UO-LISP NEWSLETTER

July 1985 Vol. 2 No. 3

Hard on the heels of the last late newsletter is this Summer issue. We switch from programming with objects to customizing the LSED screen editor by including the complete source code for an EMACS style interface.

Feature: EMACS Style Interface for LSED

LSED is a completely user programmable editor. The system provides the basic interface functions and the user configures the system to perform in whatever manner desired. This can take the form of rebinding the control character sequences to perform different functions thus permitting the special keys on some systems to operate correctly. Alternatively, the system can be configured to look like some other editor. This is particularly useful if you are used to something like Wordstar or Emacs.

The user's guide provides some simple examples of rebinding key sequences for different terminal types. In this exposition we examine a complete rework of the editor to make it look like the popular Emacs editor.

There are 4 undocumented functions that need to be examined for this effort. We include them as a bonus to newsletter subscribers.

(eFARM CHAR:integer KEYMAP:alist REPT:integer):tri-boolean

Type: EXPR.

eFARM calls the function associated with character CHAR based on the current KEYMAP. KEYMAP is the association list formed by the BIND function. REPT is the number of times that the command should be repeated. If the command succeeds, the value of the function OK is returned, if it fails, the value of the function FAIL is returned and if the command is to terminate LSED execution, the value of TERM is returned (these are checked for by the OKP, FAILP and TERMP functions). If CHAR is the first of a multiple character command, more characters will be read by eFARM to complete the command.

(eGET):integer

Type: EXPR.

eGET returns the next "raw" character from whatever input is currently selected. This includes keyboard input, input from a keyboard macro, and input from the file copy on the screen. eGET should be used for any non-echoing input of characters.

(eGETV N:integer):{list,tri-boolean}

Type: EXPR.

eGETV returns a list of characters for line N of the file being edited. It does all appropriate swapping to and from disk files and so on. It returns FAIL if there is no such line in the file.

(eREAD MSG:string):list

Type: EXPR.

eREAD causes MSG to appear in the middle window and to return a list of characters that the user types following this prompt. eREAD should be called for any user input of text information.

Emacs is more than a set of keybindings and extensions. There are many additional functions not bound to any keys that are supported. The following is about 1/3 of the total available functions in the Unix EMACS system. The following text contains very brief descriptions of functions and bindings that follow in boxes.

To assure that the file we are creating is compilable without loading LSED and the screen driver, we declare global variables we need from the basic system.

```
% Some globals from LSED and the terminal drivers.
(GLOBAL '(
    !#C           % The current key map.
    TERM!-MAXX    % The last screen column.
    TERM!-MAXY    % The top most row (starting at 0).
    !#FY          % The current file line number.
    !#FX          % The rest of the current line.
    !#SX          % Current screen column number.
    !#SY          % Current screen row number.
   ))
```

The first function allows us to have multiple keymaps: the mapping between control key sequences and functions that do things. This function would enable us to switch between the Emacs mode and the regular LSED mode, invaluable during debugging the package. We will call the standard key map UO!-LISP and the EMACS set EMACS. We switch to the (currently empty) EMACS key map before we start rebinding keys.

```
% Keep around the current key map name.
(GLOBAL '(!#KEYMAPNAME))
(SETQ !#KEYMAPNAME 'UO!-LISP)

(DE USEKEYMAP (name)
% Switch to keymap name. The current command list is saved
% under its name (in !#KEYMAPNAME) and the new keymap is
% selected.
  (PUT !#KEYMAPNAME 'LSEDKEYMAP !#C)
  (SETQ !#C (GET name 'LSEDKEYMAP))
  (SETQ !#KEYMAPNAME name))

(USEKEYMAP 'EMACS)        % Switch to emacs keymap.
```

We now work through the control keys, binding them to their appropriate functions and defining new ones when there are no equivalents in the base system. Recall that BIND has two arguments, the first a list of the ASCII codes of the control characters that trigger the evaluation of the second argument, the name of a function. The first set, ^A through ^I (tab), are bound to built-in functions. LSED doesn't work well with tabs, so we use the tab key to evaluate expressions in the top window, something that basic Emacs has no command for.

```
(BIND '(1) 'BOL)          % ^A beginning of line
(BIND '(2) 'BACKC)        % ^B backwards character
(BIND '(3) 'QUITNS)       % ^C quit without saving
(BIND '(4) 'DELFC)        % ^D delete forward character
(BIND '(5) 'EOL)          % ^E end of line
(BIND '(6) 'FORC)         % ^F forward character
(BIND '(8) 'DELBC)        % ^H backwards character
(BIND '(9) 'EEXP)         % tab eval expression
```

