A Double-Layered Text Editor*

HIROYASU KAKUDA** and TAKASHI TSUJI***

A **double-layered** text editor, called KE (=Kernel Editor), in which the user can define and redefine commands by writing "**microprograms**", is described. KE itself is in fact a skeleton. It contains a processor of a modest string manipulation language. It is by providing programs in this language, or "microprograms", defining editor commands that KE can be made to behave as a useful text editor. (Non-text editing applications are also possible.) The main advantages are that modification and experimentation in editor commands are easy, and proper insulation between the external and internal structures is provided. Experiences with developing and using KE are also discussed.

1. Introduction

It has been pointed out that tools tend to shape the users [1]. This observation, often made about programming languages, applies equally well to text editors. The user of a text editor tends to find his editor to be "the best in the world", or at least sees less faults in his than do the users of other text editors. For this reason it is debatable whether the text editors currently in use are as good as they can be.

How can we find faults in, and improve our own text editors in spite of the mental inertia we might have? One good way is to design an editor in which experimentation and modification can be done easily. By such an organization we can (1) encourage ourselves to think about possible alternatives; and (2) collect ideas from a user community as well as from the designer himself. Moreover, we could (3) customize the editor to meet special needs of special users if the occasion demands.

This paper describes a text editor called KE (=Kernel Editor) developed with the above points in mind. It has a double-layered organization. In the internal structure, it is an interactive processor of a modest string manipulation language, hereafter called MCL (=Micro Command Language). In a broad way the language is similar to SNOBOL4[2] and SL5[3]. It has a pattern matching feature with a success/failure mechanism, and has modern control structures. The (external) editor commands invoke corresponding programs written in MCL. The user can define a new command, or even redefine existing ones by writing a program in the language. KE has no editor commands initially. KE itself is not a text editor but a skeleton. By compiling and loading suitable MCL programs, KE can be made to serve as a useful text editor.

Our organization differs from that of some conventional text editors such as TECO[4]. In TECO, the users can define a macro command in terms of existing commands accessible from ordinary users. In contrast, the MCL programs in KE are separate from ordinary commands. They are not command sequences but programs having their own constructs.

Independently developed, but closely related ideas are found in Stallman's EMACS[5]. A huge TECO-based program attempts, so to speak, to fill every need any user might have. In contrast, KE is a practical tool everyone can afford.

In Chapter 2, we describe the user's view of our editor. Chapter 3 sketches a typical set of text editing commands implementable on KE. Chapter 4 gives more details of the MCL language, and illustrates how MCL programs can implement editor commands. An implementation of KE itself is described in Chapter 5. Chapter 6 describes our experiences, and finally, Chapter 7 attempts to defend our approach. For a complete grammatical description of MCL, a complete description of the set of commands sketched in Chapter 3, and a full listing of the MCL programs implementing the set of commands, see [6], of which this paper is a condensed version.

2. An Overview

2.1 The Overall Structure

KE has a number of working spaces called **internal files**. Files managed by the operating system are called **external files**. Texts in the internal and external files can be transmitted back and forth. All the text editing operations manipulate texts in the internal files. Besides, there are **control program files**, which hold "microprograms"*.

*Here, we are using the word "microprogram" as a metaphor. Our "microprogram" controls the action of the MCL monitor, a piece of software rather than hardware. However, the basic idea of having a substructure is same as in conventional microprograms directly executed by hardware. In the sequel, we will be using this word in this metaphorical sense, and leave out the quotes in some cases.

*A preliminary description in Japanese of part of this research is found in T. Tsuji and H. Kakuda: A Text Editor Based on Micro Instructions, Proc. of the 18th Annual Programming Symposium at Hakone, January 1977, pp. 20–27.

**Department of Information Science, Tokyo Institute of Technology.

***Now with Institute of Information Sciences, University of Tsukuba. The overall structure of KE, the basic design of MCL, and the implementation of the MCL compiler are due jointly and equally to the authors. The rest of the work was done by the first author.

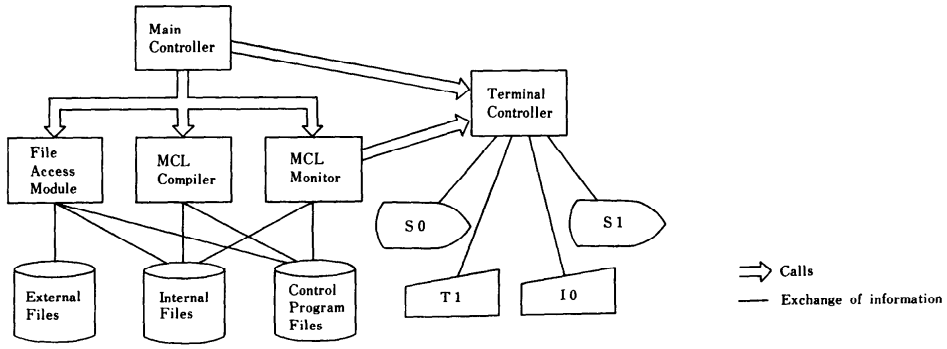

Fig. 1 The conceptual structure of KE.

