USERS' MANUAL

A product of Soft Corp for

Thinker
Toys™

1201 10th St Berkeley, CA 94710

A T E

AN ASSEMBLER AND TEXT EDITOR
(AND CASSETTE OPERATING SYSTEM)

A stand-alone 8080 program development system designed to run with Morrow's Micro-Stuff Speakeasy I/O board

Features

- One of the most versatile text editors ever written, ATE can edit any kind of text from assembly language to "English" language.
- ATE is completely programmable. You can create your own high-level editing commands (or "edit macros"). Repetitious editing operations or time consuming tape references can be run automatically.
- The assembler can handle programs larger than memory, can produce object code listings in any base you want (hex or octal or decimal or whatever), and allows you to edit the object code easily even without an object code listing.
- ATE fits into 4K of memory and runs in 8K. More memory can easily be utilized since everything (the symbol table, the source file area, etc.,) is moveable.

Contents

Page

Contents

Introduction

For the reader who enjoys learning a new language by total immersion, here is an introduction to the text addressing capabilities of ATE. Skip this part if you want -- detailed explanations follow.

```
"..
I CANNOT FIND MY WAY. THERE IS NO STAR
IN ALL THE SHROUDED HEAVENS ANYWHERE.
"(THERE)..
THERE IS NO STAR
IN ALL THE SHROUDED HEAVENS ANYWHERE.
"..(DED)
I CANNOT FIND MY WAY. THERE IS NO STAR
IN ALL THE SHROUDED

"(TH)..(H)
THERE IS NO STAR
IN ALL TH
"(TH)..(H)..(O)
THERE IS NO STAR
IN ALL THE SHRO
"(TH)..(O)
THERE IS NO

"(IN)..(ED)
IND MY WAY. THERE IS NO STAR
IN ALL THE SHROUDED
"(IN )..(ED)
IN ALL THE SHROUDED
```

```
"(IN )..(ED)
IN ALL THE SHROUDED
"/(TH)..(O)
THE SHRO
"/..(HE)
THE
"(IN )..(ED)/(TH)..(O)/..(HE)
THE
"..(HE)
I CANNOT FIND MY WAY. THE

"..2(HE)
I CANNOT FIND MY WAY. THERE IS NO STAR
IN ALL THE
"..3( )
I CANNOT FIND
"2( )..3( )
 FIND MY WAY.
"-1( )..
 ANYWHERE.
"-2( )..
 HEAVENS ANYWHERE.

"..←
I CANNOT FIND MY WAY. THERE IS NO STAR
"←..←
IN ALL THE SHROUDED HEAVENS ANYWHERE.
"(HEAVENS)*
IN ALL THE SHROUDED HEAVENS ANYWHERE.
"2←*
IN ALL THE SHROUDED HEAVENS ANYWHERE.
```

```
"(A)@@(N)
AVEN
"( )@@@( )
 ALL
"2( )@@@( )
 THE
"2( )...( )
 FIND
"15@
M
"..15@
I CANNOT FIND M
"80@
?

"(.)..(.)
. THERE IS NO STAR
IN ALL THE SHROUDED HEAVENS ANYWHERE.
"(.)+2..(.)-1
THERE IS NO STAR
IN ALL THE SHROUDED HEAVENS ANYWHERE
"(HEAVEN)+1
EAVENS

"(C)..(D)
CANNOT FIND
"<..>
CANNOT FIND
"
CANNOT FIND
"<
C
"
C
```

```
"(C)..(D)
CANNOT FIND
">
D
"
D
"(C)..(D)
CANNOT FIND
"<+2..>-2
NNOT FI
"(C)..(D)/<+3..
NOT FIND
">-2..>+3
IND MY
```

ATE Text Editor

Design Philosophy

A text editor should be:

(a) easy to learn

(b) easy to use

(c) versatile enough to edit any kind of text.

Most editors available today fail at least one of the above tests. For example, the most common kind of editor (from a hobbyist's point of view) is probably the simple line editor present in all BASICs and in most assembly language systems. This is just dandy as far as (a) and (b) above are concerned, as long as you limit your text to relatively small computer programs. But large programs are difficult and time-consuming to edit this way. And trying to edit non-line-oriented text (such as this) is ridiculous.

At another extreme are well known editors (supported on large computer systems) such as QED and its descendents. They are relatively easy to use--once you learn how. But to the non-initiate, they appear as such an ad-hocery of special conventions that few take the time to learn to use them efficiently. And even these editors usually leave something to be desired as far as (c) goes.

ATE is an attempt to achieve (a), (b), and (c) in one editor. (a), by syntactical logic and consistency more typical of a programming language than an editor; (b), by keeping verbosity to an absolute minimum and allowing immediate as well as programmed execution of all functions; and (c), by imposing almost no restrictions on the contents of a file and providing a very powerful and general text-addresssing capability.

Basic editing in ATE

There are really only a few operations involved in actual text editing. Consider what is involved in writing a rough draft by hand:

Entering (inserting) new text.
Writing the text. ^ ~~Killing (deleting) text.~~ / Moving existing text
to a new location. / Corr~~a~~ecting existing text.

The operations illustrated here boil down to 3 basics: entering, killing, and moving. (Writing the text in the first place is simply entering into an initially empty file. Correcting consists of killing and then entering the replacement.) What is needed to make these 3 basic operations work effectively is a means of viewing the text and indicating what is to be moved or killed, what the destination of the move is, or where an entry is to be made. So we come to

Text Addressing

The examples in this section assume that we already have a text file in memory. Initialization of new files and file storage and paging will be covered later. Also, ATE commands will be covered in more detail later. Most commands require an argument (or value). All the commands use the same argument format, and this section is devoted to explaining that format.

Suppose the file we are editing contains the following characters:
WRITING_THE_TEXT. $^{C}_{R}$ ENTERING_NEW_TEXT.
Here we have used _ for a blank and $^{C}_{R}$ for a carriage return. When printed out, the file would appear:

WRITING THE TEXT.
ENTERING NEW TEXT.

Suppose we wanted to eliminate the word THE from the file. The easiest way to do this would be to use the Kill function as follows:

K(THE)

To understand how this works, we need the concept of an interval. Suppose we knew that the characters THE occupied memory addresses 2009H, 200AH, and 200BH. (H = hex.) Another (seldom used) way to kill THE would be to type

```
K 2009H...200BH
```

(THE) and 2009H...200BH are both legal arguments to any ATE command, and in this case, both would evaluate to the same thing: an interval beginning at address 2009H and ending at 200BH. The evaluation process is quite different for these two arguments, however. In the second case, ATE would not look at the contents of the file at all. It would simply evaluate the two numbers (which we could also have given in decimal or octal, straight or split; but more of that later) and pass those two values to the Kill routine. The other argument involves a process called matching.

When ATE encounters an expression such as (THE) in a command argument, it searches the file for a string of characters that match the ones enclosed in parentheses. If found, the beginning and ending address of the string become the values of the expression (THE).

