is therefore desirable to minimize the cpu time that is spent on optimizing the redisplay. At this point, the speed of the communication line makes a difference. If the line is slow, extra cpu time can and should be spent (at 30 cps, it is worthwhile to spend up to 30 msec. of cpu time to eliminate one character from being transmitted (which takes about 30 msec.)). However, at higher speeds it is generally not practical to heavily optimize as it can easily take longer to compute the optimizations than to transmit the extra text. This relaxation of the optimization is subject to the user interface constraint outlined above. Memory size constrains the optimization as well. One technique used is storing the entire screen, character by character. This technique works quite well; however, where memory is tight this technique will prove too expensive to implement.

3.1 Line Wrap

There are some more pragmatic considerations involved in the design of the redisplay process. The first of these is line wrap.

Although the editor is editing a one-dimensional stream of text, this text must be placed on a two-dimensional screen in such a way that the user can understand it. There should be no constraints made by the redisplay process on the length of lines. Additionally, there are no commands to "position the screen" or anything of the sort. IT IS THE RESPONSIBILITY OF THE REDISPLAY PROCESS TO HAVE THE SCREEN SHOW MEANINGFUL INFORMATION AT ALL TIMES. The user has almost no control over this function at all, and should not need to. If commands have to be entered in order to obtain feedback, those are commands that are not doing productive editing.

There are two different ways to handle very long lines. One way is to have these lines be clipped at the right hand edge of the screen and then have some indication that the clipping is occurring. The other is to wrap the lines to the next line (i.e., the text that does not fit on one screen line is placed on the next). The first method is acceptable, but not very well human-engineered. Typing text in the middle of a line causes the line to spill and visually lose characters. This losing of characters causes uncertainty in the user's mind about what exactly is happening. In addition, it is never possible to see a long line in its entirety.

The second method is slightly less "clean" when displayed on a screen as wrapped lines will be around, but it does not suffer from either of the above problems. Inserting text might cause a line to wrap (an annoying process) but no text vanishes. Also, long lines are always visible. Finally, wrapped lines are usually only a temporary phenomenon, because most people prefer line widths in the 65-80 character range and this range fits on most terminals. Thus, the wrapped lines appear mainly during editing and will normally go away. Note that it is during the editing process that users most need the feedback. Thus, the line-wrapping method seems to be the best one to use.

In any method, care must be taken to make sure that the pathological case of very long lines works properly. Although rare, non-text (e.g., object code) files are sometimes examined with the editor. These files generally do not break up into reasonable-sized screen lines (a newline indicator might not occur for two or three thousand characters in an object file). Thus, a single line of text might more than fill up the screen. Provisions must be made in the redisplay code to allow the screen to nonetheless be positioned into the middle of such a line.

3.2 Multiple Windows

It is useful to be able to see more than one buffer (or different parts of the same buffer) simultaneously. For example, you can then examine documentation while writing a procedure call. In general, it is not too difficult to set up the redisplay to perform this multiple windowing. The few necessary details will be mentioned in the discussion of the algorithms themselves. Care must be taken that modifications made while in one window are reflected in any other appropriate windows.

3.3 Terminal Types

The redisplay process is the way to communicate to the user. It also has a strong interest in taking advantage of whatever features are supplied by the terminal in order to reduce the time taken for a redisplay. This section will undertake a brief discussion of the various classes of terminals available and how various features affect the redisplay process.

3.3.1 TTY and Glass TTY

A TTY is a canonical printing terminal. Printing terminals have the property that what is once written can never be unwritten. A glass TTY is the same as a TTY except that it uses a screen instead of paper. It has no random cursor positioning. Incremental redisplay for such a terminal usually maintains a VERY small window (e.g., one line) on the buffer and either echos only newly typed text or else consistently redisplays that small window. Once a user is familiar with a display editor, however, it is possible--in a crunch--to use it from a terminal of this type. This is not generally a pleasant way to work.

3.3.2 Basic

A basic terminal has, as a bare minimum, some sort of cursor positioning. It will generally also have clear to end of line (put blanks on the screen from the cursor to the end of the line that it is on) and clear to end of screen (ditto, but to the end of the screen) functions. These functions can be simulated, if necessary, by sending spaces and newlines. A typical basic terminal is the DEC VT52.

Such terminals are quite usable at higher speeds (960 cps) but usability deteriorates rapidly as the speed decreases. It requires patience to use them at 120 cps and a dedication bordering on insanity to use them at 30 cps. Terminals which do not have clear to end of line are even worse.

3.3.3 Advanced

Advanced terminals have all of the features of basic terminals along with editing features such as insert/delete line and/or character. These features can significantly reduce communication time for common operations. Typical terminals in this category are the HDS Concept 100, the Teleray 1061, and the DEC VT100.

These terminals are, of course, quite usable at 960 cps and similar speeds. Due to the reduced need for communication line bandwidth, at lower speeds they are more usable for editing than anything else. At 120 cps, editing text is relatively painless, but merely examining text takes place at a quite slow speed. At 30 cps, even editing is barely acceptable.

There is a subtle difference among some of the advanced terminals. The VT100 supports a scroll window (move lines x through y up/down n lines) feature while the 1061 supports insert/delete lines. Scroll window is more pleasing to see when there is some stationary text being displayed at the bottom of the screen. With insert/delete line, the appropriate number of lines must be deleted and then inserted; the text at the bottom thus jumps. Scroll window does the whole thing as one operation and does not cause the bottom to jump.

The C100 has an interesting feature. It is a fully windowed terminal and thus all operations can be confined to only affect a designated area on the screen. Insert/delete line operations thus do not cause the bottom text to jump and it is even possible to have two windows side by side as the clear to end of line operation does not affect the text in the adjoining window. The window management software thus has much more flexibility in what can be done while remaining within reasonable transmission time constraints.

3.3.4 Memory Mapped

This section covers a wide range of terminals. Their common characteristic is that the entire screen can be read or written at near bus speeds. Typically, this means that the terminal is "built in" to the computer that is running the text editor. In addition, this computer is often a dedicated one, running only one user's processes. Examples of this type of terminal are the Knight TVs (at the MIT AI lab), the Lisp Machine displays, and the wide variety of memory mapped displays available for microprocessors.

The use of memory mapped terminals has several implications for the redisplay process. First, many of the advanced features are typically not available. However, the terminal I/O is so fast that they can be emulated very quickly. Second, it is possible in some cases to use the screen memory as the only copy of the screen. Thus, if reading from the screen does not cause flicker (but writing does), the screen can be read and the incremental redisplay process will run and compare the buffer against it, changing it only when necessary. Finally, if you can write to the screen without flicker, the redisplay process merely boils down to copying the buffer into the screen as doing so is always faster than comparing. Any memory mapped terminal which has a slow access time should be though of as a basic terminal for the purpose of redisplay algorithms.

3.3.5 Terminal Independent Output

A full discussion of this topic is beyond the scope of this thesis. [Linhart] (see the bibliography) discusses this problem more fully. In essence, the problem is that every terminal manufacturer has decided on a different set of features and ways of accessing these features. What must be done to solve the problem is to specify a set of routines which can be called which isolate these differences, as well as a way of selecting among different sets of such routines as the terminal changes.

Some systems already have a solution to this problem and interfacing the editor to that solution is the best way out. For the most part (such solutions are RARE), the person who writes the editor will effectively create one. As will be mention later, the text editor might very well become the de facto solution to the problem. Other programs would merely output to editor buffers and the editor's redisplay code would take care of the rest.

The following set of routines will allow terminal independent I/O for most terminals. They allow full access to the capabilities of TTYs and basic terminals. They will not allow full access to the capabilities of advanced terminals, but they will get you somewhere. Memory mapped terminals usually use a totally different I/O package anyway and so they are not considered.

Basic I/O:

```
GetChar(Character <r>)
```

```
PutChar(Character)
InputWaiting(Number <r>)
Init(Terminal Type)
Fini
```

The first three routines are capable of handling all input and output associated with a full duplex stream device. End of record marks (e.g., new lines) are transmitted as characters. The first two routines get and put raw characters (no translation or checking of any sort is done) and the other one tells you of the state of the buffer. InputWaiting tells you if the user has typed anything that you haven't read yet. If he has, you can read it before calling the redisplay. If the input is coming from a file, InputWaiting will tell you the number of characters left in the file. This interface is a general stream oriented interface. These routines update the internally known cursor position to correspond to the new one (i.e., increment by one for the most part on output). Init sets the terminal type and initializes the terminal to a reasonable state (e.g., do not echo input). Fini undoes whatever init did so as to leave the terminal in some reasonable state for general system use (e.g., not raw I/O, echo input, etc.)

Basic Terminal Control:

```
MoveCursor(x,y)
CLEOL
CLEOS
```

MoveCursor knows where the cursor is and figures out the fastest way of getting it to (x,y). CLEOL sends a command to the terminal to clear from the current position to the end of the line and CLEOS clears to the end of the screen.

Advanced Terminal Control:

```
Insert(String)
Delete(Number)
InsertLines(Number)
DeleteLines(Number)
```

Insert takes String and figures out the most reasonable way of inserting it. Delete deletes characters on the current line. The Insert/Delete Lines routines deal with lines on the screen. In all cases, Number can be either positive or negative and a positive number signifies to the right or below of the cursor, respectively.

3.3.6 Echo Negotiation

Echo negotiation was devised for the Multics system and is a protocol for use by multi-node networks which can cut down on response time by reducing communications overhead. It is useful in an environment where the user's terminal is a one node and the computer which is running the text editor is at another. In such an environment, it can take a long time to send a character back and forth (and it takes nearly the same time to send many).

Echo negotiation can only be used when the point is at the end of a line. The editor can download the front end processor (the node closest to the terminal) with a list of approved characters. As long as the user types only those characters and does not reach the end of a screen line (necessitating a wrap), the front end can safely echo the input characters to the terminal and buffer the input text. When any non-approved character is typed (or the line fills up), the editor is invoked to process the echoed text (the number of already echoed characters is returned to the editor) and the additional character. See section 3.6.2, page 27 to see how this protocol affects the redisplay algorithm.