Fig. 1 shows the conceptual structure of KE. The information transfers between the internal and the external files, as well as those between the external and the control program files, are controlled by a **file access module**. An **MCL compiler** takes programs from one of the internal files, and writes compiled "microprograms" onto some of the control program files. An **MCL monitor** executes (interprets) the microprograms in the control program files. A **main controller** gives control to the above three modules as appropriate. A **terminal controller** provides an interface to a terminal. It is called both by the main controller and the MCL monitor. There are a number of different kinds of terminals available in this particular environment. They can be switched back and forth under the control of the main controller.

From the user's point of view, there are two command input modes. In one mode, the terminal controller is connected directly to the main controller; in the other, it is connected to the MCL monitor. We call them the **file mode** and the **editing mode**, respectively. Only the editing mode can be microprogrammed. The file mode commands are wired-down.

2.2 Some more Details

An internal file contains a sequence of lines (possibly null), i.e., a text. In the editing mode, one of the internal files is **active**. The lines contain 80 characters simply because our operating system imposes this part of the IBM culture upon us.

There are 36 **public internal files** corresponding to single letters A, B, ···, Z and digits 0, 1, ···, 9. In addition, there are 10 **hidden internal files** .0, .1, ···, .9 represented by a single digit with a dot prefixed. Each internal file has a private pointer which points at a boundary of two adjacent lines. The line just above (before) the pointer and the line just below (after) it are called the **previous line** and the **current line**, respectively*. There are 26 control program files .A, .B, ···, .Z.

The file mode commands accepted by the main controller are as shown in Table 1. Some of them appear in a sample session of Section 2.3.

*Here and in the sequel, the earlier members of the sequence of lines in a text are considered to be above the later members.

Table 1 The external command syntax.

Name	Format*	Meaning
Read	i=f1 f2···fn	Concatenates the contents of the files f1, ···, fn, and assigns the result to the internal file i.
Write	e=i1 i2···in	Writes a concatenation of the texts in the internal files i1, ···, in onto the external file e.
Compile	*i	Calls the MCL compiler with the contents of the internal file i serving as a source program.
Load	<e	Reloads the saved microprograms in the external file e onto the control program files.
Save	>e	Stores the currently loaded microprograms onto the external file e.
Enter	@i	Invokes the MCL monitor with internal file i serving as the active file.
Switch	−t	Switches to terminal t.
End	!	Finishes the KE execution.

*i, e, f, and t denote an internal file, an external file, an internal or external file, and a terminal device, respectively.

In our local environment, which is batch(!) rather than time-sharing, four different kinds of terminal devices are available. They are two CRT displays (marked in Fig. 1 as S1 and S0) which differ in screen size, color, and input facilities, also two typewriters (marked T1 and I0) which have different keyboard arrangements. Among the four, CRT display S1 is the most powerful. It has an 80*24 screen in three colors, a set of function keys, and a light pen. We will mainly talk about it in the sequel.

2.3 A Sample Session

We now describe a sample session assuming the display terminal (S1). Fig. 2(a)–(j) show what the screen looks like during the session. Some nonessential details have been omitted.

See Fig. 2(a). The user reads an external file into an internal file, say "A", by "A=FILE1, VOL1". Then he enters the editing mode by issuing the enter command "@A", Fig. 2(b). Note that the previous file

mode commands are stacked on the screen. The first stacked line "⟨FILE0, VOL0" indicates that microprograms were loaded automatically from the external file "FILE0, VOL0" at the outset of the operation.

Upon entry to the editing mode, a number of lines adjacent to the pointer position of the active file (A) is shown on the screen. In this case, the pointer points at the boundary between the top line and a hypothetical line preceding it, Fig. 2(c). Suppose that we wish to change "DRINK" to "EAT" in the second line of this Fortran program, and add a STOP statement. An N-command makes the next line become the current line.

We now have the FORMAT statement in the current line, Fig. 2(d). We issue two R-commands. First we change the leftmost (and in this case unique) occurrence of "9" into "7", and then replace the leftmost "DRINK" by "EAT", Fig. 2(e).

We now go down one more line, Fig. 2(f), and insert a STOP statement at the pointer position, Fig. 2(g). An

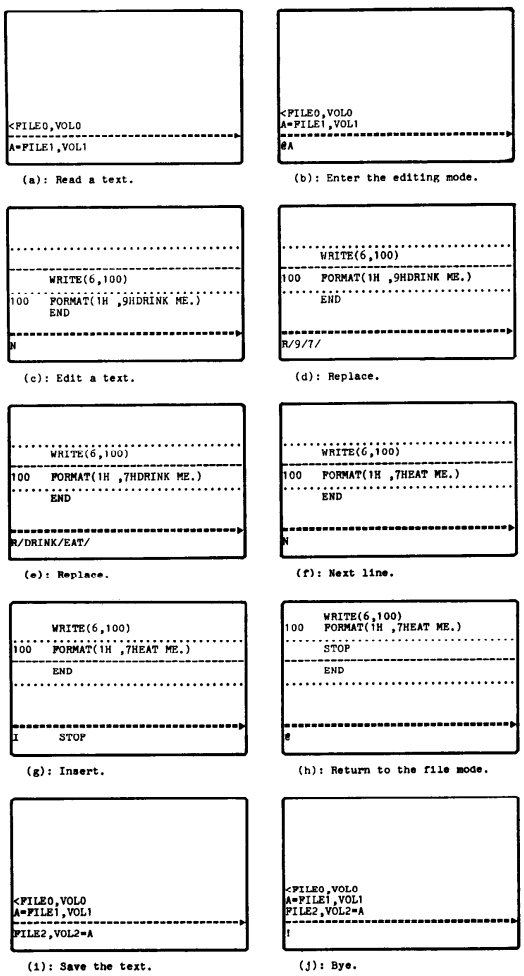

Fig. 2 A sample session.

