DWC

# TEXT EDITOR

USER'S MANUAL

PRELIMINARY

TECHNICAL DESIGN LABS

Z80

# THE ZAPPLE TEXT EDITOR

User's Manual

December 1976

# THE ZAPPLE TEXT EDITOR

## A. INTRODUCTION

Welcome to the Zapple Text Editor. This program is designed to be used for writing assembly language source code as well as any general text editing functions. This manual has been prepared using the Zapple Text Editor in conjunction with our word processing system which controls formating.

The Text Editor occupies roughly 3K of core, is relocatable (may be loaded at any address), and is romable.

The text editor is sent in TDL's relocating format, and should be loaded with either the ZAP or ZAPPLE using the "R" command.

The suggested load address is 100H (R,100) for RAM operation, and 0E000H for ROM (or protected RAM) operation (R,E000). No matter where it is loaded, however, it uses locations 0080H through 00FFH (128 bytes) for temporary storage.

The editor expects the monitor (ZAP or ZAPPLE) to be located at 0F000H. The JMP vectors at the start of the editor will have to be modified if your monitor resides at a non-standard address. Refer to the monitor manuals for further details.

## B. THE TEXT EDITING PROCESS

The creation of Text is performed in a discrete sequence: load & start the editor, type in the text, edit the text, incorporating any additions and corrections, and then output the text file. The Editor commands are designed to give you complete control over the functions which are involved in these four factors.

Generally speaking, the commands involved in the creation of text are of the greatest significance to the user - because they will be used most often.

An understanding of how text is manipulated by the editor, and how the user controls this is the most basic point to be covered.

Three primary factors are involved in the creation of text: the text buffer, the buffer pointer and the manipulation of text itself.

The text buffer is maintained by the editor as a storage area into which text is placed. The buffer is variable in size, expanding and contracting to accomodate the text as it is created. Conceptually, understand that the text buffer is never larger than the amount of text you have generated. The LIMITS on text buffer size however is controlled by two variables: sizes specified as part of the sign-on procedure, and the absolute size of the memory available. When the buffer is empty, the beginning and end are the same, and the size is ZERO.

The buffer pointer is simply a movable position indicator. By utilizing the pointer control commands, it may be moved to any position within the text buffer. Its limits as to where it may be located are few. It may NOT reside outside the buffer, and it may NOT be located "on top of" any character. The pointer MAY reside either before or after any character in the buffer. It is helpful to visulize the pointer as always being between one character and another. The only exceptions to this are when the pointer is at the beginning of the buffer, it is IN FRONT OF the first character. When at the end of the buffer, it is BEHIND the last character.

When text is being created, it is placed in the buffer IMMEDIATELY after the pointer. As each additional character is entered, the pointer moves ahead by one character position, thus causing no violation of the principle in the first sentence of this paragraph.

The family of pointer control commands are of great importance to the user because all additions and corrections are made only in reference to the pointer (before it or after it). Movement of the pointer may be either in terms of characters or lines. In other words, move the pointer so many characters forward or backward, or so many lines up or down. These commands affect the position of the pointer, but do not modify the text.

The family of commands which control the insertion or modification of text comprise the second significant group. It's best to consider these commands as those which in any way modify the content of the buffer. These ALSO move the pointer, but they ALWAYS change the text.

These commands allow direct insertion, moving the text out of the way (pushing it down), forward and backward deletion, full line deletion, and substitution.

When these commands are understood, and used together, you will find that you have complete control over the text and can very rapidly create, add to or modify that which you have created.

## C. COMMAND FORMATING

Every command used by the Zapple text editor consists of a single letter of the alphabet. These are often used in conjunction with numeric arguments describing the number of times a function is to be performed, and other command inter-relationship functions.

The execution of a given command is governed by the ESCAPE (or ALTERNATE MODE) character. When a double escape (two escapes in a row, echoed as two dollar signs '$$' ) is received by the editor, the current operation being requested is performed, and the results are entered into the buffer. Up to that point, regardless of the amount of work you have done, it is not "official" as far as the editor is concerned.

```
EXAMPLE:     T$$
```

types the line from the location of the pointer. Simply typing "T", without the two escapes following it, has no effect.

If you wish to terminate a sequence which is already in progress, such as "Type 100 lines" (100T), entering a CONTROL-C from the keyboard, prior to the double escapes, will terminate the function.

```
EXAMPLE:     (Control-C; no echo)
             *BREAK*
```

would terminate the operation, indicated by the "*BREAK*", response from the editor.

A CONTROL-C can also be used to terminate any other command in progress. For example, if you commanded the editor to "delete 20 lines" and decided, BEFORE typing the two escape characters, to NOT delete those 20 lines, typing a CONTROL-C would end the sequence, without affecting the contents of the buffer.

A numerical argument placed before a command signifies the number of times that function is to be executed.

```
EXAMPLE:     20T$$
```

will print the next 20 lines in the buffer, without moving the pointer.

```
EXAMPLE:     -20T$$
```

will print the 20 lines preceding the pointer down to the pointer.