Suppose we want to kill THE, TEXT, and ENTERING, leaving in the file only WRITING NEW TEXT.
Here is one way to do it.

```
K (THE)...(ING )
```

Note: Including the blank after the ING keeps the file from being left with two blanks between NEW and TEXT. We could just as well have typed K (THE)...(ING). Leaving a blank between the command K and its argument is optional.

To evaluate this argument, ATE first finds a match for (THE), and then searches forward from there to find a match for (ING). Then it combines these two intervals into a single one extending from the beginning of the first to the end of the second. (The ... operator can of course be shortened to .)

As may now be apparent, all ATE expressions evaluate at an interval. For example, 2009H evaluates to an interval beginning and ending at that address.

Suppose we want to kill the word TEXT in our file. It is apparent that we need to be able to distinguish different occurrences of the same string. To kill the *first* occurrence of TEXT, we need only type K(TEXT). To kill the second occurrence and not the first (assuming we are sure that it is the second occurrence we are after): K 2(TEXT)

The argument 2(TEXT) matches the 2nd occurrence of TEXT. In general, N(   ) matches the Nth occurrence of the enclosed characters, as long as the value of N is *positive* (less than 32,768). -1(   ) will cause the search to proceed backwards from the end of the file, matching the first occurrence in that direction. -2(   ) matches the second occurrence from the end, etc.

But what if we aren't sure how many occurrences of TEXT there are, and we don't want to count?
K (THE)...(ING)/(TEXT)
(This may look like a lot of typing, but things will get better.) This brings us to the concept of a *reference string*. When ATE begins evaluating an argument, it performs any required searches *within the current file*. Thus, we say that the current file is the *initial reference string*. But when ATE encounters a /, it takes the interval that has been calculated so far and makes *it* the reference string. Any subsequent searches will be performed within this new reference string.

The / operator may be used repeatedly, eg
K (THE)...(ING)/(TEXT)/(E)
This will kill the E that is within TEXT that is within THE...ING.

To kill all occurrences of TEXT within the file, we use the Repeat command, which is explained later.

Notice that giving a numerical address within an argument does not require a search, so the following is legal: 0...4095/(ABC). This will cause a search for the characters ABC within the first 4K block of memory. The initial reference string is still the current file, and this probably does not contain the interval 0...4095. But since 0...4095 does not call for a search, there is no problem of not finding this interval within the reference string. When the / is encountered, this interval becomes the reference string.

Note: The following example assumes some experience with machine language programming. Suppose that the first 4K block of memory does not contain an ASCII source file at all, but instead a version of BASIC whose I/O routines we are trying to modify. Then we wouldn't want to search for ASCII characters such as ABC, but for object code bytes such as DB 00 E6 (hex). (This is the object code for IN 0, ANI). We couldn't write 0...4095/(DB 00 E6), since this would search for the 8 enclosed ASCII characters. We need a delimiter other than parentheses to tell ATE to interpret the enclosed characters as numerical bytes. We use a number sign # for this: 0...4095/#DB 00 E6#

The bytes enclosed by the #'s must be expressed in the current operating base, which is set by the B command (see command summary). After typing B 8, we would type the above argument as 0...4095/#333 0 346# .

A feature of ATE that helps minimize typing: if a command needs an argument, but none is given, the interval computed for the last command is used. For example, here is a common editing operation. First we view an interval of text to make sure it is the one we want. To do this we use the quote command " .

```
" (THE)...(ING)          } we type this.
THE TEXT.                }
ENTERING                 } the computer responds with this.
```

The " command simply sends the addressed text to the terminal (inserting a line feed after a carriage return). After seeing the addressed text, we may decide to kill it:

```
K
```

Since no argument is given, it kills the most recently computed interval, namely (THE)...(ING).

Suppose that after seeing the text, we decide that we only want to kill the word TEXT within it (but not any other occurrence of TEXT within our file).

```
K /(TEXT)
```

Remember that / takes the most recently computed interval and makes it the reference string. (Otherwise, the current file is the reference string.) In this case, the most recently computed interval is THE...ING from the previous argument. So the match will occur within THE...ING, if at all.

Carriage returns occur frequently in most text, so there should be some way to address them. Trying to enclose one in parentheses will not work, since that would terminate the argument prematurely. We could use the numerical value of the character, #D# (hex) or #15# (octal), but a more convenient and mneumonic character has been provided: ← . For example, suppose we wanted to kill the first carriage return in our file (thus combining the 2 lines into 1).

```
K←
```

Carriage return addressing is one way to address lines in ATE. The nth line extends from the n-1st carriage return to the nth cr., not including the former. For instance, to quote the 3rd line, we could type

```
"2←...←             (although this will give us 2 cr.'s)
```

Recall that after a ..., the search proceeds forward from the interval calculated so far. To view 4 lines starting at the second cr., we could type

```
"2←...4←
```

Notice that this will quote 4 lines, not 2.

What if we want to kill the 3rd line without killing the cr. that precedes it? Here is one way (although there is an easier way that we will see shortly).

```
K 2←+1...←
```

To kill 4 lines starting just beyond the 2nd cr., we could type

```
K 2←+1...4←
```

Remember that all expressions in ATE evaluate to an interval, which is simply a pair of memory addresses. Interval+1 simply adds 1 to both of these addresses. Thus, using our original example, (TEX)+1 evaluates to the same thing as (EXT).

What does (WR)...(XT.)+1 evaluate to? Does the +1 apply to everything that comes before it, or just the (XT.)? Answer: just the (XT.). The + operator has a higher precedence than the ... operator, so it is performed first. Thus (WR)...(XT.)+1 extends from the W to the character after the period (which is a cr in this example).

Here is another way to get the same interval: (WR)...(XT.)←
This introduces another operator, <u>concatenation</u>. (XT.)← matches the first occurrence of the four characters X,T,period, and carriage return. 3(XT.)← would match the third occurrence of this four-character string. (As you can see, concatenation has a higher priority than ..., and a higher priority than "occurrencing".)

Some further examples: ←(PRINT)+1...← matches the first line beginning with PRINT. (We'll see an easier way to do this.) (ABD)(DEF) is equivalent to (ABCDEF). (ABC)←(DEF) is equivalent to (ABD)#D#(DEF). No blanks are permitted between elements to be concatenated. In fact, no blanks are permitted at all in ATE arguments, except within literals, as in (ING).

Another means of line addressing in ATE is the * operator. This takes an interval and expands it to a full line. In case the interval crosses a line boundary, it returns only the line containing the end of the interval. For example,

```
K3←*
```

kills the third line. The interval 3← is a 1-character interval contained in the 3rd line. (It is the carriage return at the end of this line.) * expands this to the entire line.

```
K←(PRINT)*
```

kills the line that begins with PRINT, not including the preceding carriage return.