@-command takes us back to the file mode, Fig. 2(h).

In Fig. 2(i), the user saves the result to an external file by typing "FILE2, VOL2=A". If he has another external file to edit, the above process may be repeated. When finished, the user types an "!", Fig. 2(j), whereupon the editing session terminates.

3. A Command System for Text Editing

To give the reader some idea about what can be implemented on KE, Table 2 gives a summary of a set of editing commands being used daily in our environment. The commands "N", "R", and "I" described in section 2.3 are not part of KE, but of the command system described here. The commands prefixed by "*" in the table, such as "C", "X" and "Y", are **ad hoc**. Some of them have been built for experimental purposes, while others, as tools for implementing other commands. The latter class of commands have been made available to users because they are useful in their own right. This particular system of commands is presented here as a sample. We have no intention to claim that this set is superior to any other. (Some of the command names, notably "until", are admittedly exotic. There are historical reasons for these [6].)

The command system can be used also with a typewriter, but we concentrate on display terminals here. Fig. 3 shows the standard display format on S1 in detail. (We note that this format is defined by a microprogram. See section 4.3.) At (1), names of the internal files currently in use are shown in green, with one exception: the name of the active internal file is shown in purple. The dashed line at (2) indicates the pointer position in the active file. The previous and the current

Table 2 A summary of standard commands.*

Code	Name	Description
*A	(Anchored)	Prefixed to U and H commands
B	Back	Same as /N
*C	Change	Similar to R, slow, with generalized pattern matching
D	Delete	
E	Execute	Command strings are executed
G	Get	Copying from another internal file
H	Hunt	Context search with deletion
I	Insert	Insertion
*J	JCL	Generates job control statements for the local operating system
L	Line count	
M	Movie	Scrolling over the text (displays only)
N	Next	Going down
P	Print	(Typewriters only)
R	Replace	
T	Top	Going to the bottom requires /T
U	Until	Context search
*X	eXhibit	Refresh the display
*Y	split line	Split the current line
*Z	utility calls	Ex. ZCONC for line concatenation
:	Set tab	
@	Exit	

The commands prefixed by "" are *ad hoc*.

lines are displayed at (3) and (4), respectively. Seven lines above the previous and below the current lines are displayed at (5) and (6), respectively. Typed commands enter the command display area, (7), and are sent to the computer upon pressing a **send-key**.

When a command moves the pointer or changes the text, the display is automatically refreshed. A refresh can be forced by an explicitly X-command (Table 2). Our terminal (S1) is sufficiently fast for the refresh operation to be invoked for each command, the transmission rate being 3000 char/sec.

In our environment, the display terminal has 16 function keys. They are used for tabbing, invocation of macros, and the like. These facilities are also coded in MCL.

4. The Micro Command Language

4.1 General Remarks

We will now show what an MCL program looks like. As noted earlier, MCL is a procedure-oriented string manipulation language with facilities for handling the internal files. It has modern control structures, a pattern-matching facility with a success/failure mechanism, and recursive procedures. On the other hand, it has no declarations except for modules and procedures. There are no global variables, and no goto statements. It partly takes care of command parsing.

There were three fundamental objectives in the design of the MCL language:

First, **simplicity**. MCL has been deliberately made simple. It has a small vocabulary and is represented in a compact syntax. No keywords are used. Instead, we use symbols such as "?", "<", and ">". Only a small number of built-in routines are introduced. Identifiers are restricted to very **basic** ones such as A, B, C, ···. On the interactive environment, compact representations are preferable, because they require fewer keystrokes and a larger part of MCL programs can be displayed on the restricted screen area. Simplicity also makes the compiler small and fast. MCL programs are compilable on-line without requiring more than very modest resources. In the quest for simplicity, readability has been sacrificed considerably but, hopefully, writability has not.

Second, **safety**. Dangerous facilities, which might be misused to cause a catastrophe in an on-line situation, have been carefully eliminated. For example, MCL has no facilities for switching the active internal file, or deleting an internal file. These must be done in the file mode. Pointer values can be changed only by using a restricted set of primitives.

Third, **machine-independence**. At least in its general structure, MCL does not depend on the particular machine or operating system.

An MCL program consists of one or more separately compilable modules. A module can contain one-level procedure declarations. No nesting of procedure declarations is allowed. Variables need not be declared. The type of a variable is that of the value which the variable contains, and is possibly changed by executing an assignment statement. MCL allows five data types: integer, string, file, boolean, and signal. The last two types are used for switching the program control flow. A value of type signals may be either "success" or "failure".