The function bound to ^J (line feed) has no counterpart in LSED. The line feed key operates like the return key except that it automatically indents the next line the same number of blanks that the previous line has. We accomplish this by retrieving the current line of characters (using eGETV and the current line number !#FY). We insert a carriage return and as many blanks as we find in the previous line. The LBLS function counts the number of blanks at the beginning of the line.

```
(BIND '(10) 'NLIND)       % <lf> new line and indent.
(DE NLIND ()
% Terminate the current line, indent the next to where
% ever this one starts.
  (PROG (cl)
     (SETQ cl (eGETV !#FY))
     (CINS 13)
     (FOR (FROM i 1 (LBLS cl)) (DO (CINS 32)))
     (RETURN (OK))))

(DE LBLS (l)
% Returns the number of blanks at the beginning of line l.
  (IF (NULL l) THEN 0
   ELSEIF (EQCAR l 32) THEN (ADD1 (LBLS (CDR l)))
   ELSE 0))
```

LSED normally irretrievably expunges deleted text from the file. The kill to end of line command (connected to ^K) places deleted text in a special buffer for retrieval at a later date. The command takes the part of the current text line following the cursor, erases it from the screen, and appends it to the kill buffer variable !#KBUFFER. The function also intercepts subsequent ^K's and causes their text to be added to the buffer. The COPYONE function copies the first element of the tail of the list as we destructively replace the CAR of this element with a

carraige return (13) character to terminate the line. This is an important fact to remember when constructing LSED functions: all lines must end with a carriage return.

```
(BIND '(11) 'KEOL)          % ^K kill to end of line
(GLOBAL '(!#KBUFFER))
(DE KEOL ()
% Delete the rest of the current line (do this by RPLACing).
% Append all this stuff to the !#KBUFFER buffer and keep
% doing so as long as ^K is pressed. Note that if we are
% already at the end of the line, don't add to kill buffer,
% but just do a DELFC.
  (PROG (c)
     (SETQ !#KBUFFER NIL)
11   (IF (EQUAL !#FX '(13)) THEN (DELFC)
      ELSE (SETQ !#KBUFFER (NCONC !#KBUFFER
                                  (LIST (COPYONE !#FX))))
           (CLEAR!-EOL)
           (RPLACA !#FX 13)
           (RPLACD !#FX NIL))
     (SETQ c (eGET))
     (IF (EQUAL c 11) THEN (GO 11)
                      ELSE (RETURN (eFARM c !#C NIL)))))

(DE COPYONE (l)
% Creates a new list l with the first element replaced.
  (CONS (CAR l) (CDR l)))
```

The ENTER (or sometimes RETURN) key is bound to SELF. SELF causes the character to be entered as is into the file, in this case, causing a new line. The OPENL function (^O) is similar in flavor except that it backs up before the inserted character efectively opening a blank line (or with the remainder of the current one) following the one the cursor is on.

```
(BIND '(12) 'REDRAW)        % ^L redraw screen
(BIND '(13) 'SELF)          % ^M new line
(BIND '(14) 'DOWNL)         % ^N next line
(BIND '(15) 'OPENL)         % ^O open new line
(DE OPENL ()
% Insert a new line at the current position but leave the
% cursor where it is by backing up one character.
  (CINS 13)
  (BACKC))
```

The ^Q key causes one more character to be read and inserted as is into the input file. This is most useful for entering control characters (such as ^A into LISPTEX document files).

```
(BIND '(16) 'UPL)          % ^P previous line
(BIND '(17) 'QC)           % ^Q quote next character
(DE QC ()
% Insert next character as is into the file.
  (CINS (eGET)))
```

The transpose character function TCHAR switches the next two characters in the file if there are any and one of them is not a carriage return. The cursor is left in its original position. Note that CINS moves the cursor ahead

```
(BIND '(19) 'FIND)         % ^S find forward
(BIND '(20) 'TCHAR)        % ^T transpose next 2 characters
(DE TCHAR ()
% Read the next two characters, but quit if they
% are anything funny.
  (PROG (c1 c2)
     (SETQ c1 (CURC))
     (IF (FAILP (FORC)) THEN (BACKC) (RETURN (FAIL)))
     (SETQ c2 (CURC))
     (IF (OR (EQN c1 13) (EQN c2 13)) THEN
        (BACKC) (RETURN (FAIL)))
     (BACKC) (DELFC) (DELFC)
     (CINS c2) (CINS c1) (BACKC)
     (RETURN (BACKC))))
```