"(TEXT)* quotes the line containing the first occurrence of TEXT, whether or not it begins the line.

"(TEXT)...(TEXT)*, using our original example, results in the computer printing

```
TEXT.
ENTERING NEW TEXT.
```

Notice that the * operator has a higher priority than the ... operator.

There are a few more special symbols to make text addressing easier. @ matches any single character (as long as there is one left in the reference string to match). (ABD)@(DEF) will match any occurrence of ABC_DEF, where the blank contains any <u>single</u> character. Contrast this to (ABD)...(DEF), which matches ABC thru DEF with any number of characters in between. Using an argument of 72@ would not only give the 72nd character in the reference string, it would test whether or not the reference string is at least 72 characters long. If not, the match would fail. (See the QF and QS commands for the consequences of match failure.)

What if we wanted to match the first 72 characters of the reference string? (72@ only matches the 72nd, not the first 72.) Here is one way:

```
"...72@
```

This quotes the first 72 characters of the current file (if there are that many). In general, ... at the beginning or end of an interval extends the match to the beginning or end of the reference string.

```
K 2←*/...(NEW)
```

will kill everything in the 2nd line up to and including NEW.

```
K 77←*...
```

kills everything in the current file from the 77th line on.

The symbols < and > refer to the left and right addresses of the most recently computed interval. They do not call for any matching, they simply return the appropriate address. K<+2...>-2 will kill everything in the most recently computed interval except the leading and trailing 2 characters.

When ATE begins computing an argument, a new value is assigned to < *as soon as* the left address of an interval is computed. This allows
"(DO YOU, MR. JONES?)...<+63, which quotes 64 characters starting with DO YOU... The symbol > retains its old value until the end of the interval is computed, either at the end of the argument or at a /.

Summary of text addressing

Most of the symbols below were covered in the preceding narrative. The rest are covered elsewhere as indicated.

Operands that invoke matching:

( )	Matches the enclosed ASCII characters within the current reference string.
# #	Matches the enclosed bytes (expressed numerically in the current operating base).
←	Matches a carriage return.
@	Matches any single character.

Operands that return values without matching.

number	1234 (decimal), 1234Q (octal), OF12AH (hex). May also be expressed in byte-oriented or "split" fashion: 123:456Q (octal), 123:456 (decimal). In both cases, the 123 gives the value of the

	high order byte, while 456 gives the value of the low order byte. 1:2Q = 001:002Q. Note that hex numbers are naturally byte-oriented: 12:34H = 1234H.
variable	X, ABC, S123, POINTER, etc. May be any length. 16-bit integer values. See the = command.
<	Left address of most recently computed interval.
>	Right address of most recently computed interval.
↑	Address of entry pointer. See the command summary.
<F>	The current file (beginning and ending addresses).
<S>	The source code area--all files.
<T>	The symbol table (see assembler).
<R>	The record most recently read in from tape.
?	The address of the most recent execution error (see programming).
' '	The numerical value of one or two ASCII characters, e.g., '3'=63Q.
&	The assembly program counter.
$	The assembly storage counter.

Operators:

Highest priority:

Concatenation	Applies when any matching-type operands are juxtaposed.

Middle priority (evaluated left-to-right):

+	Addition
-	Subtraction
*	Expands the end of the preceding interval to a full line. Must be preceded in the same argument by an interval-valued operand.
occurrencing	Applies when a value precedes a matching operand. 1+2(ABC) is equivalent to 3(ABC).

Lowest priority (evaluated left-to-right):

...	Combines two intervals into one. Can also be thought of as an operand that matches any number of characters. May be used repeatedly, e.g.,

(I HAVE)..(PAVEMENT)..(BEFORE). This would address the first 2 lines of the song "On the Street Where You Live," where (I HAVE)..(BEFORE) would address only the first line. Values connected by ... must be in non-decreasing order. 5..3..7 would fail. See the QF and QS commands. ... may be shortened to a single .

Takes the most recently computed interval and makes it the reference string.

An example

The following page presents a "typical" editing session with ATE (typical in its use of the editing commands, not in the inanity of the text). The commands presented are E (enter), K (kill), M (move), R (repeat), ↑ (set the entry pointer), " (quote), and ' (quote one line). As each command is introduced, please refer to the command summary for a detailed explanation of what it does. Even more importantly, please make sure you have read the preceding section on text addressing.

The > character at the beginning of a line is a prompt issued by ATE when it is ready to receive a command. Note that blanks are optional everywhere except within command arguments, where they are illegal except within literals.

```
>E(ENTERING NEW TEXT                                NOTES
WRITING THE TEXT
KILLING AND MOVING TEXT
)                                                     1
>"..                                                  2
ENTERING NEW TEXT
WRITING THE TEXT
KILLING AND MOVING TEXT
>↑.., M (WR)..(K), ".                                 3
WRITING THE TEXT
KENTERING NEW TEXT
ILLING AND MOVING TEXT
>K(K),K(IL).., ".
WRITING THE TEXT
ENTERING NEW TEXT
>↑(THE), '                                            4
WRITING ↑THE TEXT
>E(IN ),'                                             5
WRITING IN ↑THE TEXT
>↑↑+1,'                                               6
WRITING IN T↑HE TEXT
>↑+7,'                                                7
WRITING IN THE TEXT↑
>E( BOOK
IS FORBIDDEN), "..2←                                  8
WRITING IN THE TEXT BOOK
IS FORBIDDEN
>"3←*                                                 9
ENTERING NEW TEXT
>K/(TEXT), "..                                        10
WRITING IN THE TEXT BOOK
IS FORBIDDEN
ENTERING NEW
>'                                                    11
ENTERING NEW ↑
>K ↑-1, '                                             12
ENTERING NEW↑
>E(CASTLE
IS RISKY), "..
WRITING IN THE TEXT BOOK
IS FORBIDDEN
ENTERING NEWCASTLE
IS RISKY
>R99,K(IS),E(WILL BE)                                 13
?
>"..
WRITING IN THE TEXT BOOK
WILL BE FORBIDDEN
ENTERING NEWCASTLE
WILL BE RWILL BEKY
>K 3(WILL BE), E(IS), "..                             14
WRITING IN THE TEXT BOOK
WILL BE FORBIDDEN
ENTERING NEWCASTLE
WILL BE RISKY
>↑(IN ),E("),M(ENT)..(KY),E(" ),".                    15
WRITING "ENTERING NEWCASTLE
WILL BE RISKY" IN THE TEXT BOOK
WILL BE FORBIDDEN

>                                                     16
```