4.2 A Nonsense Operation

See Fig. 4 for a programming example. This program performs a nonsense operation: for each occurrence of words in the current line, the order of the characters is reversed. One can invoke this program by typing "K" as a command. For example, if the current line is

Time flies like an arrow.

the result is

emiT seilf ekil na .worra

The program works as follows. "&[" in line 1 and "]&" in line 12 enclose a module K. In line 2, "A=C(*)' ';" is an **assignment statement** which takes the content of the current line, appends a space to it, and assigns the result to variable A. "B=' ';" assigns an empty string to

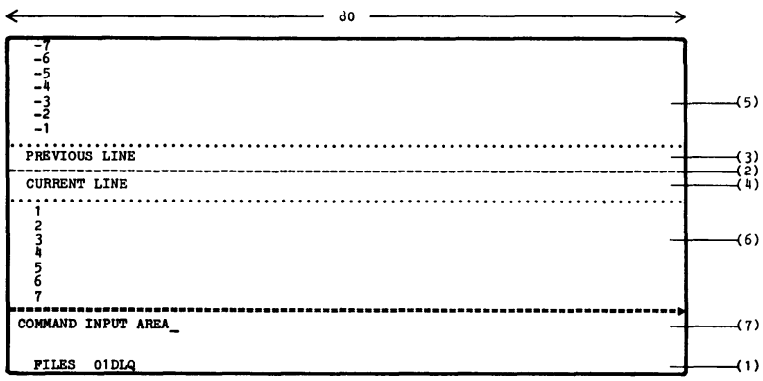

Fig. 3 The standard format of the display in S1.

variable B.

Symbols "<" in line 3 and ">" in line 10 call for an infinite repetition. All the statements enclosed by these symbols, possibly separated by semicolons, are repeatedly executed until either an exit statement or a return statement is encountered. Symbols "?" in line 4, ":" in line 4, and "," in line 9 correspond to "if", "then", and "else", respectively. The condition part, "$A=C,' ',A" in line 4, is a **matching expression.** If A contains a space, the substring preceding the first space is assigned to C, the substring following it, to A and the resultant value is "success". C may be null. If no space is found, no assignment is performed, and the value of the matching expression is "failure". The values "success" and "failure" of a type signal, when used in the "condition" part of an if statement, serve as "true" and "false", respectively. Hence, if A contains at least one space, the compound statement in lines 4 through 8 (enclosed by parentheses) is executed. "$C_1=I, C" is a **split expression**, which causes the value of C split into two pieces at position 1, and causes the divided substrings assigned to I and C, respectively. If this succeeds, i.e., if the value of C is non-null, the value of I followed by that of K is assigned to K. If the value of C is null, the split expression fails and returns "failure", whereupon "!" is executed. The symbol "!" represents an **exit statement**. It causes a one-level exit from the surrounding loop expression. In summary, the lines 4 through 8 reverse the characters in C, and append the result to B.

"*<=>B" in line 11 is a **replacement statement.** The contents of the current line is replaced by B. "*" specifies that the replacement occurs for the current line of the active internal file. In this case, the result contains one more characters than does a standard text line. (Recall that the lines have a fixed length.) The final character, which is a blank, is simply truncated.

4.3 More Realistic Examples

We now show a few further programming examples just to convey the flavor of what is actually going on in the microprograms implementing the command system of Chapter 3.

Fig. 5 implements the U-command (for a context search). "U" stands for "Until". The command moves the pointer **until** an occurrence of a specified pattern is reached. Thus, in "UDRINK" a line containing "DRINK" is searched for downwards starting from the line just below the current line. If found, the pointer is moved to the line boundary just above the first line containing "DRINK". Otherwise, an error message is displayed, and the pointer remains unchanged. In "/UDRINK" the action is a mirror image of "UDRINK" relative to the current pointer position, i.e., it searches upwards. "10UDRINK" causes "UDRINK" to be repeated ten times.

It is with the help of a mechanism built in the MCL language that this short program handles all the cases. In MCL, the module call construct determines the name of the invoked module at run time. The command "UDRINK", which corresponds to calling this module with an actual parameter 'DRINK', is taken care of by the module call "∧'UDRINK' ". Similarly "/UDRINK" corresponds to the module call "∧'/UDRING' ". Each module has an implicit direction variable. The direction to which the user wishes to search (represented by "/") is passed through the module call, and is assigned to the direction variable. An indicator "." can be used to modify the function of the file management constructs C(*) and N(*) depending on the value of the direction variable. For example, ",N(*)" is, if the value of the direction variable is empty, equivalent to "N(*)", i.e., moves the pointer downwards; otherwise, it is equivalent to "/N(*)", which moves the pointer upwards. Thus, the editing operations related to the direction are already in MCL. For further details see [6].

Fig. 6 implements the X-command (Table 2) for the multicolor terminal illustrated in Fig. 3. For example, see line 29. The screen area corresponding to (4) of Fig. 3 is refreshed by these statements. "P(G, 1120)" means that the value of G is displayed at the character position 1120. The screen format can be designed freely in this way.

This module contains two procedure declarations. One (in lines 2–14) is for refreshing the area (6) or (5) of Fig. 3, depending on the value of the direction variable. Another (in lines 15–25) is for writing the frames on the screen.