A very significant feature of the editor is its ability to handle command strings. That is, commands may be strung together, and a "macro" execution performed, providing only that string functions be separated by single escape characters.

EXAMPLE:     -20T20T$$

will print the 20 lines preceding the pointer and the 20 lines following the pointer. These strings may be extended at will, providing only that character string related commands (such as SEARCH [S], FIND [F] and INSERT [I]) must be separated from the other commands by a single escape character.

EXAMPLE:     Frod$2T$$

Tells the editor to FIND the string "rod", (positioning the buffer pointer after the "d"), and type the next two lines. Failure to close the string with an escape would cause the next command to become part of that string, rather than to act as a separate command.

Command strings may be treated as separate units by enclosing them within greater than and lesser than signs.

EXAMPLE:     20<LT>$$

commands the editor to move the pointer down one line, then print the line, then move the pointer down, etc. Entering "20L20T" on the other hand would first move the pointer down 20 lines, then print the next 20 lines; which is clearly quite a different matter.

A common usage of this involves the command string "TPL", which first prints the line, then pauses until a space bar character is received, then moves down one line, prints that line, pauses, etc.

EXAMPLE:     20<TPL>$$

This command string, enclosed in brackets and preceded by a numerical argument allows the pointer to be moved down, and each line successively printed, under control of the space bar. When a desired location was found, a CONTROL-C would end the sequence, leaving the pointer at the beginning of the last line typed.

These basic command formats cover all the basics. As you use the editor more, and your familiarity increases, the permutations of these basic applications will provide you with the full power of this editor.

## E. THE COMMAND SET

The following is an alphabetical list of the editor commands, accompanied by a brief description of what each does. It may be used as a function guide.

A - Append 50 lines of text from the monitor system reader to the end of the buffer.
B - Move the pointer to the beginning of the buffer.
C - Move the pointer + or - "N" characters.
D - Delete + or - "N" characters.
E - End of editing. Dumps the entire buffer contents to the system punch, and reinitializes the editor.
F - Find a character string and position the pointer after the last character. Limited to 16 characters in the search string.
G - Same as the "T" command, except the output goes to the assigned line printer device rather than the console.
I - INSERT input keyboard data into the buffer immediatly after the pointer. (Begin editing)
J - Prints out the absolute location of the pointer in Hex.
K - DELETE "N" number of lines in front of or behind the pointer. (+ or - K)
L - MOVE the pointer + or - N lines, positioning the pointer at the beginning of the line.
M - Prints the remaining unused workspace still available to hold text.
N - Punch 60 nulls to the punch device.
O - Insert "N" number of rubouts after a carriage return.
P - PAUSE in the function until a "space bar" character is received from the console.
Q - Ring the system bell. Useful to know when a lengthly job cycle is complete, and attracting the operators attention.
R - Indicates the current pointer position in "number of lines from the beginning". I.E. "Z$R$$" will print the total number of lines in the file.
S - SEARCH AND SUBSTIUTE. Search for a string and substitute with another string. Effectively combines the "F" and "I" commands.
T - TYPE "N" number of lines + or - the pointer position.
U - Dumps the entire buffer contents to the punch, without destroying the buffer contents. Includes even parity on the saved text file.
V - Tells how many characters of text the buffer contains.
W - Writes "N" number of lines from the buffer to the punch, and deletes them from the buffer.
X - EXIT back to the monitor. Editing may resume by either typing "G(cr)" from the monitor, or restarting the editor at the CONTINUE vector. (start +3).
Y - Identifies the end of the buffer (in hex).
Z - Moves the pointer to the bottom (end) of the buffer.

## F. SPECIAL CONTROLS

TAB (Control-I) - Moves the print head to the next tab column. Tab stops are fixed at every eight positions.

RUBOUT - Deletes the most previously typed character, and echoes them as they are removed.

CONTROL-C - Aborts all printing operations, and can stop the execution of a "W", "U" or "E" command. The buffer is preserved in this instance.

"<" & ">" - Multiple command delimiters. The commands enclosed in these characters (I.E. 2<25<20W>>$$ which means "2 times 25 times write 20 lines to the punch device." or "write 1000 lines to the punch device") will be performed as many times as the numeral preceeding them requests, or until a condition is not met, such as the inability to "F1235$$" (find the string "1235") 10 times. For example: "10<F1235$-1C$I4$0TT$>$$" means "Find the string "1235", back up one character, insert a 4, print the corrected line, and then look for the next occurence of that string, and repeat until you have done it 10 times. In this example, if there had only been 7 occurences of the string "1235", it would have finished repairing the 7th one, and then upon searching for the 8th occurence, it would fail to find it (reaching the end of the text buffer) and print the message:

```
CAN'T FIND "1235"
*
```

Note: The number of left brackets must be matched with the same number of right brackets. Maximum number of enclosures is 8.

There are other diagnostic messages which may occur from time to time, which are self-evident upon examination of the situation.