3.4 Approaches to Redisplay Schemes

There have been two major approaches to performing redisplay. The first is for the routines which are invoked by the user to tell the redisplay code exactly what they did (e.g., "I deleted 5 characters from here"). This approach is not a very clean one and it is prone to error. This is an especially important consideration because we would like to encourage novice users to write their own commands. The extra effort of getting the redisplay correct might make this an impractical goal.

The second approach has been to have the redisplay know nothing about what has occurred. It must rescan the buffer and decide for itself what has and has not changed. This process requires a copy of the screen and can be expensive in cpu time. This algorithm will be presented first because of its relative simplicity.

There is a compromise between these two approaches which seems to solve all of the problems. This compromise is to have the memory management software communicate with the redisplay software. User routines know nothing of this communication and cannot cause bugs in it. On the other hand, the cpu time require for a redisplay is somewhat reduced and is more spread out and so it is not as noticeable. Extra memory is required to handle the communication, but in some cases, the screen representation can be discarded and so the net result could be a memory gain. It is this compromise that is the heart of the "modern" redisplay and it is the other one to be presented.

3.5 The Framer

The framer is the part of the redisplay that decides what will appear on your screen. In the stable state, there are two different approaches used.

First, the TopOfScreen and BottomOfScreen marks are kept around. As long as the point stays within these marks, we expect that the point will remain on the screen. Thus, the top of the screen can be assumed to be in the proper place and the redisplay algorithm can be started directly. If it does not (the redisplay code detects this error and generates a FramerError), the framer runs again, but uses the next approach.

Second, if the point is outside of the screen marks, it is simplest to assume that the entire screen will be changed. Thus, the framer wants to recenter the point on the screen. It can start by counting back <screen height> / 2 lines. Assuming that there are no wrapped lines, this method would work fine. At this point, the framer checks this assumption (that there are no wrapped lines) by counting forward character by character, keeping track of how many lines are actually used along with the intermediate results. If there are no wrapped lines, the new guess will work fine. If there are wrapped lines, it will look at the intermediate results and decide how many lines to throw away to leave you approximately centered. If the advanced redisplay algorithm is used, these intermediate results should be recorded as they might be needed.

If all the lines have to be thrown away (i.e., the current line is VERY long), the third and most desperate mode must be used. Here, the framer figures out, character by character, where each character on the current line is. It then decides how many characters to move back before starting the redisplay, while staying within the same line.

3.6 Redisplay Algorithms

Here are presented the two major redisplay algorithms and an discussion of how to adapt these algorithms for memory mapped terminals. These algorithms will not go into every detail (or even most of them) as doing so would inundate the description with too much detail. This detail is discussed in later sections.

3.6.1 The Basic Algorithm

```
call Framer;
           /* TopOfScreen is a mark returned by
           the framer */
BufLoc = TopOfScreen;
           /* loop over the whole screen */
do Row=1 while(Row <= HeightOfScreen);
   do Col=1 while(Col <= WidthOfScreen);

           /* found a NewLine char */
       if Buffer(BufLoc)=NewLine
        then do;
           /* is the rest of the line blank? */
          do i=Col to WidthOfScreen;
             if Screen(i,Row) ↑= " "
               then do;
                     /* if not, make it so by
                     sending a CLEOL at the
                     non-blank */
                 call MoveCursor(i,Row);
                 call CLEOL;
                 do j=i to WidthOfScreen;
                    Screen(j,Row)=" ";
                 end;
                 leave;
               end;
          end;
          BufLoc = BufLoc + 1;
           /* move to next line */
          Row = Row + 1;
          Col = 1;
          leave;
        end;

           /* no NewLine, so has there been a
           change in the buffer? */
       if Screen(Col,Row) ↑= Buffer(BufLoc)
        then do;
                /* if so, change the screen
```

```
                                  to match */
                              call MoveCursor(Col,Row);
                              call PutChar(Buffer(BufLoc));
                              Screen(Col,Row)=Buffer(BufLoc);
                           end;
                        BufLoc = BufLoc + 1;
                        Col = Col + 1;
                               /* save the (x,y) of the point so
                               that we can put the cursor there later */
                        if BufLoc=Point
                          then do;
                              PointX = Col;
                              PointY = Row;
                           end;
                  end;
                  Row = Row + 1;
                  Col = 1;
            end;
                      /* framer missed--it almost never happens */
            if BufLoc < Point
              then call FramerError;
            EndOfScreen = BufLoc;
            call MoveCursor(PointX,PointY);
```

This algorithm is quite straightforward. It first calls the framer to match the top of the screen with some point in the buffer. It then iterates through the buffer and the screen simultaneously, matching characters as is goes. As long as the character on the screen matches the character in the buffer, no action is taken. When there is a discrepancy, the cursor is moved to that position by means of the MoveCursor routine, the changed character is printed, and the screen array is updated. If the line gets to be too long, it is wrapped automatically. If a NewLine character is encountered, the rest of the line is checked to make sure that it is all blanks. If not, blanks are put there. Finally, a note is made of where in the buffer the end of the screen falls.

This is your basic, garden variety redisplay algorithm. It will work on any terminal that supports cursor positioning (the CLEOL call can be faked by sending spaces). It will work quite well on anything running at 480 cps or over. Its only memory requirements are an array large enough to hold the screen (typically 1920 characters). The only inte:.:tion between the redisplay algorithm and the memory management system is two marks. Finally, it is not told anything about what changes were made and so it figures everything out for itself each time it is called. There can thus be a cpu time penalty associated with this algorithm that might make it slow enough to be painful. The next section describes with an algorithm which gets around this penalty.

A complete redisplay can be generated quite easily us:.:g this algorithm. The GenerateNewDisplay routine will set the cursor to home and then clear the screen and the internal screen array. It then calls the incremental redisplay routine. The incremental redisplay routine will simply do its normal job, which in this case implies sending all of the non-blank characters to the terminal. The NewDisplay routine must also remember to send such things as status displays, which are not sent during an ordinary redisplay.

A status display is text that is kept on the screen but is not often changed. For example, the Emacs status display has the editor name, the mode name, the current buffer name, and the file name displayed on a line near the bottom of the screen. Ordinarily, the redisplay code ignores this section of the screen.

3.6.2 The Advanced Algorithm

The advanced redisplay algorithm serves two vastly different purposes. First, it provides a way of efficiently taking advantage of the insert/delete line/character functions which are supplied with some terminals. Second, it provides a low cpu overhead way of performing a redisplay on basic terminals.

The basic idea used by this algorithm is to assign a unique id to each buffer line that appears on the screen. Note that a buffer line can take up more than one line on the screen by wrapping. Just to make sure that the definitions are clear, here they are: a buffer line (BufferLine) is either the text between two newline characters (in the buffer gap memory management scheme) or the text in one element of the line list (in the linked line scheme). A screen line (ScreenLine) is a horizontal row of characters on the user's display.

The unique id can be in any form. One method is to use a 32 bit counter and increment it each time any line is changed. After the change is made, the line is assigned the current value of the counter. If it is changed again, it gets the new value of the counter. The assignment can be made in an otherwise unused part of the header (for linked line) or in a special mark (for buffer gap). In a linked line scheme, the pointer to the line can serve as a unique id.

These unique ids only have to exist for lines that appear on the screen. Thus, the buffer gap scheme only has a few of these special marks that must be maintained. The special marks are placed at the beginning of each line that appears on the screen. They contain a version number for the line as well as the location of the mark.

The memory management scheme is responsible for maintaining this extra information. Thus, it and the redisplay code can interact heavily and the specific redisplay process chosen will affect the internal structure of the memory management scheme.

There are two flags that can be kept by the memory management software which will aid the redisplay process. First is the buffer modified flag. This flag is usually kept anyway so that the editor can detect when the buffer has been modified. (The details of manipulating it were discussed with the interface routines in section 2.1, page 3.) If it has not been set, the redisplay code knows that it generally will not have to do anything except move the cursor. If the point is still on the screen (remember that we have beginning and end of screen marks), its position on the screen can be calculated with much less effort that is required for a full redisplay. If the flag has been set, a full redisplay is required and the flag will be reset (the editor proper ORs this flag in with a private flag (Modifiedp; mentioned in the buffer data structure descriptions) in order to properly remember whether the buffer has been modified).

Another flag (which has not been mentioned before) can significantly reduce redisplay computation in some cases. Assuming that you are located at the end of a BufferLine, it tells you whether or not *any operation other than inserting a character has been done. If the flag says not, all* that the redisplay has to do is output the one character (after checking for wrap, etc.). A significant amount of time can be saved this way, but it is most useful with a negotiated echo protocol (see section 3.3.6, page 23). The exact interface to this flag will not be defined.

The redisplay algorithm itself starts by trying to find a match between the BufferLines and the ScreenLines by using unique ids. The unique ids are compared, line by line. If they match, no work needs to be done and the redisplay proceeds to the next line. If they don't, it can be for one of three reasons:

- An additional line (or lines) was inserted between the two ScreenLines. This condition is detected by comparing the ScreenLine unique id with all of the BufferLine's unique ids and finding a match. (Remember that the ScreenLines are what the BufferLines were one redisplay iteration ago.) We thus have the situation where we used to have A,B and now have A,<junk>,B. Clearly, the most reasonable assumption is that <junk> has been inserted. We thus count how big <junk> (the framer has already calculated this information) is and tell the terminal to insert the appropriate number of lines. (Before you do this, however, you must first delete the same number of lines from the end of the window in order to keep from losing the text at the bottom of the screen.)

- A line (or lines) was deleted. This is detected by comparing the BufferLine unique id with all of the ScreenLine's unique ids and finding a match. We thus have A,B,C becoming A,C. We delete the appropriate number of lines and then insert them again at the bottom of the window.