Finally, Fig. 7 implements the command analyzer of the command system described in Chapter 3. The U- and X-modules in the above are called from this module. A string typed at the user's terminal is read by the program by means of **get content** built-in routine call "G( )", and is analyzed as an editing command by the module call "E∧M". No explicit syntax analysis for the

```
 1  &[K:
 2      A=C(*)' '; B='';
 3      <
 4          ?$A=C,' ',A:(
 5              K='';
 6              < ?$C_1=I,C: K=IK, !>;
 7              B=BK'¯'
 8              )
 9          , !
10          >;
11      *<=>B
12      ]&
```

Fig. 4 A sample program for a nonsense operation in MCL.

```
 1  &[(N)U(S):
 2      ?N=0:N=1; S(*);
 3      <
 4          ?N<=0: !!S'4'; N=N-1;
 5          <
 6              ?/(.N(*)): ( R(*); !!F#'EOF'Z(N) );
 7              ?$.C(*)=,S,:!
 8              >
 9          >
10      ]&
```

Fig. 5 The U-module in MCL.

editing command is needed except for checking the appearance of the @-command for exiting from the command interpreter. All are done within the module call mechanism.

The typed command is executed in the following way. If the user types "UDRINK", the M-module of Fig. 7 causes the U-module to be invoked with an actual parameter 'DRINK'. If 'DRINK' is found in a search done in the program of Fig. 5, a pair consisting of '4' and the signal "success" is returned; otherwise a pair consisting of 'EOF' and the signal "failure" is. ('4' is simply a flag.) If a "success" returns, '4' is assigned to E and "%'D'E" is executed, causing the internal procedure D to be invoked with the argument '4', whereupon the display screen is refreshed. If a "failure" returns, 'EOF' is assigned to E, and "P(E)" is executed, whereupon an error message 'EOF' is displayed.

4.4 Interface Between KE and MCL

The enter command (@) of Table 1 conceptually invokes an MCL program, in an obvious transliteration, as follows:

```
enter:
loop
  call module M returns(A);
```

```
1  &[X(A):
2     &[D:
3        S(*);
4        L=80; ?#='/':L=-L; P=1040+L+L+L;
5        S='                                                  '
6          '                                                  ';
7        T=7; .N(*);
8        <
9           ?T<=0:!; T=T-1;
10          ?(.N(*)):G=./C(*), G=S;
11          P(G,P);  P=P+L
12          >;
13       R(*)
14       ]&;
15    &[X:
16       U=960; D=1120; L=80; Z=-80; M=1040;
17       P('                                        '
18         '........................................', U-L);
19       P('                                        '
20         '........................................', D+L);
21       P('                                        '
22         '========================================', Z);
23       P('                                        '
24         '----------------------------------------', M)
25       ]&;
26    S='                                            '
27      '                                            ';
28    A=I(A);
29    ? A=0 : ( G=C(*); ?G='':G=S; P(G,1120) )
30    ,? A=1 : ( G=/C(*);?G='':G=S; P(G,960) )
31    ,? A=2 : ( ^'X0'; $'D' )
32    ,? A=3 : ( ^'X1'; $'/D')
33             ( $'X' )
34    ]&
```

Fig. 6 The X-module in MCL.

```
1  &[M:
2     &[E:
3        M=G();
4        ?M='':(%'D4';!!S);
5        ?$M=Y,'@',X:(?Y='':?X='':!!S'!',!!FX);
6        ?E^M:%'D'E, P(E)
7        ]&;
8     &[D(E):
9        ?E='':!!S;
10       ?$E=,'/',X:($'1032'=Y,X,;X=L(Y)) , X=I(E);
11       ?X>3:(^'X3';^'X2'), ^'X'Z(X)
12       ]&;
13    %'D4';
14    < ?X%'E':(?X<>'':!), !!FX  >
15    ]&
```

Fig. 7 The main module M in MCL.

```
if success
  then exit
  else change the active file name to A
end loop;
```

Thus, the M-module is activated by a module call. If it returns with "success", the enter command is completed. Otherwise, the active file is changed to that one specified by the user, and the M-module is invoked again. No construct is available for switching the active file in MCL.

5. Implementation

Implementation of KE is rather obvious. One particular implementation of KE on a FACOM 230–45S computer [7] is as follows:

The **internal files** use a doubly linked list structure. The elements are blocked with a certain blocking factor, and are stored in a direct-access disk file, which in fact accommodates all the internal and control program files. Each internal file, in addition to the doubly linked list, has four pointers (record numbers) pointing at the top line, the bottom line, the current line, and the saved line, respectively. The saved line pointer is used for marking a point in an internal file.

The **control program files** use the same linked list structure as above. They have only two control variables each: the location of the first element of the linked list, and the size of the microprogram.

The **main controller** is wired-down and written in an assembly language. The **MCL compiler** includes a syntax-directed top-down analyzer. Each module declaration, after being compiled, enters a separate control program file. The **MCL monitor** consists of 33 primitive routines, one process controller, two separate stacks (for integers and strings), and one cache area in which microprograms are loaded. To minimize the runtime memory consumption, the object code normally resides on a disk file. The **terminal drivers** provide uniform interfaces to the four terminals. There are a number of primitive routines corresponding to the MCL I/O functions.