Notes

1. When ATE is first powered up, it initializes an empty file (among other things), issues a prompt character > , and waits for a command. In this case, the first command is an Enter. After typing the third line, we typed a carriage return <u>before</u> typing the closing parenthesis. This isn't strictly necessary, but if a file contains more than one line, it is good practice to end every line, including the last, with a carriage return.
2. In this example, we have used two dots .. for readability. Most often, you will probably only want to use one dot for brevity.
3. ↑.. sets the entry pointer to the beginning address of its argument. Since .. matches the entire reference string, and since the reference string is the current file unless otherwise indicated, ↑.. sets the entry pointer to the beginning of the file. This pointer also indicates the destination of an M (move) command, as this example shows. (After the move, the pointer is updated to the end of the inserted material, ready for additional entries or moves. See #15 below.)
4. This introduces the single-quote ' command. See the command summary.
5. Note that we inserted 3 ASCII characters, and that the pointer was updated past the inserted material.
6. The character ↑, when used as a command, means set the pointer to the beginning of the following interval. When used in an argument, it returns the (old) address of the pointer. Thus ↑↑+1 is analogous to X=X+1 with the = command. (But note that the ↑ command does not use an equal sign.)
7. This shows that ATE will recognize ↑+7 as an abbreviated version of ↑↑+7. Note that the entry pointer is now set between the letter T and a carriage return.
8. Here we have asked ATE to quote everything in the file up through the 2nd carriage return, i.e., the first two lines.
9. Quote the line containing the 3rd carriage return, i.e., the third line. See the explanation of * under <u>text addressing</u>.
10. Recall that / takes the most recently computed interval (in this case the 3rd line) and makes it the reference string. This way, we won't kill any other occurrence of TEXT within the file.

11. Note that no matter where it was before, the entry pointer is left in the position of the deleted text.
12. Here we are killing the character just before the pointer, again leaving the pointer in the place of the deleted text. (The deleted character was a blank.)
13. R means repeat. After 3 times, ATE could find no more occurrences of IS, so it responded with a ?.
14. Here we are saying "kill the 3^{rd} occurrence of WILL BE". In a large file, we probably would have quoted the offending text, and then used the restriction operator / .
15. Notice that the entry pointer is continually updated throughout this process.
16. We have left 2 carriage returns at the end of the file, as we can see by the empty line that was quoted there. See the programming commands for an example of an editing macro that can be used to clean up a file that has accumulated adjacent carriage returns and blanks.

ATE Command Summary

Line Typing

Rubout	Deletes the last character, echoing a back-slash \ .
Control-Z	Deletes the line being typed, echoing a back-slash, carriage return, and line feed.
Control-A	While ATE is typing output or running a program, hitting any key will halt the current process (without otherwise affecting it) and wait for you to peruse the output (or whatever). At this point, typing control-A will abort the process and return control to the terminal. Hitting any other non-printing character will continue the process where it left off.

Note: With the exception of the above three characters, ATE will ignore whatever is being typed until the line is terminated by a carriage return. But if the line being typed exceeds 72 characters, ATE will echo a back-slash after each additional character to indicate that the character has been lost. The "terminal width" can be set to less than 72 characters if desired (see the section on initialization). If the line being typed exceeds the terminal width, ATE supplies a new line (crlf) automatically and lets you continue typing the line (until the 72 character limit). The automatic carriage return and line feed will not be part of the line.

The terminal width can be set to exceed 72 characters in order to obtain a wider printout, but input lines will still be limited to 72 characters.

If you set the terminal width too small, the printout will not be wide enough to accommodate an assembly listing. In that case, see the address LISTR in the appendix for a patch.

ATE Command Summary

Note: Most of ATE's power lies in the arguments that you can give to its commands, not in the commands themselves.. All command arguments have the same format, which is explained in the section on text addressing.

Basic Editing Commands

↑ "Set pointer." ATE maintains an entry pointer to indicate where text is to be entered, or what the destination of a Move or Copy instruction is. Using ↑ as a command will assign the beginning address of the argument to this entry pointer. For instance, suppose that WRITING is the first word in the current file. ↑(TING) will leave the entry pointer pointing to the letter T. However, it is customary to think of ↑ as pointing between the T and the preceding I, WRI↑TING, since that is where entered text will appear. So when you type ↑(TING), it helps to think "set the pointer to precede TING."

The ↑ character may also be used in an argument to reference the entry pointer address. For instance, continuing the above example, typing "↑ would cause ATE to respond with a T. Typing ↑↑+1 will increment the pointer. As a special case, ATE will recognize the command↑+l as an abbreviation of ↑↑+1, not as an attempt to set the pointer to the absolute address 1. (To do this, you could type ↑1.) You may use an absolute address (a number or a variable, instead of a matching operand) and set the pointer outside of your text files altogether. In this case, the Enter command will behave somewhat differently. See the Enter summary for a full explanation.

As is the case with all ATE commands that take an argument, ↑ may be typed without an argument, in which case the most recently computed argument will be assumed. For instance, after typing "3←* to view the 3rd line, we could type ↑,E(LABEL) to insert LABEL at the beginning of this line. Or we could type ↑>,E(COMMENT) to append COMMENT to the end of the line. In fact, after viewing the 3rd line with "3←*, we could type the following:

```
↑,E(LABEL),↑> ,E(COMMENT)
```

to accomplish both these operations. To understand why this works, read about the Enter and Move commands. In a nutshell: the Enter command does not change the default argument. But it may cause text to be moved (to expand the file), and whenever ATE moves anything that it knows about (such as >, <, the file, the symbol table, etc.), it remembers the new location.

The only commands that affect the position of the entry pointer are: ↑,E,K,M,C,F,N,O, and L. ↑ sets it to the beginning of the addressed interval, K sets it to the deletion point, and all others set it to the end of the interval in question.

E "Enter." Text which follows the E command (as long as it is properly delimited) is entered at ↑, and ↑ is set past the entered material, ready for continued entry. The text to be entered must be delimited in the same manner as for matching-type operands: parentheses around ASCII characters, and number-signs around numerically expressed bytes. These two types can be concatenated (with no intervening space), as can the symbol ← for carriage return.

However, in contrast to matching-type operands, the delimited text can be many lines long, i.e., it can contain carriage returns. Also, no matching takes place. ATE does not compute any new argument values for the E command, so the default argument (the most recently computed interval) remains unchanged.

The interpretation of what it means to "enter the text at ↑" is necessarily different depending on whether or not ↑ lies within a text (source) file. If it does lie within a source file, then we think of ↑ as pointing *between* characters (just before the one it was addressed to). The source file (and any adjacent files in the source code area--see the memory file commands for an explanation) is expanded and the given text is inserted. If ↑ is not in any source file recognized by ATE, then the given text simply overwrites what is already in memory, beginning right *at* ↑. This latter operation is used most often in editing object code (= machine language program). For an example of this, see the # command.

It is legal to include parentheses in the ASCII text to be entered, as long as they are balanced, i.e., as long as they occur in matched pairs. For instance, E(LET X=SIN(Y)) is legal. It will enter all but the outer-most parentheses. In ATE, this feature is often used as follows. We might type:

```
E(*BLANKS, QF(  ), K<,  (*BLANKS))
```

This enters a string of ATE commands (most of which we haven't covered yet) into a file. Later, we might ask ATE to execute these commands. Their effect would be to eliminate multiple blanks in whatever file was then current. This is called an edit macro, and will be covered with the programming commands.