When we implemented LSED we decided that the 80x24 character screens common on most small computers were not sufficient to support multiple window editing. Instead we adopted the strategy of editing files simultaneously. The following commands implement this facility and switching between the files. Note that some of these commands require regular characters as modifiers for ^X (for example ^X E). In this case we provide a binding for both upper and lower case versions of the character.

```
(BIND '(21) 'REPT)          % ^U repeat next command
(BIND '(22) 'FORP)          % ^V forward page

(BIND '(24 3) 'QUITNS)      % ^X ^C exit emacs
(BIND '(24 5) 'ECOM)        % ^X ^E middle window eval
(BIND '(24 6) 'SAVEX)       % ^X ^F save and exit
(BIND '(24 9) 'INSF)        % ^X ^I insert file
(BIND '(24 17) 'SAVEF)      % ^X ^Q save but no exit
(BIND '(24 22) 'VISITF)     % ^X ^V visit file
(BIND '(24 26) 'SW)         % ^X ^Z shrink window
(BIND '(24 40) 'DKM)        % ^X (  define keyboard macro
(BIND '(24 41) 'EKM)        % ^X )  end of keyboard macro
(BIND '(24 69) 'KM)         % ^X E  execute keyboard macro
(BIND '(24 101) 'KM)
(BIND '(24 78) 'OTHERF)     % ^X N  other file
(BIND '(24 110) 'OTHERF)
(BIND '(24 80) 'OTHERF)     % ^X P  other file
(BIND '(24 112) 'OTHERF)
```

The yank from kill buffer command copies the kill buffer into the file at the current position. This feature is normally used as in cut and paste editing or to move blocks of text between two files. Since the kill buffer is not destroyed by the command it can be inserted in more than one place or carried between files. The FINALLY clause removes the final carriage return ending the last line of the kill buffer.

```
(BIND '(25) 'YFKB)          % ^Y yank from kill buffer.
(DE YFKB ()
% Yank the stuff from the kill buffer and place in the file.
% This kludgy version updates the screen. This probably
% shouldn't happen unless the kill buffer is small. The kill
% buffer is not cleared.
  (FOR (IN l !#KBUFFER)
       (DO (FOR (IN c l) (DO (CINS c))))
       (FINALLY (DELBC))
       (RETURNS (OK))))
```

The LTTOP function (bound to escape !) rolls the line the cursor is on (!#SY on the screen) to the top of the screen. It does this by rolling up the screen a line at a time until either the end of file is encountered (ROLLUP returns FAIL) or the line is at the top of the screen. Note that ROLLUP causes !#SY to be

incremented for every successful roll.

```
(BIND '(26) 'ROLLUP)        % ^Z roll screen up
(BIND '(27 3) 'QUITNS)      % esc ^C quit no save
(BIND '(27 27) 'ECOM)       % esc esc execute expression
(BIND '(27 33) 'LTTOP)      % esc ! line to top of window
(DE LTTOP ()
% Move the current line to the top of the window by
% scrolling.
  (PROG ()
loop (IF (EQN !#SY TERM!-MAXY) THEN (RETURN (OK)))
     (IF (FAILP (ROLLUP)) THEN (RETURN (FAIL)))
     (GO loop)))
```