We had one nasty problem: our operating system used the EBCDIC code, while the terminals used various variants of the ASCII 7-bit code. As a compromise, an extension of ASCII was introduced. Code conversion from EBCDIC to this code, and *vice versa*, are performed within the handlers of file-mode read and write commands. No other parts of the program see EBCDIC.

To minimize memory consumption, the file access module, the MCL compiler, and the MCL monitor are overlaid. So are the four terminal drivers. KE consumes only 24 KB of the main storage with a satisfactory response time.

As our environment happened to be batch, KE had to be very small in order not to obstruct other jobs. This restriction of storage affected the design of MCL and its implementation. Even in a time-sharing system, however, a small working set is always welcome. Moreover, by pursuing low storage consumption, we were

forced to make the system compact and well-structured.

Given a set of primitives for handling internal files as well as terminal drivers, it is an easy task to implement KE. Our experience was that only 20 man-days were required for assembling KE from parts of another editor (a forerunner of KE having a single-layered organization).

6. Experiences

6.1 A History

KE evolved out of a conventional display-based text editor, for which we can only cite [8] as a very remote ancestor. It was developed locally around 1974 by the first author, and had a single-layered organization. It had ten internal files 0, 1, ···, 9. The current line pointer pointed at a line rather than a line boundary.

This single-layered editor was used as a daily tool for about two years within the authors' environment as well as in some other sites, and underwent several enhancements and modifications. However, the process of the change was slow, chiefly because any modification required reprogramming in an assembly language.

During the period the need for a macro facility became apparent. Various designs extending the existing command syntax were considered, but none of them satisfied us completely. Enough extensibility always seemed to contradict integrity, especially with respect to command delimiters. After several agonizing trials we reached the conclusion that a double-layered organization should be used. A version of KE was first implemented in October 1976, and its first test was the writing of a microprogram simulating the external behavior of the single-layered editor.

From that time on, the evolution of the command repertoire proceeded fast. We were now able to test new ideas and users' suggestions quickly. We no longer had to worry about the impact of a change beforehand. We could simply make a change in the MCL source program, compile it, and test it. We could revert to the original system any time if the change did not please us.

Change of colors, extensive use of function keys, and the like were later additions. Some of them did require reprogramming in the kernel, but the effort was always minimal because most of the logic could be supported by MCL programs.

In retrospect, it was a big plus for us that we had intimate interaction with the users. We could ask a user whether he liked a change prompted by his suggestion before it was frozen. The turn-around time was very short, often a matter of minutes. We feel that such an environment is essential in developing software such as a text editor.

Of course, casual users need not know MCL. More skilled ones may wish to write MCL programs themselves. Several people actually developed their own editors on KE. Among them, perhaps the most prominent is Sado's light-pen oriented editor for S1[9]. It allows the user to do most editing operations by simply pointing at appropriate points on the screen. Sado succeeded in building a smooth man-machine interface by writing a big MCL program.

6.2 Examples of the Changes

The following are some of the changes we made in our microprogram during the above-described process of evolution.

Keeping a command on the screen. In early versions of our microprogram, we were erasing a command string in the command input area after it was executed, but it was a poor idea: we often wished to review the command, or even reuse it. We decided to leave the executed command on the screen, simply returning the cursor to the home position. It now became possible to move the cursor to the end of the command by pressing an off-line key, and then press the send-key to reissue the command.

Anchored searches. It is desirable to be able to search for a string that begins at the first column of a line. We first introduced a V-command for doing this by an analogy to the U-command. Only after installing it we realized that this naming was awkward; what name could we use for an anchored H-command? For a more systematic naming, we prefixed a non-alphabetic character '[' to the commands. But this alternative was also unsatisfactory because the character, which is at the right end on the keyboard, was not easy to type on our terminals. Finally, we decided on the idea of prefixing the character A (for "anchored"); AU did an anchored search, and AH, an anchored search with deletion. This naming turned out to be comfortable.

A utility library. In addition to conventional text editing commands, several utility operations were introduced. Some of them are: J (for generation of JCL programs for the local operating system), ZCOMPARE (for comparison of two internal files), and ZCONC (for concatenation of two lines).

The reader may have been shocked by the liberal ad hoc commands in the anchored search and the utility call commands. As for anchored searches, it may indeed be better to extend the syntax of search patterns to include a head-of-line mark, a tail-of-line mark, and the like. We have partly done this on a trial basis in the C-command. In general, however, ad hoc commands don't hurt too much in the KE environment. The reason seems to be that, by intimate interactions with the users, even ad hoc commands can capture some of the underlying psychological realities.

6.3 Non-text-editing Applications

KE is a powerful tool beyond text editing. For example, elementary pretty printing can be done on KE.

KE is useful for rapid prototyping in interactive systems. The experimental system built on KE some-

times turns out to be practically usable. For example, a multi-font typer program was built in this way.

Fig. 8 illustrates a system, built overnight on KE, for designing graphic patterns. Points are specified by the light pen, and, upon hitting a function key, line segments are drawn by the computer. We could design representations of some Chinese characters. Fig. 9 shows the result of our first experiment. The two Chinese characters represent "Tokyo".