Error handling: suppose that the current operating base is 16, and you type E#C3 12 AX# . ATE will enter the C3 and the 12, but it will not recognize the X as a legitimate base 16 digit. ↑ will be updated past the entered 12, and an error sign ? will appear at the terminal. Then you could type E#AC# (if AC was your intention) to complete the operation.

K "Kill." The addressed text is deleted, and ↑ (no matter where it was before) is set to the deletion spot. Thus a Kill followed by an Enter will replace the deleted text, with no need to use the ↑ command. For instance,
K(SAMUEL CLEMENS), E(MARK TWAIN) replaces the first occurrence of SAMUEL CLEMENS With MARK TWAIN. R999, K(SAMUEL CLEMENS), E(MARK TWAIN) will replace all occurrences (unless the text is extraordinarily repetitious of that name). (See the Repeat command.)

As with Enter, Kill behaves differently when used to edit object code, i.e., memory data outside a text file. If the addressed text is *inside* a source (text) file, then it is moved outside the source file area, and the remaining source code is compacted to fill the gap. (Thus the killed text is not actually overwritten, and can be retrieved, until the next Enter causes the source area to expand.) But if the addressed interval is *outside* the source area, then it is *zeroed*, i.e., the addressed interval is filled with zeros, and ↑ is left pointing to the first zero.

Machine language example: Suppose we have a version of Basic whose input routine we want to change. From the documentation, we know that it does teletype IO using the Processor Technology standard ports (0 for status, 1 for data). We could load Basic at, say, 3000H and then type the following:

```
K 3000H..5000H/#DB 00 E6#@@@@#DB 01#
E #CD E1 23#
```

What we have done is put zeros (NOPS) in place of the old input code, and then put in a call to our new input routine at the beginning of this zeroed section. We used the four @'s so that we didn't have to worry about what the mask was, whether the jump was a JZ or JNZ, and what the jump address was.

In actual practice, we would want to look at the code before we zeroed it. See the # command for an example of this.

M "Move." The addressed text is removed from its present location and inserted at the entry pointer. The pointer is updated past the inserted material, ready for additional moves or entries. If the addressed text included any of the internal pointers known to ATE, then these are updated to their new location. These are: <F> (the current file), <S> (the source area), <T> (the symbol table), <R> (the record most recently read in from tape), < (the left address of the most recently computed interval), > (the right address of the same), the user command table, the command interpretation pointer (in case a macro is currently executing), and any return or repeat addresses. The Kill command uses the same subroutines to move text out of the source area, and the Enter command uses them to expand the source area if necessary.

Example: To put lines 10 through 12 between lines 2 and 3,
↑3←*, M 10←*..2←   Or we could type
↑2←+1, M 9←+1..3←
In either case, if we then typed a " command without an argument, we would see the text that had been lines 10 through 12, but in the new location. ↑ will now be at the end of these lines, but if we want it at the beginning, all we have to do is type ↑ without an argument (or type ↑<).

As with the Enter command, Move behaves somewhat differently when used to edit object code. If ↑ is outside the source area, then the addressed interval is simply copied to ↑, overwriting what is already there. This happens no matter where the addressed interval is, and the original interval remains unchanged (unless ↑ was within it). For example, to move the symbol table up 100 bytes: ↑<T>+100, M<T>
If you didn't want ATE to know about the new copy of the symbol table, you would use the Copy command.

C "Copy." This is a seldom-used command. It copies the addressed interval to ↑, overwriting what is already there, and updates ↑ past the copied material. There are two differences between this and the Move command. (A) Copy is insensitive to whether or not ↑ lies within source code. Even if it does, Copy simply overwrites what is there; it never expands and inserts. So you will probably not want to Copy anything into a source file.
(B) Copy "hides" the new location of the copied material from ATE, i.e., none of ATE's internal pointers are updated. One

example of when you might find this useful: you might want to create a duplicate copy of (say) the current file somewhere else in memory, without having that copy actually become the current file.

Printing Commands

" "Quote." Sends the addressed characters to the terminal, inserting a line feed after each carriage return. For example

```
>"(THE)..(NEW)
THE TEXT
ENTERING NEW
```

If the terminal prints garbage in response to this command, you have probably given it an argument outside your source files. In that case, you probably wanted to have the characters expressed numerically. Use the # command for this.

' "Quote one line." Takes no argument. This quotes the line containing the entry pointer, showing the position of this pointer by a ↑ character. The ↑ appears just before the position to which it is addressed, since this is where Entered or Moved text will appear. For instance:

```
>↑(E T), ', E(ES), '
WRITING TH↑E TEXT
WRITING THES↑E TEXT
```

Using ' does not affect the default argument.

P "Print." This is mainly used with assembly language programs. It prints the lines containing the addressed text in assembly language format. Each line is given a line number (which is not part of the text). The first line of the file is 1, the second is 2, etc. Here is an example that shows the difference between " and P.

```
>"(IN )..(MASK)
IN STATUS GET THE STATUS BYTE
 ANI MASK
>P
   79   INCHR  IN    STATUS     GET THE STATUS BYTE
   80          ANI   MASK       SCREEN OFF IRRELEVANT BITS
>"
IN STATUS GET THE STATUS BYTE
 ANI MASK
```

Note that P only prints entire lines, even if the argument does not come up to line boundaries. But this does not change the default argument--it remains exactly as typed. (No implicit * operation is performed.)

P can be used to get the line number of a non-assembly-language line, even though the output will be strange. (The listing can be control-A'ed.)

B "Base." Sets the current operating base to the given value. This does not produce any output itself, but subsequently all numerical output will be generated in this base. The only effect on input is that numerically-expressed bytes delimited by #'s in command arguments must be expressed in this base.

The base may not be less than 5. Typing B8 gives you octal, B10 gives decimal, and B16 gives hex.

"Quote numbers." This is similar to the "command, except that the characters are expressed numerically in the current operating base. The beginning address of the interval is printed, followed by up to b (=base) bytes, and then followed if necessary by more such lines. (This command is often called "Dump" in other systems.)

Machine language example: (This expands the example given with the Kill command.) Suppose we have a version of Basic whose input routine we want to modify. From its documentation, we know that it comes set up for teletype IO using Processor

Technology standard ports (0 for status, 1 for data). We can load Basic at (say) 3000H, and then look for the machine language code for IN 0, ANI __, followed by IN 1.

```
>#  3000H..5000H/#DB 00 E6#..#DB 01#
31DA  DB 00 E6 80 CA DA 01 DB 01
>K, E#C3 #1 23#, #
31DA  C3 E1 23 00 00 00 00 00 00
```

After looking at the code and deciding that it was what we wanted, we zeroed it, entered a call to our new input routine, and then looked at it again to confirm that the change was made correctly.