- The line was modified. This is detected by not finding either of the above matches. At this point, we switch to intra-line work and do the following:

  *Do a string compare starting from the beginning of each line (the BufferLine and the ScreenLine) and see how much they have in common. (If this says the whole line matches, no more work has to be done.) For example, if the ScreenLine is "abcdef" and the BufferLine is "abxdef", they have two characters in common from the start.

  *Do the same thing starting from the end. The example strings have three characters in common from the end.

  *Compare the line lengths. If the two lines are the same length, you only need to rewrite the changed part (e.g., two characters were interchanged). In the example strings, the lengths are the same (6). This optimization can be done even on a basic terminal. If the two lines are not the same length (for example, the ScreenLine is "abcdef" and the BufferLine is "abxyzdef"), rewrite as much of the portion between the common text sections as possible ("x") and then either insert or delete the required number of characters (in this case, insert two blanks) and finish writing the modified text ("yz"). Remember that if there is no common text at the end and the BufferLine is shorter than the ScreenLine, a CLEOL call is appropriate.

  *Wrapped lines can pose a problem. There may be no end common text, and yet an insert or delete character operation might be the appropriate one. (If the screen width is six characters, the ScreenLine is "abcdef", and the BufferLine is "abcxdef". Here, the BufferLine will ultimately become two ScreenLines, "abcxde" and "f".) This case is detected by having no end common portion and noticing that the line wraps. A more complicated matching process can detect the situation and appropriate action can be taken.

3.6.3 Memory Mapped

Redisplay for memory mapped terminals boils down to one of three cases. Each case is relatively simple.

1. Reading from and writing to the screen cause flicker. The solution is to use the basic terminal redisplay scheme.

2. Reading does not cause flicker but writing does. The solution is to use the basic terminal redisplay scheme, but use the actual screen memory for storing the screen array.

3. Neither reading or writing cause flicker. On each redisplay cycle, merely copy the buffer into screen memory, not forgetting to process new lines, etc., as needed.

3.7 Other Details

There are a number of other details that must be carefully watched when writing redisplays. None of them are particulary worrisome in themselves, but they collectively clutter the algorithms a great deal. The problems that they pose will be described and they are each simple enough that specific solution algorithms are not required.

3.7.1 Tabs

It helps to think of a tab character in a buffer as a cursor control command saying, "think of me a N blanks, where N is the number of columns to the next tab stop." Thus, whenever you see a tab you want to figure out what N is, and then check to see that the next N colums are blanks, increment the cursor by N, etc. Tab stops can be set in an array (for arbitrary placement of tabs) or set every C columns. In a one origin numbering system, tabs set every C columns are set at positions 1, C+1, 2C+1, 3C+1, ... For example, when C=8, tabs are in columns 1, 9, 17, 25, 33, etc. Again, assuming a one-origin system, the equation for N is:

```
N = C - mod(X-1,C)
```

(X is the column position.)

3.7.2 Control Characters

In general, only the new line character(s) and tabs are interpreted; other control characters are displayed in some reasonable printing representation. One popular representation is "↑" followed by the character whose ASCII value is <control char> + 64. The character control-a is thus printed as ↑A. (The ASCII DEL character, 127, can be printed as ↑?.) This convention has been followed in this thesis. When displaying control characters, you must remember that while the character itself is only one character, it displays in a two character wide sequence. In addition, it is the actual displayed sequence that is stored in the screen array (e.g., "↑" and "A", not "↑A"). Care must be taken to insure that control characters can wrap properly across line boundaries (e.g., the "↑" is not displayed at the end of one line with the "A" at the beginning of the next).

3.7.3 End of the Buffer

If the entire buffer fits on the screen, you will run out of buffer before you run out of screen. Thus, whenever BufLoc is incremented, a check should be made against the buffer length. If you do run out of buffer, remember to finish blanking the rest of the screen if it needs it.

3.7.4 Between Line Breakout

The redisplay process does not have to run to completion before editing resumes. Instead, it can get to a convenient spot (in the basic algorithm, almost any spot will do; in advanced algorithm, stop after finishing a line) and check the input buffers. If more input has arrived, it can abort the redisplay and process the input. Remember that the purpose of redisplay is to provide feedback to the user. If he has already typed something, he does not need feedback immediately. (However, if you can give it to him in a way that does not slow him up, do so.)

3.7.5 Proportional Spacing and Multiple Fonts

Displaying text in a proportional spaced font is not too difficult. Instead of assuming that each character has a width of one, the width can vary and it must be looked up each time it is needed.

Displaying multiple fonts implies receiving a command to switch fonts at some time during the redisplay process. These commands can be stored in the buffer (in which case like NewLines they are interpreted and not displayed just) or in some other structure.

3.7.6 Multiple Windows

There is a database somewhere which describes what windows (i.e., what part of which buffers) are to appear on the screen. One way to perform redisplay with multiple windows is to call the incremental redisplay routine and pass it as an argument each window descriptor in turn. Another way is more suitable for use with the advanced algorithm and it involves having a separate descriptor for each line of the display (i.e., the same database sorted backwards as well). This descriptor tells you where to get each line from.

If a row of dashes ("-----") or any other character string is used as a visual separator between windows, it can be implemented as an additional buffer/window combination and no special casing is required for the redisplay code.

4. The Command Loop

This command loop is the part of the editor that actually implements the logic of the editor. It is responsible for reading in commands, executing them, and "printing" the results. In the process of executing them, it must accept arguments and bind the input characters to functions. This chapter will discuss the command loop. It will also discuss some distantly related issues: the tradeoffs between kill buffers and an undo function, the provisions for recovering from errors, and considerations for selecting implementation langauges.

4.1 Basic Loop: Read, Eval, Print

The basic loop is:

```
do while(TRUE);
        call GetChar(Char);
        call Eval(Char);
        if abort
            then leave;
        if InputWaiting() = 0
            then call IncrementalRedisplay;
end;
```

Note the two details that have been added to what was mentioned in the section heading. First, an abort flag is checked to see whether we are supposed to exit the edit session. This flag is set by the Eval routine. Eval works by invoking a function which was specified by the input character. This function's only result is the change in the state of the editor (e.g., an "x" has been inserted). The "printing" (actually, an incremental redisplay to the screen) is done only if the user has not typed in anything more to be processed.

4.1.1 The Philosophy Behind the Basic Loop

The basic loop as described puts the fewest restrictions on the user interface that can be managed. Each character, in its raw form, is mapped to a procedure which is in turn evaluated. Any arbitrary syntax and semantics can be implemented with this base.

In theory, a syntax of commands being words (e.g., "delete", "move", etc.) could be implemented in this structure by having either a large number of dispatch tables (and thus implementing a symbol state table architecture) or a procedure which parses the syntax of the command via conditional statements. For reasons which will be stated, this syntax is not generally implemented.

Consider the thought that every character that is typed at the keyboard causes a function to be executed. The first conclusion that results is that it is silly to type "insert x" or anything like that when you want "x" to be inserted. As this is a very common operation, it makes more sense to bind the key "x" to the InsertX function. (Actually, it is probably bound to SelfInsert, a function which looks at how it was invoked--the input character--to determine what to insert).

Now, all of the straight, printing, ASCII characters have been taken and bound to SelfInsert. (While there are a large number of special characters that are not often typed, leaving them in consideration does not materially affect the conclusions.) The remaining things that can be entered from an ASCII keyboard are the control characters, the delete key, and the break key. These could be bound to functions that implement a complex syntax, but why bother? It is not too difficult to learn even a large number of key bindings, so let us bind the control keys directly to useful functions. For example, ↑F could be ForwardCharacter, ↑D could be DeleteCharacter, etc.

33 functions are not enough for even the commonly used functions. Thus, some of the keys should be bound to functions which rebind the dispatch table. For each of these rebinding functions, 128 new functions are made available (there is no reason for the printing characters in them to be bound to SelfInsert). Note that the break key is not used in this scheme as it is hard to work with (it does not have an ASCII value).

Thus, even though we began with a structure for the command loop that did not to impose any constraints on the syntax of commands (and thus was as general as possible), we arrived at a specific syntax for commands. This syntax is to bind the printing characters to SelfInsert, bind the control characters to a mixture of useful functions and rebinders, and to have about three or four alternate dispatch tables (enough to supply many hundreds of commands). Thus, commands are rarely more than two keystrokes long. The price that is paid for this brevity is a longer lead time in learning to use the editor effectively.

(Note that most of the increased lead time in learning the editor is NOT from the brief commands, but because there are more commands to learn. Given a "conventional" editor (e.g. DEC's SOS) and an equivalent subset of an Emacs-type editor, novice users will learn the subset of the Emacs-type editor faster.)

4.2 Error Recovery

Errors come in two flavors. There are internal errors which are in the editor itself (e.g., a subscript out of range) and external errors which are caused by the user (e.g., attempt to delete off the end of the buffer). There is also a non-error, the normal exit, which will be treated as an error in this discussion. These errors will, in general, be indicated both from within the editor and from the outside world (the operating system).

The first category to be considered will be internal errors. These errors cause an immediate exit to the operating system with no questions asked and no delays tolerated. They will be internally generated by such things as arithmetic overflows and bad subscripts. (While the editor might catch and process some of these, it will not in general process them all.) They can also be generated externally and often are (e.g., process switching). The factor in common is that they are unpredictable and the state of the editor should remain exactly intact. The user should also be able to signal such an error to abort out of the editor. He might want to do this because of a problem with the editor itself (e.g., infinite loop) or because he wants to do something else. This signalling is usually done with the help of the operating system. In any case, the precise state of the editor should be retained so that it can be resumed exactly where it left off. Most operating systems have some facility for doing this; they differ principally in the freedom of action that they allow before losing the state. This freedom ranges from nothing to doing arbitrarily many other things.