7. Discussions

7.1 Advantages and Disadvantages

By now it should be clear that our double-layered organization comfortably provides extensibility and flexibility necessary for quick experimentation and modification. As discussed in the preceding chapter, this aspect of KE was of a real help in improving the command system of Chapter 3.

An additional, less obvious advantage of the double-layered organization can be found in the insulation between the concrete realization (implementation) of a command, and its use in editing. An MCL program can be constructed in the editing mode, but before it can be invoked, the file mode must be entered, because a compilation of MCL programs can be done only in the file mode. The users cannot accidentally destroy a command during ordinary use of the editor.

Other minor advantages are procedural style and machine independence. A command written in the procedural style is much more manageable than one in a command style for those users familiar with conventional programming languages. In the general structure, KE does not depend on the particular machine or operating system. KE is easily implementable in other environments.

A possible disadvantage is that our organization could consume more resources than conventional single-layered organizations. In particular, the time for interpreting the microprograms could be a problem. Our experience is that a KE-based editor is somewhat slower than directly coded ones, but the difference is so small that the flexibility by far overweighs the additional resource consumption.

7.2 Comparison with TECO

There is an alternative to our approach: we could incorporate an extensive macro facility. This has been done in several existing editors, notably TECO[4]. In TECO, there are two kinds of commands, i.e., basic commands and macro commands. Basic commands are wired down, and cannot be changed. Macro commands are constructed from basic commands (and from other predefined macro commands). By issuing a macro execution command with a specified storage name representing a special store (the Q-register) containing a sequence of commands, TECO can execute the sequence of commands, or a macro. (It is also possible to specify an external file in which a sequence of commands resides.) Certainly it is multi-layered. However, one big difference is that, in TECO, the lower level can be seen from the user, i.e., the two levels cannot be insulated.

As for extensibility and flexibility, TECO is comparable with KE. However, there are following points.

a. Two classes can be distinguished among macro commands: those created by the casual user for one-time use, and those that are carefully designed and implemented for use as standard tools. In TECO, both must be created by the same mechanism, while in KE, they can be distinguished. The one-time commands can be serviced by a

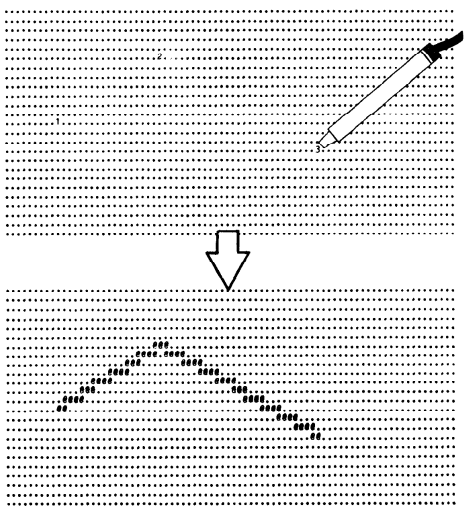

Fig. 8 Drawing figures with a light pen.

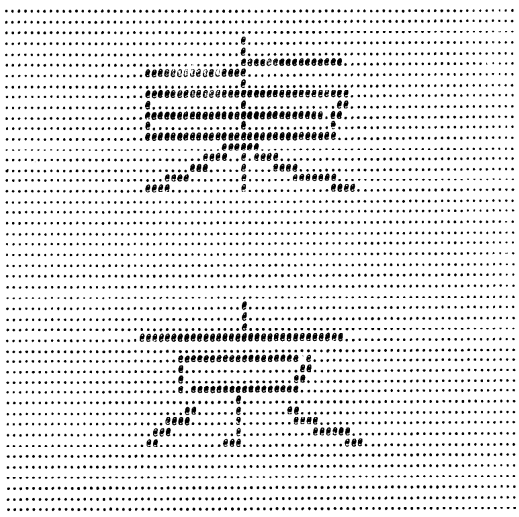

Fig. 9 "Tokyo" written in Chinese characters.

microprogrammed easy-to-use macro facility (e.g., E-command of Table 2).

b. In TECO, basic commands cannot be undefined or redefined.

c. In defining a macro command, we sometimes wish to define substructures on which to base the implementation. In KE, this can be done by defining internal procedures, while in TECO, there is no convenient equivalent.

d. In TECO, macro commands have a predetermined format. The prompting symbol, the escape symbol, and the like are fixed. The format of error messages is also fixed. In KE, no such restrictions exist.

As for insulation and security, KE is significantly better than TECO. In fact,

a. The commands which are not intended for general use should be hidden from the eyes of the user. In TECO, we often need such a hidden command because internal procedures are not available. All that we can do, however, is to define and call subsidiary commands. There is no easy way to insulate them from the others. If the user makes a typographical error, he might inadvertently invoke one of these commands, and be surprised.

b. In TECO, macro definitions are usually contained in the Q-registers. There is no protection mechanism for them. If the user changes the content of the Q-register, a disaster might result.

Finally, TECO suffers from its command style. Thus in TECO, the basic commands have been designed for on-line editing, and are not necessarily suitable for writing complex macro definitions. It requires skillful (tricky) techniques which would normally be used only for coding in a very low-level programming language. An MCL program is much more lucid.