## G. TEXT EDITING SAMPLES

In order to better demonstrate the effect of various editing commands, let us suppose that the editor has just been loaded, and execution has begun at the starting address. The editor will type:

```
BUFFER START-
```

This is a request by the editor as to where you wish the text buffer to reside. Answering with a carriage return will cause a default starting address, (the actual location depending on where the text editor has been loaded) and answering with an actual location (in HEX) will cause THAT location to be used as a start of buffer. If there is any conflict between an actual address and the location of the editor, it will re-state the question. In this example, we have answered with a carriage return.

The editor then asks:

```
BUFFER END-
```

This request may also be answered with a carriage return, allowing all of the available remaining memory to be used by the editor. It may be kept within a fixed area by answering with a specific limit address (in HEX).

The editor then calculates the workspace remaining, and prints this on the console:

```
-WORKSPACE = XX,XXX  (total buffer size, in decimal)

*
```

You are now in the editor's command mode, as shown by the asterisk. If we now wished to begin a file, we would type "I", and then begin typing the text we wish to work on. A carriage return is all that is required to cause BOTH a carriage return AND a line feed to be entered into the buffer. However, if/when deleting characters from the buffer, remember that there are TWO characters between lines, even though you only typed the single CR. This occurs only during keyboard input, and NOT when loading a text file from the assigned reader device. To continue, let us now type in some text:

```
*I(cr)
The quick fox jumps over the lazy dog's back. 1123456789.
(cr)$$ (two escapes, echoed as dollar signs)
*
```

At this point, the buffer contains the following:

```
CR,LF,The quick ....etc.... CR,LF
```

And if a "B" and a "2T" command were given, it would re-type the line on the console:

```
*B$2T$$

The quick fox jumps over the lazy dog's back. 1123456789
*
```

Now, suppose we wished to change the mistake in the line (1123). There are many ways this may be accomplished, but the easiest way would be:

```
*S11$1$$
*
```

Notice that the editor simply prints the asterisk at the completion of the task. To verify that it has been corrected, we can either again type the command "B$2T$$" or we could type "0TT". The "0T" means to type from the START OF LINE to the current location of the pointer, and a "T" means to type from the current pointer position to the END OF LINE. Combining them as "0TT" will print the whole line, without moving the pointer. For example:

```
*0TT$$
The quick fox jumps over the lazy dog's back. 123456789
*0T$$
The quick fox jumps over the lazy dog's back. 1*T$$
23456789
*
```

Note that the asterisk was printed following the numeral one, showing the pointer position to be between the "1" and the "2", or just after the character entered as a result of the previous "S" command. The execution of two escapes in a row will always force a carriage return / line feed, thus the balance of the text following the pointer is printed on the next line.

If you now wished to enter additional text into the buffer, typing a "Z" would assure the pointer going to the end of the buffer, and then typing an "I" would again begin the insertion of text from the keyboard.

If during the inputting of text, you wish to abort the entire operation, typing a CONTROL-C will cause a *BREAK* message, and NOTHING will have been entered into the main buffer.

Please experiment with the various commands, but of course practice on a file that has no importance, and make sure you have a good feel for the various commands, and their effects, before starting on any important editing jobs.

To continue, if you now wished to SAVE the text that you have been working on, typing an "E" command will cause the entire buffer to be sent to the monitor assigned punch device. If the editor detects that both the console and the punch are the teleprinter (model 33 for example) it will first wait for the operator to start the punch, and then by typing a space on the console, the punching will begin. The file will be punched, and an end-of-file mark will be appended to the end of the tape, followed by some blank trailer. Typing a space again (after the punch has stopped) will cause the editor to be re-started. Remember you can return to the monitor with the "X" command.

Now suppose you wanted to re-load the file just punched for further editing. Assuming the editor is loaded and initialized, and the monitor's assigned reader has the file just punched loaded, you would then type the command "A$$", and the first 50 lines of the file would be loaded into the buffer. An asterisk will be printed at the conclusion of the load. If there were more than 50 lines of text in the file, additional A's could be executed, or initially the command "AAA$$" could have been used. When loading large files, or from cassette, it is advised to use the multiple command technique, and ask for more lines than are actually in the file. Each time the CALL to the reader device returns with the carry set (an OUT OF DATA condition), it cancels one "A". Thus, the command "20<A>$$" would cause as many as 1000 lines to be read in.

The buffer pointer does not move during the Appending process. In addition, if you are not able to hold all of the text source in the editor workspace at one time, due to memory limitations, etc., you can bring in some text, edit it, write the edited part out ("W" command) and then append ("A") more into the end of the buffer, edit that, write some more out, etc. The "A" always appends text starting at the END of the buffer. The "W" always writes from the start of the buffer, and the remaining text is PACKED down to make room for additional appends.

Again, study of the list of commands will help you to familiarize yourself with all of the power available in this Z-80 text editor. Any comments or questions, sent to T.D.L. in writing, will be answered and greatly appreciated.

TECHNICAL DESIGN LABS, INC.
RESEARCH PARK, Bldg. H
PRINCETON, NEW JERSEY
08540

609-921-0321