At the user's discretion, the editor should be restartable either from exactly where it left off or at a safe restart point. This point is ordinarily a portion of the editor which recovers the buffers and other current state and then resumes the command loop.

External errors are principally user errors. The action ordinarily taken is the display of an error message and a return to command level. The implementation of this level of recovery is built in to the procedures which implement the commands.

There is a variation of external errors which are generated manually by the user. Typically, these involve backing out of an undesired state (e.g., the unwanted invoking of a dispatch table rebinding or aborting an undesired argument). The bell character (ASCII ↑G) has often been used for this purpose. In this case, the procedures will know that a bell has been typed and will implement the backout protocol.

Finally, provisions to exit the editor must be made. This is ordinarily by means of an abort flag of some sort as can be seen in the previous code fragment. Note that various other uses might be multiplexed onto this abort flag, signifying varying levels of "exiting." For example, one level could used by buffer switching in order to rebind the dispatch tables (see the section on later in this chapter).

Ordinary exiting involves several types of processing. The editor might ask the user what to do with buffers that have been modified but not written out. If, as is ordinarily assumed, the state of the editor is preserved across invocations, the state must be saved. If not, it must be sure that all memory is deallocated. Finally, the user's environment should be restored as it was found. This implies such varied things as cleaning up the stack, closing files, deallocating unneeded storage, and resetting terminal parameters.

4.3 Arguments

Arguments are specified by the user to modify the behavior of a function. The Emacs argument mechanism will be described as an example of three diverse ways in which arguments are obtained.

There are three standard argument types. First are prefix arguments. These are invoked by a string of functions (which are in turn invoked by characters typed before the "actual" command) and are an example of using the key/function binding to implement a more complicated syntax. Next are string arguments. When obtaining a string argument, the editor is invoked recursively on an argument buffer and upon return from the recursive invocation the contents of that buffer are given to the requesting procedure. Last are positional arguments. These are the internal variables of the editor.

4.3.1 Prefix Arguments

Prefix arguments are entered before the command whose behavior they are modifying, thus, their interpretation must not depend upon the command. Emacs limits these to numeric values.

Ordinarily, commands will have an internal variable available to them named something like "argument" and it will have a value of one. Prefix arguments allow the user to change that value to any other positive or negative integer.

Arguments are used for two different purposes. First is to specify a repeat count for a command. Thus, <12> ↑F would go forward twelve characters (assume the ↑F key is bound to the ForwardChar function). The other use is to tell a command to use an alternate value for a parameter. If

FillParagraph was bound to ↑P, then <65> ↑P might say to, for this time only, use 65 as the desired width of the paragraph (the right margin) after it is filled. Alternatively, it might say to reset the default value of the right hand margin to 65 and then use that value. It is useful to provide a predicate to allow procedures to determine whether an argument has been given. This allows them to differentiate the default argument of one from the user entering one as the argument value.

Emacs uses ↑U as the UniversalArgument function. It can be used in either of two ways. ↑U ↑F means to go forward four characters. Adding another ↑U means to multiply the current argument by four. Thus, ↑U ↑U ↑U ↑F means to go forward 64 characters. The factor of four was selected because five is too large (1, 5, 25, 125 goes up too fast) and, while three might have better spacing (1, 3, 9, 27, 81, 243), the powers of four are known by all people who are likely to be around computers.

The other use is more complicated. ↑U 1 2 ↑F means to go forward twelve characters. ↑U - 1 4 7 ↑A means to give ↑A an argument of -147. The ↑U in this case serves as an "escape" to logically rebind the 0-9 and - keys.

On some terminals, there are two sets of numeric keys (one set that sends the ASCII "0" - "9" codes and another that is labeled with digits but sends different codes) to generate "numbers" than simply sending the appropriate ASCII codes. In this case, these "other numbers" can be bound directly to argument generating functions and the initial ↑U is not needed.

4.3.2 String Arguments

String arguments are specifically requested by a procedure. A prompt is displayed and the user enters the value of the argument. The procedure uses this value in any way it desires.

One way to implement such a way of entering arguments is to create an argument buffer in a new window, display a prompt, and call the editor recursively with that as the current buffer. By following this scheme, the full power of the editor is available to correct typing mistakes or otherwise make the entry process easier.

When implementing any argument entry scheme, there are three things to take into account. First, the key or key sequence used to indicate that the entry process is over should be able to vary depending upon who is asking for the argument. ↑M (<cr>) and ↑[(<esc>) are both commonly used as delimiters. Second, there should be a clean way to abort out of the argument entry process (↑G is commonly used for this purpose). In this case, the calling procedure should be told about the abort in order for it to terminate gracefully. (Most of the routines that ask for arguments do all of the asking at once and then proceed to do a large amount of work (e.g., ReadFile). Thus, aborting out of the argument entry process effectively aborts out of the command. Aborting cannot be done cleanly if commands are written to get an argument, do some work, get another argument, etc.) Finally, null arguments (the user enters only the delimiter character) can be used to cut down on typing errors if the procedures supply some reasonable default values.

Here are some examples of using string arguments:

SearchString:	Ask for a string and look for it in the buffer. If the user enters a null string, use the same string that he searched for before.

ReadFile: Ask for a string and, using it as a filename, read the file into the buffer. If the user enters a null string, use the current filename associated with the buffer.

ChangeBuffer: Ask for a string and, using it as a buffername, make that buffer the current one. If the user enters a null string, use the buffer that he was in last (i.e., the one that he was in before the one that he is in now).

Note that SearchString typically uses ↑[(<esc>) as the delimiter while ReadFile and ChangeBuffer typically use ↑M (<cr>). In order to help the user, it is nice to automatically remind him which delimiter is being asked for. Here are some example prompts:

```
Search String(<esc>):
Input File Name(<cr>)  (Default is >u>fin>test):
Buffer Name(<cr>)  (Default is foo):
```

Note that some prompts helped the user by reminding him of the default value.

While all of the examples asked for and wanted a character string, this might not always be the case. It is quite practical to use this method to enter numeric values. The requesting procedure merely has to convert the read-in character string to a numeric value.

4.3.3 Positional Arguments

Positional arguments are not directly specifiable by the user. They are the internal variables that are used in the editor. Such variables include both those required by the editor (e.g., the length of the buffer, the locations of the point and the mark, etc.) and those which have a specialized purpose (e.g., the current value of the right hand margin, the tab spacing, etc.).

Often these values are used in unusual ways. For example, the horizontal position (column) of the point can often be a more pleasant way of specifying a value than entering a number. The user can indicate that "this is where I want the right margin to be" instead of having to count characters to get a number. The user indicates this value by using other commands (e.g., ForwardChar, ForwardWord) to move the point to the desired location. See also section 5.2, page 42 for information about how graphical input devices (mice, tablets, touch sensitive displays) affect positional arguments.

4.4 Rebinding

Rebinding is a name for the act of changing at run time what a key or procedure does. The distinction between the two (keys and functions) is important. Changing the binding of a key means that when that key is typed, the new procedure (the one that is now bound to the key) will be executed instead of the old one. Changing the binding of a procedure means that whenever that procedure is invoked, the new version will be executed instead of the old one. This change affects not only any keys bound to that procedure but also any internal references to it.

There are two levels of rebinding functions. Level I rebinding is when the new procedure must be known before invoking the editor. Level II rebinding is when the new procedure can be defined after the editor is invoked. Unless otherwise stated, level II rebinding is assumed.

To a first approximation, editors that are written in compiled languages (e.g., PL/1) can only change the key bindings and interpreted editors (those written in, say, Lisp) can change both bindings. Dynamic linking, however, allows both bindings to change in compiled editors and so this distinction is not always a proper one to make.

4.4.1 Rebinding Keys

The process of key rebinding is a relatively simple one and it is done essentially the same way in all implementations. A set of dispatch tables is used to map keys (represented by their ASCII values) to their respective functions.

In languages such as Lisp and PL/1, the table can contain the procedures themselves. In less powerful languages such as Fortran and Pascal, the dispatch table branches to a different part of the same routine that contains the table. There, the procedure call is made. In languages that supply it, a case statement can be used instead of the n-way branch.

None of these command procedures have any formal parameters, and so they can all be invoked with the same calling sequence. Thus, the Lisp and Pl/1 direct invocations can work properly. Note also that simple commands do not have to have a separate procedure assigned to them, but the code to execute them can be placed in-line in place of a call (where the case statement equivalent is used). Doing this substitution loses some potential flexibility.

4.4.2 Rebinding Functions

Level II function rebinding is ordinarily a language-supplied feature and so it will not be discussed in depth. Two comments will, however, be made on how to simulate it.

If the underlying operating system has dynamic linking (e.g., Multics), a procedure may be rebound at run time. Dynamic linking is a way of linking procedures together in which the actual link is not made until the procedure is about to be executed. At that time, the procedure is located in the file system and brought into memory. The link may either be left alone, in which case the next call will have the procedure re-located (a relatively expensive process) or it may be snapped. Snapping a link implies converting the general call instruction (which is kept in a special, writable part of the program) into a call instruction to the appropriate address. If a link is snapped, it must be explicitly unsnapped before any re-locating is done.

If the operating system does not support dynamic linking, the editor writer might choose to simulate it manually. Such a process is complex and some thought will have to be given to exactly how desirable rebinding functions is. The process is tantamount to explicit overlaying.

This all has a straightforward bearing on rebinding functions. Rebinding a function involves changing the definition of the procedure that is invoked by referencing it. What has been discussed are ways of changing such a procedure definition. Note that if the code to execute a function is inserted in-line in the basic editor, it cannot be rebound by any of these methods.

If dynamic linking is not available and is unfeasible to simulate, there is still one way out. This way will only provide level I rebinding. Instead of just using one dispatch table which indicates a procedure to be called directly, use two. The first table maps from keys to the operation to be performed (e.g., ↑F is mapped to moving forward one character). The second table maps from the operation to be performed to a procedure to perform it (e.g., moving forward one character is mapped to ForwardChar).