7.3 A Comparison with EMACS

We now compare KE with EMACS, an independent effort [5]. EMACS was started from TECO, it extended and improved the command language, and grew into something else. The original EMACS was written in a TECO-like language, while post-EMACS systems were written in Lisp [5, p. 20]. EMACS has many things in common with KE: it is highly extendible; it is in a way double-layered; its command definitions, given in a "library", are protected from on-the-fly modifications during an editing session, and so on. It is designed for a full-duplex terminal, has a lot of luxurious features, which makes it a huge system.

We admit that EMACS is superior in the following points:

(1) explicit provision of a library facility;

(2) self-documenting; and

(3) tighter coupling with the user in editing sessions.

Thus, in KE, the users must establish and follow suitable conventions within his community if command definitions are to be shared as a library. In this respect, EMACS is certainly better.

EMACS's on-line documentation is enviable. Here we could only point out that the KE system is so small and simple that we have much less need for extensive on-line documentation.

As for (3), we had no choice. Our operating system had only half-duplex terminals. Our approach would have been different had the operating system supported full-duplex communications.

On the other hand, we feel that KE is better in the following points:

(1) separation between the file and the editing modes;

(2) simplicity of the system and of the language MCL;

(3) better user-service.

Thus, the user is better protected from errors because anything he does in the editing mode cannot affect an external file or a compiled microprogram. This contributes to the simplicity of editing mode commands. The user need not even *think* about operating system functions while he is within the editing mode.

We acknowledge that a command calling the operating system would be a nice addition to the internal command repertoire. It is important, however, that the editing and the operating system commands do not intertwine. As Card, Moran and Newell [10] have observed, one of the important dimensions to the performance of a user-computer system is "concentration", i.e., the smallness of the number of things which the user must keep in mind while using the system.

As for (2), MCL is designed explicitly as a language for writing editor commands. It does not contain anything beyond basic text processing. This has made KE compact, easy to implement, maintain, and transport. It could be implemented on a microcomputer if one so wishes. We could even microprogram our "microprograms".

To the best of our judgment, KE seems to provide better user-service. It has a simple organization. Users can easily understand KE and MCL. In fact, several people have succeeded in developing their own command systems on KE without the help from the authors, see Section 6.1.

7.4 Areas for Further Improvement

We admit that MCL requires improvements in the following directions.

(1) The identifiers are too restrictive. One-letter identifiers are not sufficient. (This particularly troubled Sado [9] since his MCL program was much bigger than ours.)

(2) The string manipulation primitives are weak. More powerful pattern matching is needed. In fact, an attempt to introduce more powerful matching in the U- and H-commands within the present framework resulted in intolerably slow implementations.

(3) We need better cosmetics and more syntactic

sugaring.

(4) We need some more primitives such as date/time interrogation.

Another area is KE's interface to the operating system. It is not convenient that the file mode is wired-down. We have a plan to redesign KE to enable micro-coding of the file mode. (This will make a library facility available as a fringe benefit.) Moreover, It would be desirable to introduce more flexible directory structures such as trees.

Acknowledgment

The authors are grateful to Izumi Kimura for his patient guidance and help in crystallizing the ideas of this paper. Many people, too numerous to mention, as users provided precious feedback to the design of KE and MCL. We thank them. Special thanks are extended to Kazuo Ushijima and Naomi Fujimura for giving us a chance to transport KE to their machine. Thanks are also due to Akinori Yonezawa for helpful comments, and to Craig Everhart and R. M. Stallman for discussions.

References

1. DIJKSTRA, E. W. The Humble Programmer, *CACM* **15**, 10 (Oct. 1972), 859–866.

2. GRISWOLD, R. E., POAGE, J. F. and POLONSKY, I. P. The SNOBOL4 Programming Language, 2nd ed. Prentice-Hall, Inc., Englewood Cliffs, N.J. (1971).

3. GRISWOLD, R. E. and HANSON, D. R. An Overview of SL5, *SIGPLAN NOTICES* **12**, 4 (April 1977), 40–50.

4. Digital Equipment Corporation: DEC System 10 TECO Text Editor and Corrector Program Programmer's Reference Manual, DEC-10-ETEE-D, Digital Equipment Corporation, Maynard, Mass., (1972).

5. STALLMAN, R. M. EMACS The Extensible, Customizable, Self-Documenting Display Editor, *AI Memo* No. 519, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (June 22, 1979).

6. KAKUDA, H. and TSUJI, T. A Double-Layered Text Editor, *Research Reports on Information Sciences*, No. C-28, Department of Information Science, Tokyo Institute of Technology (March 1980).

7. Fujitsu Limited: A Guide to FACOM 230–45S, FACOM 230–45S 45EX0002E-1, Fujitsu Limited, Tokyo, Japan.

8. DEUTSCH, L. P. and LAMPSON, B. W. An Online Editor, *CACM* **10**, 12 (Dec. 1967), 793–799.

9. SADO, K. Building a Display-Oriented Text Editor on a "Microprogrammable" Kernel, in preparation.

10. CARD, S. K., MORAN, T. P. and NEWELL, A. The Keystroke-Level Model for User Performance Time with Interactive Systems, *CACM* **23**, 7 (July 1980), 396–410.

(Received August 11, 1980: revised October 19, 1981)