? "Where?" Prints the beginning and ending addresses of the argument. For example, ?.. will give the addresses of the current file. (Note: this works unless the file is empty. .. is a matching operator, and it will fail if the reference string is empty. On the other hand, <F> is an operand that returns values without matching, so ?<F> will always work.)

Memory File Commands

ATE keeps its text files adjacent in one area of memory, called the "source area," denoted <S>. The left address of <S> is ordinarily fixed (although it may be changed by M<S> or by an O command), while the right address varies dynamically in response to enter and kill commands. <S> consists of a zero byte, followed by the first file, followed by another zero byte, followed by the second file, etc., ending with a zero byte. There is no limit on the number of files. No separate file directory is maintained; files can be accessed (via the F command) by addressing any of their contents. By convention, files can be "named" by entering a uniquely identifying name as the first line. (The name should be preceded by a * if the file is going to be assembled.) The F command can then address this name. No check is made to keep <S> from overflowing memory, but the user can periodically check its size by typing ?<S> .

F "File." This finds the file containing the given argument, makes it current, and sets ↑ to the end of the file. Unlike any command covered so far, the initial reference string of this command is <S>, not <F> . So for instance, F(WRITING) will find the first occurrence of WRITING within the source area, and then make the file that contains this current.

If for some reason you have memory files isolated from the source area (by use of the O command, perhaps, or by loading from a peripheral not recognized by ATE), you can access them by giving an absolute address with the F command. Suppose you have loaded a file from a disk at address 4000H. As long as the limiting zeros are in place, you can type F 4000H . ATE will look forwards and backwards for the limiting zeros. If the given address contains a zero, ATE assumes that it is the beginning of the file.

After finding the file boundaries, ATE checks the relationship of these boundaries to the source area. If the file is within the source area, it is made current, ↑ is set to the end, and nothing else happens. But in addition, if one edge of the file is within <S> while the other isn't, then <S> is expanded to include the new file. If the new file is entirely outside <S> , then ATE forgets the old source area (leaving it intact) and adopts the new file as the new source area. The old source area can be recovered later by repeated use of the F command, or with the O command.

N "New file." Opens a new, empty file at the top of the source area and makes it current, ready for entry. Specifically, a zero is written into memory just beyond the last zero in the source area. <S> is expanded to include this new zero, and the current file pointers and the entry pointer are set to the empty file between these two zeroes. Note that N does not take an argument.

O "Originate new source area." Sets up a new source area at the given address(es). If only one address is given, as in O 3000H, then two consecutive zeros are written into memory starting at this address (at 3000H and 3001H, for example). If two addresses are given, as in O 3000H..36DAH, then zeros are written at these addresses. In either case, the last (possibly empty) file of this new source area is made current, and ↑ is set to its end.

Programming commands

ATE can create (and edit) source files which are actually strings of ATE commands. Later, these commands can be executed by typing a D or > command addressed to the desired point.

D "Do." This is analagous to an assembly language CALL or a Basic GOSUB. ATE remembers the location of this command, computes the argument, and then starts executing commands at this address (the beginning address of the argument). In computing the argument, the initial reference string is <S> . ATE will continue executing commands until it encounters an end-of-file or one of the Quit commands, at which point it will return to the command following the Do. If there is no command following the Do and no previous Do to return to, or if an error is encountered, control returns to the terminal. See the end of this section for an example of an ATE program using Do's.

Do's may be nested. To see how deeply they can be nested, you could type the following:

```
 N, E(X=X+1, D(X=X+1)), X=0, D(X=X+1)
```

This line creates and then executes an ATE program. The trouble with this program is that it never lets ATE return from the Do. It keeps executing Do's until ATE runs out of storage for return addresses. At that point, ATE issues an error sign (a ?) and returns control to the terminal. (See "Error Handling" for more information.) You could then type #X or ?X to see how many Do's had been stored. (Note: the return address storage area is also shared by the Repeat command.)

Notice that when ATE sees D(X=X+1), it does not immediately increment X. Instead, it stores the return address, searches the source area for the first occurrence of the string X=X+1, and then begins executing commands at that point.

Rather than address commands directly, as in D(X=X+1), we usually address labels, as in D(*COUNT). See the * command below. For maximum brevity, we can use a variable to hold the absolute address of the desired command. We could say Y=(X=X+1) or Y=(*COUNT), and later type DY. This is useful for often-repeated edit macros (see the example at the end of this section). But then we must be careful not to change the absolute location of the command by some edit operation. Placing it at the bottom of the source area (or in an isolated source area) will ensure against this.

Caution: D(STRING) contains an occurrence of STRING. If you want to address an occurrence of STRING at some *later* point in the program, you could use D 2(STRING).

"Goto." Causes ATE to execute commands beginning at the given address. Like Do, > may be used within a program, or it may be used to start a program from the terminal. Unlike Do, no return address is stored, so ATE will never automatically return to the succeeding command. *As with D, the initial reference string is <S>* . (The only ATE commands for which this is true are F, D, and > . Note that the initial reference string never includes the command line being typed.) Here is a simple example. Note that the prompt character is not a Goto command.

```
>N, E("(HELP, I'M TRAPPED IN AN INFINITE LOOP ),>(")), >(")
```

This command line creates and executes a program that quotes HELP, I'M TRAPPED IN AN INFINITE LOOP indefinitely until control-A'ed.

R "Repeat." May be used in a program or in the command line. If used in a program, the rest of the file up to an end-of-file zero byte, or up to a Quit command (see below) is repeated the given number of times. (If this value is 0, the following commands are not executed.) When the repetition is exhausted, a <u>return</u> is performed (to an outer loop or to a Do, whichever is more recent, or to the terminal). For example:

```
 N, E(S=0,N=0,R 100,N=N+1,S=S+N), D(S=0), #S
```

This creates and executes a program that finds the sum of the integers 1 to 100.

When R is used directly in a command line, then the rest of that line will be repeated the given number of times before control returns to the terminal (unless an error or a control-A forces an early return). For instance, R999,K(SOON),E(IMMEDIATELY) will replace all occurrences of SOON with IMMEDIATELY and then return with an error sign ? when it can find no more. ↑0CC00H, R1024, E( ) will clear the screen of your VDM (i.e., it will fill 1K of memory, beginning at 0CC00H, with blanks).

***** "Label." This use of * is similar to its use in assembly language--it tells the system to ignore the following characters. (This has no relation to *'s use as a line operator in command arguments.) When ATE encounters a * as a command, it skips ahead to the next command, ignoring all intervening characters. Thus these intervening characters can be a mneumonic label for that point in the program. For example, we could create a program as follows:

```
>N, E(*BLANKS, K(  ), E( ), >(*BLANKS))
```

Later, whenever we wanted to eliminate double blanks from the current file, we could type

```
> >(*BLANKS)
```

We could use a shorter mneumonic, of course. This program has the defect that it will always end with an error, when no more double blanks can be found. So it cannot be called by a Do command with any hope of returning. For this we need the Quit commands.