4.5 Modes

Modes are collections of rebindings which are done all at once. They can either be done automatically or can be explicitly asked for by the user.

An example of an automatically loaded mode might be PL/1 mode. This mode will automatically be loaded whenever a file whose name ends in ".pl1" is read into a buffer. Such a mode might do several things. It might rebind the internal variable that identifies which characters are legal in tokens (i.e., variable names) to also include the "$" and underscore characters which can occur within PL/1 names. This change would make the ForwardWord function treat a PL/1 variable name as a word. The mode might also rebind the ";" key to be an electric semicolon (i.e., finishing one statement would cause it to automatically indent properly for the next one).

The process of autoloading is related to automatically loaded modes. The trigger is the main difference. In autoloading, the trigger in completely internal. An example could be the set of S-expression hacking commands. Although they are defined at all times, the code for them is not necessarily a part of the editor. Instead, when any of the commands is invoked, they are autoloaded into the editor and the command is executed.

An example of a user requested mode would be auto fill mode. This mode rebinds the space character to one that checks to see if you are typing past the right margin. If you are, it breaks the line up to fit within the right margin. It also inserts the space.

A printing terminal mode would use function rebinding. It would be loaded automatically whenever the editor is used from a printing terminal instead of a display. It might rebind the SelfInsert function (which is used by all of the 95 printing keys) to one that prints the character that it is inserting on the terminal (and then inserts it). In this case the definition of the function changed and so function rebinding is called for. Note that this change is global over all buffers and so it is not readily simulatable by changing the bindings of keys to operations.

The function rebindings that are commonly done by an editor are known in advance and so they can be done by any implementation (see the preceding section for a discussion of the difficulties involved in function rebinding). Fully dynamic rebinding (the new definition of the procedure is not known until run time) is desirable for several reasons.

- Debugging is greatly eased if the trial-and-error cycle time is reduced by not having to compile and link the whole editor each time. Instead, only one function has to be recompiled and linked. (In languages such as Lisp, it is more accurate to say compiled/linked as the two operations are synonymous.)

- Space savings are achieved if unneeded modes and autoloaded single functions are not brought into memory until asked for.

- If the editor is implemented in an interpreted language (see the next section) users can develop their own functions relatively easily. Such "sideline" development is advantageous because it allows many people to develop useful code and so the editor can be specialized in many more ways than any reasonable support group could ever implement on their own. It also encourages tailoring the editor to a user's own taste and so his productivity is enhanced.

4.5.1 Implementing Modes

Modes are on a per buffer basis and so provision must be made for changing these bindings as buffers are switched. The general technique for doing this is to have a set of default bindings and a set of current ones. When a buffer switch is made, the default bindings are copied to the current ones and then a series of procedures are run which modify the set of current bindings to be the correct ones for the modes that are active on this buffer.

A different approach would be to have a separate environment for each buffer which is created with the buffer, is modified as modes are added, and is never thrown away. This approach leads to efficiency problems because of the large amount of storage overhead associated with each buffer.

Sidenote Calculation: Assume that there are two dispatch tables of 128 commands each and that each entry is four bytes (big enough for an address). This leads to 1K bytes just for the dispatch tables per buffer. In addition, you have another 1K bytes for a default table to use when creating a new buffer. With a current/default dispatch table scheme, you have 2K bytes per editor and so you are always as efficient and better in the case where you have more than one buffer. Procedural storage overhead is essentially the same. In one case, you invoke the state building procedure once (but in general cannot undefine the procedure) and in the other case, you invoke it with each buffer switch. It does, on the other hand, take longer to switch buffers but the incremental time is usually minimal.

There is *an important flexibility tradeoff*. With a mode list and the associated default/current dispatch tables, it is possible to remove a mode from a buffer. If each buffer has its own dispatch table which is incrementally changed whenever a new mode is added, it is not generally possible to undo such changes. Note that while the dispatch tables were used as an example, it is by no means the only variable whose value may change on a per-buffer basis.

4.6 Kill and UnDo

An Emacs maintains a kill ring which is a place where all significant chunks of deleted text get placed. (Those deleted with C-d and <del> do not get saved.) There are commands to push and pop things from the current spot in the ring and to rotate the ring so that different text is at the current spot. Typically, a maximum of ten or so items are kept in the kill ring.

Moves and copies of text are done with this ring. Thus, there is a mechanism by which the user can recover accidentally deleted text. This type of error is the most harmful one that can occur as it involves losing information.

The InterLisp system (and others) provides a more general undo facility. Invoking this facility will cause the system to "undo" whatever it was that you just did (for one command only: a second "undo" will undo the first one). In order to implement this facility, the system must keep track of everything that you do and what its effects were.

While this general purpose facility has good applications, it is not clear that a text editor is one of them. There are three basic areas where undo applies to text editing. These are: moving around in text, deleting text, and file i/o. The Emacs approach and the undo approach will be compared for each of these.

Moving around in text is simply solving the problem "I am at x and I want to be at y." The Emacs solution involves translating this difference into a sequence of commands to move the point from z to y. If a mistake is made in the process of implementing the solution, the problem is merely restated to "I am at x' and I want to be at y" and it is re-solved. The undo solution differs by detecting the error (i.e., deviation from the intended solution), saying "undo" to put you back on the original path, and proceeding. Ordinarily this difference in the two solutions is not very great.

If the accidentally typed command is one that moves you a great deal (e.g., move to the beginning of the buffer), it is not always easy to recover with the Emacs solution because you might not remember exactly where you were. Emacs solves this by having the large movement commands set the mark to where you were. Thus, an interchange point and mark sequence will recover from the error.

The undo actually helps less in the text deletion case. There, the "canonical" undo will only recover the last command and, hence, the last delete operation. There is no provision for deleting something, moving somewhere else, and undeleting it. Nor is there a provision for recording multiple deletions. Thus, the Emacs approach is more flexible.

Finally is the case of file i/o. Different implementations of Emacs will do different things but the basic idea is to let the user do what he wants. Obvious things will be checked (the file was modified by someone else since it was read in, for example) and such things as deletions will be double checked with the user but no recovery will be provided. On the other hand, not all systems can support the overhead of the multiple copies of a file that would be required by undo, nor are there always ways to manage these extra files conveniently. (The DEC TOPS-20 operating system does do a reasonable job at this, but it is far from perfect.)

The basic conclusion is that while an undo facility is nice, it is not all that useful in the context of an Emacs type text editor.

4.7 Implementation Languages

The language that the editor is implemented in can greatly affect the ease of writing, maintaining, and extending it. Some brief comments will be made about several classes of programming languages which might be considered as implementation languages.

4.7.1 TECO

(This discussion refers to MIT TECO and not the TECO which is supported by DEC on several of its machines. MIT TECO is much more powerful.) TECO is a text editor. Its command language is so powerful that it is usable to write other programs in. It is tailored for writing text applications and so would seem a good choice. It has two major problems:

- It is the only language less readable than APL. A listing of a TECO program more resembles transmission line noise than readable text. Thus, maintenance is a problem.

- Its only implementation is on the PDP-10/DEC 20 series of computers. Implementations on other machines involve asking the question of what you write the TECO in.

4.7.2 Sine

Sine is a Lisp-like language tailored for text applications. Its only implementation to date is on Interdata 7/32 (or Perkin-Elmer 3200) minicomputers running the MagicSix operating system developed at MIT. It is interesting because it is a language tailored for implementing editors. It is a example of an "ideal" implementation language. [Anderson] discusses this language in detail.

Sine is composed of two parts. Sine source code is assembled into a compact format. This object code is then interpreted. It allows function rebinding and other such nicities and the interpreter implements such things as memory management and screen redisplay automatically. Thus, the resulting editor is nicely structured, with "irrelevant" details hidden away.

4.7.3 Lisp

Lisp is probably the best choice, if it is available. The Lisp must, however, have string operations in order to run with any efficiency. It is best suited for the linked line form of memory management because of its view of memory management. Lisp provides a nice interpretive language for escaping into to easily write complicated editing macros. It also is quite readable and maintainable. It also provides function rebinding. Some Lisps have compilers whose code can run very fast, so speed need not be a problem.

4.7.4 PL/1, C, etc.

PL/1, C, and other such "systems languages" are widely available in reasonably efficient implementations. They allow the straightforward manipulation of complicated data structures and yet remain generally readable. They specifically support containment of detail by independently compiling several related routines and their internal data structures.

As a specific example of the latter, it is possible to write a buffer management abstraction in which the only visible parts are the entry points. The specific method chosen to represent the buffer remains well hidden.

4.7.5 Fortran, Pascal, etc.

Fortran, Pascal, and other such languages are the least acceptable (except, of course, for assembler). In general, one must either simulate a missing basic feature (e.g., Fortran and If-Then-Else) or circumvent a "feature" (e.g., Pascal and lack of multiple entry points to procedures) in order to do useful work in such languages.

5. User Interface Hardware

The only way for a user to interact with the text editor specifically or the containing operating system generally is by means of the keyboard/screen combination. The chapter on Incremental Redisplay discusses the use of the screen in detail. This discussion is on the keyboard part of the combination.

5.1 Keyboards

The keyboard is the primary means of interacting with the system. In most cases, it is the only way of doing so. Many thousands of characters will be entered in the course of a normal working session. Thus, the keyboard should be tailored for the ease of typing characters. While the previous statement might seem trite, there are a large number of keyboards on the market which are not very good at all for entering characters. Here is a discussion of the various keyboard features and why they are or are not desirable:

N-KEY ROLLOVER is a highly desirable feature. Having it means that you don't have to let go of one key before striking the next. The codes for the keys that you did strike will be sent out only once and in the proper order. (The "n" means that this rollover operations will occur even though every key on the keyboard has been hit.) The basic premise behind n-key rollover is that you will not hit the same key twice in a row. Instead, you will hit a different key first and the reach for that key will naturally pull your finger off of the initial one. However, the timing requirements are quite loose about exactly when your finger has to come off of the first key. Thus, typing errors are reduced. Note that n-key rollover is of no help in typing double letters. Note also that shift keys and the control key are handled specially and are not subject to rollover.