We now branch into some word processing functions, those dealing with words, and sentences. For example, the esc ^ command causes the cursor to skip over all non-printing characters and then invert the case of all alphabetic characters until some non-alphabetic character is encountered. The function CINV inverts the case of a single character and returns NIL when a non-alphabetic is encountered. CWINV scans over control characters and blanks and then the word inverting case. Notice that one must always check for FAILP of a cursor movement operation. Not doing this will generally cause an infinite loop when the cursor reaches end of file.

```
(BIND '(27 44) 'BOP)      % esc , beginning of window
(BIND '(27 46) 'EOP)      % esc . end of page
(BIND '(27 60) 'BOF)      % esc < beginning of file
(BIND '(27 62) 'EOF)      % esc > end of file
(BIND '(27 94) 'CWINV)    % esc ^ case word invert
(DE CINV (c)
% Invert the case of character c. Returns NIL if c is not
% an alphabetic character.
  (IF (AND (GEQ c 65) (LEQ c 90)) THEN (PLUS c 32)
   ELSEIF (AND (GEQ c 97) (LEQ c 122)) THEN (DIFFERENCE c 32)
   ELSE NIL))

(DE CWINV ()
% Invert the case of a word.
  (PROG (c)
11   (IF (GREATERP (CURC) 32) THEN (GO 12))
     (IF (FAILP (FORC)) THEN (RETURN (FAIL))
      ELSE (GO 11))
12   (IF (SETQ c (CINV (CURC))) THEN
        (DELFC) (CINS c)
        (IF (FAILP (FORC)) THEN (RETURN (FAIL))
                           ELSE (BACKC) (GO 12)))
     (RETURN (OK))))
```

Moving backwards a word is a bit more difficult as the CURC function alwyas returns the next character in the input line. As before we bind both esc B and esc b to the command. This command checks for failure of BACKC so that the result of backing up to the beginning of file des not result in an infinite loop.

```
(BIND '(27 66) 'BW)       % esc B backwards word
(BIND '(27 98) 'BW)
(DE BW ()
% Move backwards a word.
  (PROG ()
11   (IF (MEMBER (CURC) '(32 13)) THEN
       (IF (OKP (BACKC)) THEN (GO 11)
                         ELSE (RETURN (FAIL))))
12   (IF (MEMBER (CURC) '(32 13)) THEN (RETURN (OK))
     ELSEIF (OKP (BACKC)) THEN (GO 12)
                          ELSE (RETURN (FAIL)))))
```

Delete forward word is a function in the same fashion.

Notice that many of the key bindings for commands operating on words are bound to key sequences that are difficult to remember. If you are in the habit of moving the cursor through words and sentences you might consider binding these sequences to single character codes or to the special function keys.

```
(BIND '(27 68) 'DFW)      % esc D delete next word
(BIND '(27 100) 'DFW)
(DE DFW ()
% Delete all the blanks up to and including the next word.
  (PROG ()
l1    (IF (MEMBER (CURC) '(32 13)) THEN
       (IF (OKP (DELFC)) THEN (GO l1)
                         ELSE (RETURN (FAIL))))
l2    (IF (MEMBER (CURC) '(32 13)) THEN (RETURN (OK))
       ELSEIF (OKP (DELFC)) THEN (GO l2)
                            ELSE (RETURN (FAIL)))))
```

Forward sentence and forward word are more of the same.

```
(BIND '(27 69) 'FS)       % esc E forward sentence
(BIND '(27 101) 'FS)
(DE FS ()
% Move forward a sentence. A sentence is terminated with a
% . ! or ? and followed by a blank, EOF, or end of line.
  (PROG ()
s1    (IF (MEMBER (CURC) '(46 33 63)) THEN (GO s2)
       ELSEIF (OKP (FORC)) THEN (GO s1)
       ELSE (RETURN (FAIL)))
s2    (IF (FAILP (FORC)) THEN (RETURN (OK))
       ELSEIF (MEMBER (CURC) '(32 13)) THEN (RETURN (OK))
       ELSE (GO s1))))

(BIND '(27 70) 'FW)       % esc F forward word
(BIND '(27 102) 'FW)
(DE FW ()
% Move forward over blank characters and A-Z characters to
% the end of this word.
  (PROG ()
l1      (IF (MEMBER (CURC) '(32 13)) THEN
          (IF (OKP (FORC)) THEN (GO l1)
                           ELSE (RETURN (FAIL))))
l2      (IF (NOT (MEMBER (CURC) '(32 13))) THEN
          (IF (OKP (FORC)) THEN (GO l2)
                           ELSE (RETURN (FAIL))))
        (RETURN (OK))))
```