Specifically, * causes ATE to skip ahead to the next blank, comma, carriage return, or end-of-file (zero), whichever comes first.

Error Handling: Ordinarily, *any* error in a command argument causes ATE to stop an execution, issue an error sign ?, and return control to the terminal. To see what caused the error, you could type

```
>" ?-5..?
```

This will quote 6 characters from the program, ending with the one that ATE was looking at when it gave up. (Of course, any other number of characters could be used.) For instance, suppose we executed *BLANKS given above.

```
> >(*BLANKS)
?
>" ?-5..?
K(  ),
```

This shows that ATE was unable to evaluate the argument to the K command, i.e., it could not find any more adjacent blanks in the current file.

QF "Quit on Failure." In general, both quit commands (QF and QS) mean "quit this subroutine." When the argument to QF is evaluated, a *match failure* or a *comparison failure* will not abort the program. Instead, it will cause a return to the latest Do or to the terminal. In performing this return, one Repeat loop will be broken, if present. If Repeats are nested, the outer loops will not be broken.

Examples: We can use this command to repair the defect in our program *BLANKS mentioned above (with the * command).

```
*BLANKS, QF(  ), K>, >(*BLANKS)
```

Now this routine can be called with a Do. As long as there are adjacent blanks in the current file, this will Kill one of them and loop. When it cannot find any more adjacent blanks, the QF will force a return instead of an error.

Since QF will break a repeat loop, we could also write *BLANKS this way: `*BLANKS, R9999, QF(  ), K>`

In addition to match failure, QF will force a return instead of an error on a comparison failure. Values connected by ... must be in non-decreasing order. So QF X..Y will succeed and continue if $X \leq Y$, and will fail and return if $X > Y$. More than two values at a time may be checked, as in QF X..Y..Z . QF X..Y..X will return if $X \neq Y$.

QS "Quit on Success." This is the same as QF above, except that it forces a return if its argument is successfully computed. If a match failure or comparison failure occurs in the argument, then execution continues. If any other kind of error occurs in the argument, the program is aborted and control returns to the terminal. For an example, see the end of the section on cassette commands.

= "Equals." This is the only command that doesn't precede its arguments. It is used in the conventional manner to set the value of a variable, e.g., X=X+1, POINTER=VALUE, S1=-1, etc. Blanks around the = are optional. Variables can be any length, must start with a letter, and may contain only upper case letters and digits. Values are 16 bit unsigned integers. (So S1=-1 is equivalent to S1=0FFFFH) Any ATE argument (including no argument) may occur to the right of the =. The variable on the left is assigned the beginning address of the (default) argument. F=2(X=X+1) assigns the address of the 2nd occurrence of the string X=X+1 to the variable F. X=, Y=> saves the current default argument in X and Y (although it creates a new default argument).

Variables are kept in a symbol table shared with the assembler. This allows you to set external references prior to an assembly, and to address object code symbolically after an assembly. See the Z commands for more information.

X "Execute." This is used to call machine language subroutines (as opposed to D which calls ATE subroutines). The machine language routine may end with a RET (as long as the stack has not been lost) in which case X can be part of any ATE command line or program just like any other ATE command. You have between 20 and 40 stack levels (40 to 80 bytes) depending on how deeply nested the Do's and Repeats are when the X is encountered. If your routine loses the stack, it should end with a jump to address SYS1 (see appendix). This returns control to the terminal ignoring any commands following the X.

Your routine can evaluate an ATE argument:

```
X ROUTINE X..Y, OTHER COMMANDS
```

Leave a <u>space</u> between the address of your routine (which you can set with an =, e.g., ROUTINE=3456H) and the argument you want to evaluate. X..Y can be any ATE argument. Your routine can CALL address CVALS (see the appendix). On return, HL and DE will contain the beginning and ending values of the argument. Additional arguments can be evaluated by repeated calls to CVALS, as long as the additional arguments are separated in the command line by blanks, not commas or carriage returns.

User machine language routines can also be accessed by entering a name and address for the routine into the <u>user command table</u> (explained later).

<u>Example of a useful ATE program</u>

This is a program (or "edit macro") to "clean up" an English language file after extensive editing. Unless you are quite careful when editing such a file, you will probably end up with lots of adjacent blanks and very short or long lines that will spoil the looks of the file when it is printed out. After creating the following program, typing D(*CLEAN) will eliminate multiple blanks and fix the carriage returns so that each line is ≤ LENGTH long. (Don't forget to set LENGTH first, e.g., LENGTH=72.)

*CLEAN uses three subroutines: *CRS replaces all carriage returns with blanks. *BLS eliminates multiple blanks. *LNS fixes the line length.

```
*CRS, QF←, K, E( ), >
*BLS, QF(  ), K>, >(*BLS)
*LNS, QF ↑..LENGTH@, K /-I( ), E←, >(*LNS)
*CLEAN, D(*CRS), D(*BLS), ↑.., D(*LNS)
```

This program can easily be extended to detect special symbols and replace them with new paragraphs (a carriage return and several spaces), or new pages (several carriage returns, depending on the number of cr's since the last new page).

Introduction to the ATE assembler

If you already have some experience with assembly language programming, you should skip ahead and read the Assembler Summary. If any of the summary is unclear to you, then come back and read this introduction.

If you haven't had any experience with 8080 machine language (in particular, if you haven't learned the instruction mneumonics such as CALL and XCHG and what they do), then you should read a text on 8080 machine language before continuing.

This manual assumes that you know at least enough about 8080 machine language programming to code the following subroutine:

> Take the byte in the memory location addressed by the HL register pair, add it to the byte addressed by the DE register pair, and store the result at the address in the BC register pair.

If you were doing all your programming by hand through the front panel keys or switches, you might first write down the mneumonics for the desired instructions, look up their values in a table, and key these values into memory:

Mneumonics	Hex	Octal
LDAX D	1A	032
ADD M	86	206
STAX B	02	002
RET	C9	311

For this little subroutine, there is not much work involved. But when you try to write larger programs this way, you begin to wish that your computer could do some of the busy-work for you. The first step--figuring out the instructions that will do thejob--is not always busy-work. Sometimes this may involve ingenuity. But the second and third steps are easily automated. Looking up mneumonics in tables and putting the values where they belong in memory are the major tasks of an assembler.