AUTO-REPEAT has both good and bad sides to it. It is useful on a system which does not supply such things in software but its drawbacks (leaning on a key can be deadly) makes it out of place on a system with a sophisticated editor. (If you want a row of "."s, just type "↑U 80 .".)

TOUCH-TYPABILITY is the single most critical feature. It is simply the ability to type the useful characters without moving your fingers from the standard touch-typing position (the "asdf" and "jkl;" keys). As more and more people who use keyboards are touch typists and can thus type at a reasonable clip, they should not be slowed down by having to physically reach their hands out of the basic position. It can take one or two SECONDS to locate and type an out-of-the-way key. (The row above the digits is out-of-the-way, as are numeric key pads and cursor control keys.) One second is from three to ten characters of time (30 - 100 words per minute). Thus, it takes less time in general to type a four or five character command from the basic keyboard than to type one "special" key.

Because of the desire for touch-typability, it is worth at least considering doing away with such keys as "shift lock." They are rarely, if ever, used and the keyboard space that they occupy is in high demand.

Other things which keyboard manufacturers have done can be deadly. Two examples are illustrative. First, the timing on the shift keys can be blown. The result of doing so is that when "Foo" is desired, "FOo," "fOo," and "foo" are more likely to result. The other example is having a small "sweet spot" on each key. Missing this "sweet spot" will cause both the desired and the adjoining key to fire. Thus, striking "i" can cause "io" to be sent.

More generally, the packaging of a keyboard can be important. Sharp edges near the keyboard or too tightly packed keys can cause errors and fatigue.

5.1.1 Special Function Keys and Other Auxiliary Keys

Keyboard manufacturers seem to have decided that a plethora of special keys is more useful than a more general approach. Thus, you can get "insert line" or "cursor up" or--gasp--"PF1". These keys, when pressed, will either do the function that they name, do something totally random, or send a (usually pre-defined) sequence of characters to the computer. For reasons that have been covered already, having the terminal do the named functions is a losing approach. Having them send pre-defined sequences of characters is not much more useful. For example, the "cursor up" key might send ↑[E and your editor has this sequence bound to MoveToEndofSentence. Note that this problem exists even though the editor is fully extensible (i.e., it is not an acceptable solution to rebind the ↑[E command in the editor to MoveUpLine) because the user might still want to use the ↑[E command for its original purpose. This problem can be avoided if the keys are down loadable with a sequence of characters to send. Thus, the editor can tell the "cursor up" key to send, say, ↑P.

Aside from the problems of compatibility with whatever software is being run, the placement of the keys is the worst problem. As has just been stated, keys that are off to one side take too long to hit. Thus, typing is slowed down considerably.

There is yet one more problem. Additional keys are not free and so the number of them that you want to pay for is limited. However, it is desirable to have the ability to specify a large number of functions (i.e., have a large number of codes that can be specified by the user). The number of special keys required grows linearly with the number of codes.

5.1.2 Extra Shift Keys

A more general solution is to provide extra shift keys. These are keys that modify the actions of the other keys. "Shift" and "control" are the two most common examples of such keys. The Teleray 1061 terminal has a "meta" key as an option. This key sets the top (128) bit of the character that is specified. There are thus 256 codes that can be specified instead of the usual 128 from a full ASCII keyboard.

The number of extra shift keys required grows as the log of the number of codes. Thus, 512, 1024, and even 2048 code keyboards are conceivable.

Finding room on the basic keyboard for these extra shift keys is not easy. That is one reason why the removal of the "shift lock" key was suggested earlier. These keys must be on the basic keyboard in order to preserve touch-typability. (It does not take noticeably longer to type the shifted version of a key than the non-shifted version.) The Knight keyboards in use at the MIT Artificial Intelligence Laboratory have several shift keys. They are, unfortunately, located far enough away from the basic keyboard to prevent touch-typability.

5.2 Graphical Input

Another way of interacting with a computer is by means of a graphical input device. The advantage of a graphical input device is that it can reduce the number of commands needed. Such a device is used for pointing at sections of the screen. It is thus possible to specify items there without having to specify the numerical address of the location or a command string to move you there.

5.2.1 How It Can Be Used

A graphical input device is used by thinking of the screen as one menu with the device pointing to one entry. A cursor of some sort is used to provide feedback about which menu item is currently selected. There are usually one or more "flags" that can be specified conveniently from the device. These flags provide control information. One flag is special and it provides "Z-axis" information.

The basic loop is to track the device with the cursor. When the Z-axis flag is entered, the currently selected action is taken. The screen is logically broken up into two or more sections. One section has the text that is being edited. Moving the cursor here provides a convenient way to move the point around; typing a character could cause it to be inserted wherever the cursor is. Other logical screens can specify menus of possible actions to select from. It is thus a very sophisitcated and general way of specifying a position as an argument to a function.

The desired logical screen can be selected by means of the flags or, where the number of flags is limited, by physical position of the cursor on the screen. The Lisp Machine editor and Xerox PARC's Bravo editor both use graphical input devices heavily.

5.2.2 Devices: TSD, Mouse, Tablet, Joystick

There are several types of devices that are either available commercially or experimentally. They shall be discussed in order of usability.

A Touch Sensitive Display (TSD) is just what it sounds like. The screen is covered with a special transparent material that you touch with your finger and it reports the absolute x,y coordinates of where you touched. No "flags" are available until someone can figure out how to track your finger as it brushes the surface as well as when you press more firmly (creating a Z-axis touch). It is the nicest of the devices, although obtaining feedback is hard because your finger covers the most interesting part of the screen.

A mouse is a small box with wheels. It reports the relative movement that you give it (i.e., "he moved me n units up and m units left") as opposed to absolute coordinates ("I am at position x,y"). It can have several flags. It moves along the floor, table, books, legs, or anything else.

A tablet is an absolute version of a mouse (actually, it came first). It can be run with an electronically detected puck (a small box) or a pen. A physical tablet is required for detection and it is usually about 15" x 15" x 1/2". The absolute coordinates are relative to the tablet. There can be several flags for a puck; a pen usually only has Z-axis reporting.

A joystick is a small stick mounted on a couple of potentiometers. It can report either absolute position, first derivative (relative movement) or second derivative. As it is moved small distances, getting good resolution and avoiding "stickiness" and "jumpiness" are hard. It is generally not as nice to use as the others. Flags are usually by means of regular keyboard keys.

Finally, an imaginary but useful device should be considered. That device is a foot-operated mouse. Using your feet rather than your hand to operate the mouse solves one of most nagging problems of any of these devices, which is that your hands must leave the keyboard with the usual and aforementioned results.

These devices all assume a high bandwidth connection to some computer. Such a connection is not practical over, say, 30 cps phone lines. What must be done in that case is to have the device report to the terminal, which moves the cursor around and reports when a flag has been hit. Thus, it is possible to supply the immediate feedback that is necessary. A 30 cps connection would be quite satisfactory for this operation (but probably not satisfactory for the screen refresh that would follow, say, the selection of a menu).

6. The World Outside of Text Editing

Text editors have been used for many things besides editing text and, in the future, they will undoubtedly be used for more diverse things. Here are some examples:

A text editor can be the primary interface to a mail system. Messages can be composed by editing a buffer and sent with a special command. Mail can be read and managed by reading it into a buffer and having special commands to perform such operations as move to the next message and summarize all messages. Having the full power of a text editor available can make such things as undeleting an accidentally deleted message or copying the text of a message that is being replied to quite easy to implement.

A text editor can be the primary interface to the operating system. Command lines can be edited with the full power of the editor before being evaluated. The past record of interaction can be kept and parts of it examined or re-used in new command lines. If the operating system does not have support for advanced terminals, a display editor can offer its interface for use by other programs. Other programs would then take advantage of the terminal independence of the editor. Alternatively, other programs would insert their output into a buffer and the editor would become an entire terminal management system.

A text editor can be used by a debugger. Multiple buffers and multiple windows can be used to examine (perhaps multiple) source files, interact with the debugger, and see the output/input of the program as it runs. In additions, a debugger might take over an extra window or two to do such things as constantly show selected variables.

A text editor can be an interface to a complicated file. For example, an indexed sequential file can be updated by providing editor commands to read and write entries (adding or deleting them can be managed as well). Within the entry, the full power of the editor is available for editing it.

A text editor can provide a smooth interface to the file system. A directory can be read by the editor and "edited" by the user. Files can be deleted or otherwise changed in a smooth manner by merely moving to the file name and giving a command (e.g., "delete").

(All of the preceding are currently subsystems within Multics Emacs. They are enthusiastically accepted by the user community.)

A text editor can be used to examine and--when absolutely necessary--modify object files. It can thus replace various patching programs.

A text editor is an integral part of a word processing system. Such systems often have features like automatic pagination and continual justification (the document in general and the current paragraph in particular are constantly kept right justified by rejustification after each modification). These features exist in the ALTO editor Bravo, written at Xerox PARC as well as a number of the word processing packages supplied currently for micro computers.

A text editor can deal with proportionately spaced fonts as well as fixed with ones. (The redisplay gets a lot more complex.)

The editor can be interfaced with the compiler to incrementally compile and/or check a program.

Here, the principle of "sticky compiling" must be introduced. Assume that a program has been properly compiled. Now, change a statement by deleting a few characters and inserting a few others. The editor/compiler combination should not give an error message even though the program has been temporarily illegal. Rather, it should be quiet until you have either finished entering the new statement or it is clear that you are making a mistake. (Deciding when you have made a mistake can be hard.) The editor/compiler combination is generally also interfaced with a debugger. This trio supplies the essence of an integrated program development system.