Delete backward word is a bit different than just backing up as we don't want to delete extra characters. Changing the case of a word to lower is, of course, the analog of changing it to upper case.

```
(BIND '(27 72) 'DBW)      % esc H delete backward word
(BIND '(27 104) 'DBW)
(DE DBW ()
% Delete the blanks before and the previous word.
  (PROG ()
11    (IF (MEMBER (CURC) '(32 13)) THEN
          (DELFC)
          (IF (OKP (BACKC)) THEN (GO 11)
                            ELSE (RETURN (FAIL))))
12    (IF (MEMBER (CURC) '(32 13)) THEN (RETURN (OK)))
      (DELFC)
      (IF (OKP (BACKC)) THEN (GO 12)
                        ELSE (RETURN (FAIL)))))

(BIND '(27 76) 'CWL)      % esc L case word lower
(BIND '(27 108) 'CWL)
(DE CWL ()
% Lower the case of the next word. Skip over any blanks in
% the way.
  (PROG (c)
11    (IF (MEMBER (CURC) '(32 13)) THEN
       (IF (OKP (FORC)) THEN (GO 11) ELSE (RETURN (FAIL))))
12    (SETQ c (CURC))
      (IF (AND (GEQ c 65) (LEQ c 90)) THEN
          (DELFC) (CINS (PLUS c 32))
       ELSEIF (MEMBER c '(32 13)) THEN (RETURN (OK))
       ELSEIF (OKP (FORC)) THEN (GO 12)
       ELSE (RETURN (OK)))))
```

The replace command requires collecting two strings and then performing replacements throughout the rest of the file. We first check the strings for the presence of ^G. We use this character to signal the command to be aborted. Once both strings have been read, the FND function locates the first occurrence of the string. If none are found the routine returns success or failure based on the number of replacements performed. If an occurrence of the !#ROLD string is found, its characters are deleted and replaced with those of the new string and the search and replacement continues.

```
(BIND '(27 82) 'RPL)      % esc R replace
(BIND '(27 114) 'RPL)
(GLOBAL '(!#ROLD !#RNEW))
(DE RPL ()
% Search for the string !#ROLD and replace it them with
% !#RNEW. Display the count of replacements in the
% message window.
  (PROG (rplcnt)
     (SETQ rplcnt 0)
     (SETQ !#ROLD (eREAD "Search for:"))
     (IF (MEMBER 7 !#ROLD) THEN (MSG '("aborted"))
                                (RETURN (FAIL)))
     (SETQ !#RNEW (eREAD "Replace with:"))
     (IF (MEMBER 7 !#RNEW) THEN (MSG '("aborted"))
                                (RETURN (FAIL)))
loop (IF (FAILP (FND !#ROLD)) THEN
       (IF (ZEROP rplcnt) THEN (MSG '("no replacements"))
                               (RETURN (FAIL))
        ELSE (MSG (LIST rplcnt "occurrences replaced"))
             (RETURN (OK))))
     (FOR (FROM i 1 (LENGTH !#RNEW)) (DO (DELBC)))
     (FOR (IN c !#ROLD) (DO (CINS c)))
     (SETQ rplcnt (ADD1 rplcnt))
     (GO loop)))
```

The final commands and functions are mere repetitions of earlier ones.

```
(BIND '(27 85) 'CWU)      % esc U case word upper.
(BIND '(27 117) 'CWU)
(DE CWU ()
% Convert the next word to all upper case.
  (PROG (c)
11   (IF (MEMBER (CURC) '(32 13)) THEN
       (IF (OKP (FORC)) THEN (GO 11) ELSE (RETURN (FAIL))))
12   (SETQ c (CURC))
     (IF (AND (GEQ c 96) (LEQ c 122)) THEN
        (DELFC) (CINS (DIFFERENCE c 32)) (GO 12)
      ELSEIF (MEMBER c '(32 13)) THEN (RETURN (OK))
      ELSEIF (FAILP (FORC)) THEN (RETURN (FAIL))
      ELSE (GO 12))))

(BIND '(27 86) 'BACKP)    % esc V backwards page
(BIND '(27 118) 'BACKP)
(BIND '(27 88) 'ECOM)     % esc X execute lisp in mini
(BIND '(27 120) 'ECOM)
(BIND '(27 90) 'ROLLDN)   % esc Z roll down
(BIND '(27 122) 'ROLLDN)
(BIND '(27 127) 'DELBC)   % del delete backward character
```

To compile this batch of functions you must first load the MACROS package. To run the system, you must first load LSED and then the emacs file.

In the next Issue:

The next issue of the UO-LISP newsletter will present details of the upcoming release of version 2.17 for Z80 CP/M systems, and version 3.2 for IBM PC users. The feature article will be a rule based expert system for generating musical scores.