In summary, a text editor can be used for a wide variety of things besides editing text. Taking the intended use into account when designing a new system can provide useful feedback and new constraints on the design of the system as a whole.

I. Annotated Bibliography

This bibliography includes many different types of documents. Some of the documents are user manuals for various editors. Others of them describe the implementation of specific editors. Still others discuss language tradeoffs or input/output system interfaces.

They are grouped by the type of editor that they refer to. Each entry is annotated to help place it in perspective. Documents that are marked with "*" are especially valuable or interesting.

6.1 Emacs Type Editors

There are four prinicipal implementations of Emacs type editors and there are enough documents to justify their separate listing.

6.1.1 ITS EMACS

Ciccarelli, Eugene
An Introduction to the EMACS Editor
MIT Artificial Intelligence Laboratory, MIT AI Lab Memo #447,
Cambridge, Massachusetts
January 1978

A primer on the editor's user interface.

*Stallman, Richard M.
EMACS: The Extensible, Customizable, Self-Documenting, Display
Editor
MIT Artificial Intelligence Laboratory, AI Memo #519,
Cambridge, Massachusetts
June 1979

Provides arguments for the Emacs philosophy.

Stallman, Richard M.
Structured Editing with a Lisp
letter in Surveyor's Forum (includes a response by Sanderwall)
Computing Surveys, vol 10 #4, page 505
December 1978

This is a response to the Sanderwall paper (referenced later).

On-line Documentation
MIT-AI: .TECO.; TECORD >
A more detailed command list for TECO
MIT-AI: .TECO.; TECO PRIMER
A primer for TECO
MIT-AI: EMACS; EMACS CHART
A four page command list for Emacs

```
        MIT-AI: EMACS; EMACS GUIDE
                A detailed user interface manual
        MIT-AI: EMACS; EMACS ORDER
                A more detailed command list for Emacs
```

6.1.2 Lisp Machine Zwei

*Weinreb, Daniel L. & Moon, David
The Lisp Machine Manual
MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts

January 1979

The user interface for Zwei.

Weinreb, Daniel L.
A Real-Time Display-Oriented Editor for the Lisp Machine
S.B. Thesis, MIT Electrical Engineering and Computer Science
Department, Cambridge, Massachusetts
January 1979

How Zwei works internally.

6.1.3 Multics Emacs

Greenberg, Bernard S.
Emacs Extension Writer's Guide
Order #CJ52, Honeywell Information Systems, Inc.
(In publication)

How to write extensions.

Greenberg, Bernard S.
Emacs Text Editor User's Guide
Order #CH27, Honeywell Information Systems, Inc.
December 1979

The user interface.

*Greenberg, Bernard S.
"Multics Emacs: an Experiment in Computer Interaction"
Proceedings, Fourth Annual Honeywell Software Conference,
Honeywell Information Systems
March 1980

A summary of MEPAP (referenced below).
Also, MIT-AI: BSG; NMEPAP >

```
Greenberg, Bernard S.
Read-Time Editing on Multics
Multics Technical Bulletin #373
Honeywell Information Systems, Inc., Cambridge, Massachusetts
April 1978


On-Line Documentation:
(by Greenberg, Bernard S.)
        MIT-AI: BSG; LMEPAP >
                Why Lisp was chosen for the implementation
                language
*       MIT-AI: BSG; MEPAP >
                A detailed history of Emacs in general and the
                Multics implementation in specific.
                Very valuable.
        MIT-AI: BSG; R4V >
                A proposal for a terminal independent video
                terminal support package.
        MIT-AI: BSG; TTYWIN >
                A look at the good and bad features of video
                terminals.
```

6.1.4 MagicSix TVMacs

*Anderson, Owen Ted
The Design and Implementation of a Display-Oriented Editor
Writing System
S.B. Thesis, MIT Physics Department, Cambridge, Massachusetts
January 1979

How TVMacs works internally. It concentrates on describing not the editor itself but rather the implementations language: SINE.

Linhart, Jason T.
Dynamic Multi-Window Terminal Management for the MagicSix
Operating System
S.B. Thesis, MIT Electrical Engineering and Computer Science
Department, Cambridge, Massachusetts
June 1980

A video terminal management system. Contains many useful comments on terminal independence and redisplay problems.

6.1.5 Other Emacs

This section covers editors which have the same general user interface as an Emacs (e.g., screen-oriented, similar key bindings) but are not extensible or otherwise fall noticably short of the Emacs philosophy.

Finseth, Craig A.
VINE Primer
Texas Instruments, Inc., Central Research Laboratories, Systems
and Information Sciences Laboratory, Dallas, Texas
August 1979

User interface manual for the complete novice.

Schiller, Jeffrey I.
TORES: The Text ORiented Editing System
revised from S.B. Thesis, MIT Electrical Engineering and
Computer Science Department, Cambridge, Massachusetts
June 1979

On-Line Documentation
CMU-10A: fine.{mss prt}[s200mk50]
User manual for FINE, running at Carnegie-Mellon
University. *Written by Mike Kazar.*

6.2 Non-Emacs Display Editors

Bilofsky, Walter
The CRT Text Editor NED -- Introduction and Reference Manual
Rand Corporation, R-2176-ARPA
December 1977

Irons, E. T. & Djorup, F. M.
A CRT Editing System
Communications of the ACM, vol. 15 #1, page 16
January 1972

Joy, William
Ex Reference Manual; Version 2.0
Computer Science Division, Dept of Electrical Engineering and
Computer Science, University of California at Berkeley
April 1979

Joy, William
An Introduction to Display Editing with Vi
Computer Science Division, Dept of Electrical Engineering and
Computer Science, University of California at Berkeley
April 1979

Kanerva, Pentti
TVGUID: A User's Guide to TEC/DATAMEDIA TV-Edit
Stanford University, Institute for Mathematical Studies in

the Social Sciences
1973

Kelly, Jeanne
A Guide to NED: A New On-Line Computer Editor
The Rand Corporation, R-2000-ARPA
July 1977

Kernighan, Brian W.
A Tutorial Introduction to the ED Text Editor
Technical Report, Bell Laboratories, Murray Hill, New Jersey
1978

MacLeod, I. A.
Design and Implementation of a Display-Oriented Text Editor
Software Practice and Experience, vol. 7 #6, page 771
November 1977

Weiner, P., et. al.
The Yale Editor "E": A CRT Based Editing System
Yale Computer Science Research Report 19
April 1973

Seybold, Patricia B.
TYMSHARE's AUGMENT -- Heralding a New Era, The Seybold
Report on Word Processing
Vol. 1, No. 9, 16pp, ISSN: 0160-9572, Seybold Publications, Inc.,
Box 644, Media, Pennsylvania 19063
October 1978

On-Line Documentation:
SAIL: E.ALS[UP,DOC]
User manual again. Stanford University.

6.3 Structure Editors

Ackland, Gillian M., et al
UCSD Pascal Version 1.5 (Reference Manual)
Institute for Information Systems, University of
California at San Diego

Donzeau-Gouge, V.; Huet, G.; Kahn, G.; Lang, B.; & Levy, J.J.
A Structure Oriented Program Editor: A First Step Towards
Computer Assisted Programming
Res. Rep. 114, IRIA, Paris

April 1975

Teitelbaum, R. T.
The Cornell Program Synthesizer: A Microcomputer
Implementation of PL/CS
Technical Report TR 79-370, Department of Computer Science,
Cornell University, Ithaca, New York

6.4 Other Editors

Benjamin, Arthur J.
An Extensible Editor for a Small Machine With Disk Storage
Communications of the ACM, vol. 15 #8, page 742
August 1972

Talks about an editor for the IBM 1130 written in Fortran.
Not extensible at all.

Bourne, S. R.
A Design for a Text Editor
Software Practice and Experience, vol 1, page 73
January 1971

User manual

Cecil, Moll & Rinde
TRIX AC: A Set of General Purpose Text Editing Commands
Lawrence Livermore Laboratory UCID 30040
March 1977

Deutsch, L. Peter & Lampson, Butler W.
An On-Line Editor
Communications of the ACM, vol. 10 #12, page 793
December 1967

QED user manual

Fraser, Christopher W.
A Compact, Portable CRT-Based Editor
Software Practice and Experience, vol. 9 #2, page 121
February 1970

Front end to a line editor.

Fraser, Christopher W.
A Generalized Text Editor

Communications of the ACM, vol. 23 #3, page 154
March 1980

Applying text editors to non-text objects,

Hansen, W. J.
Creation of Hierarchic Text with a Computer Display
Ph.D. Thesis, Stanford University
June 1971

Kai, Joyce Moore
A Text Editor Design
Department of Computer Science, Univ of Ill at Urbana-Champaign,
Urbana, Illinois
July 1974

Describes both internals and externals on the editor. However, the design is a poor one.

Kernighan, Brian W. & Plauger, P. J.
Software Tools
Addison-Wesley, Reading, Massachusetts
1976

This book has a chapter which leads you by the hand in implementing a simple line editor in RatFor.

*Roberts, Teresa L.
Evaluation of Computer Text Editors
Systems Sciences Laborary, Xerox PARC
November 1979

A comparative evaluation of four text editors. Quite well done. Unfortunately, it does not include Emacs (it uses DEC TECO instead).

Sanderwall, Erik
Programming in the Interactive Environment: the Lisp Experience
Computing Surveys, vol. 10 #1, page 35
March 1978

Talks about the editor for InterLisp.

Sneeringer, James
User-Interface Design for Text Editing: A Case Study
Software Practice and Experience, Vol 8, page 543
1978

User manual and a discussion of user interface concepts.

Teitelman, Warren
InterLisp Reference Manual
Xerox Palo Alto Research Center, Palo Alto, California
October 1978

How to use the InterLisp (non-display) structure editor.

van Dam, Andries & Rice, David E.
On-Line Text Editing: A Survey
Computing Surveys, Vol. 3 #3, p. 93
September 1971

Contains a general introduction to the problems of text editing. Out-dated technology, though.

II. Some Implementations of Emacs Type Editors

This is a partial list and is intended to provide a general guide and not a comprehensive list.

	Name	System	Implementation Language
	TECO	ITS	Midas (assembler)
Full Emacs			
	EMACS	ITS	TECO
	Emacs	Multics	Lisp
	Emacs	Tops-20	TECO
	TVMacs	MagicSix	Sine
	Zwei	Lisp Machine	Lisp
Partial Emacs			
	FINE	Tops-10	Bliss
	MINCE	CP/M	C
	otv	MagicSix	PL/1
	Tores	UNIX	C
	VINE	VAX/VMS	Fortran

III. Partial Emacs Command List

This list is of the command set that is generally common to all of the full Emacses. Specific command bindings can and do vary from implementation to implementation. This list is not complete, nor can it be as commands are constantly being added and changed.

Command designations reflect both the name and the manner in which they are entered. For example, the C-f command is named "control f" and it is entered by typing the ↑F character. Most of the C- commands can be given directly from an ASCII keyboard. Escapes are provided for those that are not. The M-a command is named "meta a" and it is entered from an ASCII keyboard by typing the <esc> key and then the command. Thus, M-a is given by typing <esc> a and M-C-a (or C-M-a) by typing <esc> ↑A.

C-@	place the mark at the point
C-a	move to the beginning of the current line
C-b	move backward one character
C-c	a prefix for control-meta commands. see below
C-d	delete the following character
C-e	move to the end of the current line
C-f	move forward one character
C-g	abort: abort execution of the current command and return to the edit loop
C-h	same as C-b
C-i	insert <tab>
C-j	insert <newline>; insert <tab>
C-k	delete the text to the end of the current line; if at the end of the line, delete the newline character; push deleted text onto the kill buffer
C-l	rebuild the display from scratch
C-m	insert <newline>
C-n	move down one line staying in as nearly the same horizontal position as possible
C-o	insert <newline>; move backward one character
C-p	move up one line staying in as nearly the same horizontal position as possible
C-q	insert the following character as typed
C-r	search for a string before the point; see C-s for details
C-s	search for a string after the point. There are lots of things that you can do
	typing characters builds up the search string
	<del> deletes the previous character
	C-s search for the next occurrence of the string
	C-r search for the previous occurrence
	C-g abort
	<alt> terminate search; if the string is null, the previous string is used
C-t	interchange the characters on each side of the point, leaving the point after the second one; if at the end of a line, interchange the previous two characters
C-u	universal argument.

```
                There are two forms
          C-u C-u .... <command>  do <command> 4, 16, 64, 256, ...
                times depending upon the number of C-us.
          C-u <integer> <command> do <command> <integer> times.
          (e.g., C-u 3 5 C-f means to C-f 35 times)
C-v       move the bottom of the current screen to the top of the
                screen
C-w       delete the text between the point and the mark; push
                the deleted text onto the kill buffer
C-x       a prefix for control-x commands.  see below
C-y       copy the top item from the kill buffer to the point;
                place the mark at the beginning of the
                block and the point at the end
C-z       return to superior
C-[       a prefix for meta commands.  see below
C-\       a prefix for meta commands.  see below
C-]
C-↑       a prefix for control commands.  See this list
C-_

 !"#$%&'()*+,-./          insert themselves
0123456789
:;<=>?@
A..Z
[ ]↑`_
a..z
{|}~

bs,back space
          same as C-h
tab       same as C-i
lf,line feed
          same as C-j
cr,carriage return,return
          same as C-m
esc,escape
          same as C-[  (the <alt> key)
del,delete,rubout
          delete the previous character

C-<alt> you are now typing at whatever is running the editor

C-%       ask for the old string, then the new one and replace
          all occurrences of the old with the new
C-/       give help
C-<       place mark at the beginning of the buffer
C->       place mark at the end of the buffer
C-?       give help

C-x C-b print a list of all buffers and associated information
C-x C-d display the current directory
C-x C-f ask for the name of a file and read it into a buffer
                whose name is derived from the filename; if
```

```
                    there is a conflict with an existing
                    buffer, you are asked for a name to use
C-x C-i indent the region
C-x C-l convert the region to lower case
C-x C-o delete the blank lines around the
                    point
C-x C-p move to the top of the current screen; place the mark
                    at the end of the current screen
C-x C-r ask for the name of a file and read it into the
                    current buffer
C-x C-s write out current buffer to the current filename if it
                    has been modified
C-x C-u convert the region to upper case
C-x C-w ask for the name of a file and write the buffer to
                    that file
C-x C-x exchange point and mark
C-x 1   use one window
C-x 2   use two windows
C-x 3   use two windows and stay in the first
C-x =   print where you are in the buffer
C-x A   ask for the name of a buffer; append the region to that
                    buffer
C-x B   ask for the name of a buffer and put you there
C-x D   edit directory
C-x F   set the fill column to the horizontal position
C-x I   run INFO
C-x M   send mail
C-x O   in two window mode go to other window
C-x R   read mail
C-x ↑   grow window by
C-x a   same as C-x A
C-x b   same as C-x B
C-x d   same as C-x D
C-x f   same as C-x F
C-x i   same as C-x I
C-x m   same as C-x M
C-x o   same as C-x O
C-x r   same as C-x R

M-"     ditto. copy the word directly above onto this line
M-%     QueryReplace. ask for an old string and a new string
                    At each occurrence of the old string, it
                    is displayed and you are asked for a command
        <sp>        replace and go on
        <del>       don't replace and go on
        ,           replace and wait
        .           replace and exit
        <alt>       exit
        ↑           return to previous old string (jump to mark)
        C-w         delete old string and enter C-r recursively
        C-r         normal edit, but recursively invoked
        C-l         redisply screen
        !           do not ask any more
```

```
                   there is a conflict with an existing
                   buffer, you are asked for a name to use
C-x C-i indent the region
C-x C-l convert the region to lower case
C-x C-o delete the blank lines around the
                   point
C-x C-p move to the top of the current screen; place the mark
                   at the end of the current screen
C-x C-r ask for the name of a file and read it into the
                   current buffer
C-x C-s write out current buffer to the current filename if it
                   has been modified
C-x C-u convert the region to upper case
C-x C-w ask for the name of a file and write the buffer to
                   that file
C-x C-x exchange point and mark
C-x 1   use one window
C-x 2   use two windows
C-x 3   use two windows and stay in the first
C-x =   print where you are in the buffer
C-x A   ask for the name of a buffer; append the region to that
                   buffer
C-x B   ask for the name of a buffer and put you there
C-x D   edit directory
C-x F   set the fill column to the horizontal position
C-x I   run INFO
C-x M   send mail
C-x O   in two window mode go to other window
C-x R   read mail
C-x ↑   grow window by
C-x a   same as C-x A
C-x b   same as C-x B
C-x d   same as C-x D
C-x f   same as C-x F
C-x i   same as C-x I
C-x m   same as C-x M
C-x o   same as C-x O
C-x r   same as C-x R

M-"     ditto. copy the word directly above onto this line
M-%     QueryReplace. ask for an old string and a new string
                   At each occurrence of the old string, it
                   is displayed and you are asked for a command
        <sp>       replace and go on
        <del>      don't replace and go on
        ,          replace and wait
        .          replace and exit
        <alt>      exit
        ↑          return to previous old string (jump to mark)
        C-w        delete old string and enter C-r recursively
        C-r        normal edit, but recursively invoked
        C-l        redisply screen
        !          do not ask any more
```

M-(	nsert "()"; leave point between them
M-<	move to the the beginning of the current buffer
M->	move to the end of the current buffer
M-?	help
M-A	move to the beginning of the current sentence
M-B	move backward one word
M-C	capitalize the following word
M-D	delete the following word; push deleted text onto the kill buffer
M-E	move to the end of the current sentence
M-F	move forward one word
M-G	fill text in the region
M-H	move to the beginning of the current paragraph; place the mark at the end of the current paragraph
M-L	convert the following word to lower case
M-Q	fill the current paragraph (make each line as long as possible); C-u M-Q means do justify (same, but make right margin even)
M-S	center the current line on the screen
M-T	interchange the adjoining words, leaving the point after the right hand word
M-U	convert the following word to upper case
M-V	move the top of the current screen to the bottom of the screen
M-W	push a copy of the region onto the kill buffer
M-X	ExecuteCommand
M-Y	(after C-Y) delete yanked text and yank previous kill buffer entry
M-[	move to the beginning of the current paragraph
M-\	delete the <sp> and <tab>s around the point
M-]	move to the end of the current paragraph
M-a	same as M-A
M-b	same as M-B
M-c	same as M-C
M-d	same as M-D
M-e	same as M-E
M-f	same as M-F
M-g	same as M-G
M-h	same as M-H
M-l	same as M-L
M-q	same as M-Q
M-s	same as M-S
M-t	same as M-T
M-u	same as M-U
M-v	same as M-V
M-w	same as M-W
M-x	same as M-X
M-y	same as M-Y
M-<del>	delete the previous word; push deleted text onto the kill buffer
C-M-)	move up one level of list structure backward

C-M-(	move up one level of list structure forward
C-M-A	move to the beginning of the current defun
C-M-B	move backward one S-expression
C-M-E	move to the end of the current defun
C-M-F	move forward one S-expression
C-M-G	format the current S-expression
C-M-H	move to the beginning of the current S-expression; place the mark at its end
C-M-K	delete the following S-expression; push the deleted text onto the kill buffer
C-M-O	move the rest of this line vertically down, inserting <tab>s and <sp>s as needed
C-M-T	interchange the adjoining S-expressions, leaving the point after the following S-exprssion
C-M-W	the following delete-and-push will be part of the current entry in the kill buffer
C-M-<del>	delete the preceeding S-expression; push the deleted text onto the kill buffer