Example 1. Power up ATE and type the following. (The > sign at the beginning of a line is a prompt issued by ATE when it is waiting for a command. No prompt is issued while you are entering text. Notice that we put each mneumonic on a separate line, and precede each one with a blank. The reason for this will be covered shortly.)

```
>E( LDAX D
 ADD M                                        You type this.
 STAX D
 RET)
>G
0000  1A          1          LDAX D
0001  86          2          ADD  M
                                              ATE responds with this.
0002  02          3          STAX B
0003  C9          4          RET
```

Here is what we have just done: We Entered the mneumonics into a file in the computer's memory, and then we told ATE to Generate a machine language program from these mneumonics. ATE stored the resulting machine instructions in memory starting at address 0, and printed out the hex code for each instruction along with the mneumonic that produced it. (It also numbered these mneumonics for later reference.) Of course, we could have told ATE to use some other address than 0 --this will be covered later. And we could have told ATE to use octal or decimal, rather than hex, by typing B8 or B10 (see the B command).

In order to use the subroutine that we have just written, we of course need to call it. We don't want to write CALL 0 each time, however. We want to give the subroutine a name and write the following:

```
        CALL MADD
        .
        .
        .
MADD    LDAX D
        ADD M
        STAX B
        RET
```

This way, we don't have to know the address of the subroutine while we are writing the program--we can let ATE figure this out later.

Example 2. Type the following. (Notice that we are using some of the text editing features of ATE. We will explain them briefly in the notes below. They are described fully elsewhere in this manual.)

```
                                                  notes
>↑.                                               1
>E( CALL MADD
*                                                 2
*
MADD)
>".                                               3
 CALL MADD
*
*
MADD LDAX D
 ADD M
 STAX B
 RET
>P.
  1          CALL MADD                           4
  2    *
```

```
3     *
4     MADD   LDAX D
5            ADD  M
6            STAX B
7            RET
```

Notes (1) ↑ stands for the "entry pointer", i.e., the position in the current file at which new text will be entered. The dot "." as it is used here stands for the current file. ↑. tells ATE to position the entry pointer at the beginning of the current file. Note that the current file already contains the mneumonics from example 1.

(2) An * at the beginning of a line (i.e., not preceded by a space) has a special meaning to the ATE assembler. We can put anything we want on the rest of the line (including nothing), and the assembler will ignore the whole line. This lets us comment our programs and "space out" the instruction mneuonics.

(3) ". tells ATE to quote the current file. This shows us the file as it resides in memory. Notice that the instruction mneumonics are preceded by blanks, but the *'s and the subroutine label MADD are not preceded by blanks.

(4) P. tells ATE to print the current file. Spaces and line numbers are added to the printout to make the assembly language easier to read.

Example 3. Now let's assemble this program.

```
>G
0004  CD 00 00   A  1          CALL MADD
                    2    *
                    3    *
0007  1A            4    MADD  LDAX D
0008  86            5          ADD  M
0009  02            6          STAX B
000A  C9            7          RET
```

The "A" in the first line of the program listing is an *error message*. It stands for *argument error*: ATE did not know the value of the argument to the CALL instruction, i.e., it did not know the address of the MADD subroutine.

The problem is this: the assembler looks at the program one line at a time, beginning with the first line. When ATE saw CALL MADD, it had not yet come to the subroutine labeled MADD, so it did not know what address to use with the CALL instruction. So it simply used an address of 0 and flagged an error.

This is often called the "forward reference" problem. One way to solve it would be to have ATE look forward through the program, counting instruction bytes until it comes to MADD. Then it could go back to the CALL MADD instruction with the correct address. But the trouble with this is that the program might be on tape (if it were too large for memory), and moving tape back and forth is very time consuming.

To get around this, we would like ATE to look over the *entire* program once, before it begins to print anything. This way it can figure out the address of the subroutine MADD (and any other subroutine) before it actually needs it. It can store the label MADD together with the proper address in a *symbol table*. Then later when it sees CALL MADD, it can look in this table to find the appropriate address.

What this all boils down to is that the assembler can make *two passes* over our program. On pass 1, it reads through the program and constructs a symbol table, i.e., a list of labels and their corresponding machine language addresses. Then on pass 2, it rereads the program and actually generates the machine language instructions and stores them in memory.

The command A tells ATE to do pass I. (It stands for *A*ssemble the symbol table.) It is not necessary if the program has no labels (as in example I) or if the symbol table is already in memory (possibly restored from tape). The command G (for *G*enerate machine language) tells ATE to do pass 2. We can command ATE to do both passes by typing A,G . (Most assembly language systems don't let you command the two passes separately. They use a single command such as ASSM where ATE would use A,G. But there are real advantages to the A,G approach, as we will see.)

Example 4. The current file still contains the same program as in example 3. Suppose we now type

```
>Z,A
```

The first command Z (Zero the symbol table) simply makes sure that we are starting out with a clean slate. It removes any old symbols that might be left over from a previous programming session. (Sometimes we want to save these symbols. More about this later.) The A command then does pass 1 over our program and puts MADD (and any other labels) into the symbol table. (Notice that this does not produce any printout.) To see that ATE does know the address of MADD now, we can type

```
>?MADD
000E 000E
```

(To see why ATE responds with two values, try typing ?MADD..MADD+9)

Now we can do pass 2 over our program.

```
>G
000B  CD 0E 00      1              CALL MADD
                    2     *
                    3     *
000E  1A            4     MADD     LDAX D
000F  86            5              ADD  M
0010  02            6              STAX B
0011  C9            7              RET
```

This time there was no error. Note: we could have typed Z,A,G all on one line to accomplish the same thing.

How do we tell ATE where in memory to store the assembled machine language? Actually, there are two problems here. (a) Where in memory will the machine language program be located when it is executing? and (b) Should the program be temporarily stored somewhere else first? For instance, we might want to assemble a program that will begin executing at address 1000H. But ATE itself begins at 1000H, so we would want the assembler to store the new machine language somewhere out of the way until we are ready to use it.

The symbol & stands for the "assembly program counter" (remember that they both begin with an "a"). This holds the execution address of the instructions being assembled. The symbol $ stands for the "storage pointer" (remember that they both begin with an "s"). This holds the address where the assembled instructions are being stored. (In many assembly language systems, $ stands for both; the two uses cannot be separated.) For example, we can use & and $ as commands:

```
>&1000H,$0D00H
```

This tells ATE to set the assembly program counter to 1000H, and set the storage pointer to 0D00. Now, the next program to be assembled will be stored at 0D00. But it will have to be loaded at address 1000 in order to execute properly. (In more detail: the next A command will assemble the symbol table assuming that the program begins at address 1000. Thus all CALL and JMP addresses will be based on this starting address. The next G command will use the symbol table and generate machine language, but will store this machine language beginning at 0D00.)

Example 5. Let's write a new program.

```
>N
>E(FIRST LXI H,1234H
SECOND MVI A,1
 JMP FIRST
 JMP SECOND THIS IS A SILLY PROGRAM
)
>&1000H,$0D00H,Z,A,G
```

```
1000  21 34 12     1    FIRST  LXI  H,1234H
1003  3E 01        2    SECOND MVI  A,1
1005  C3 00 10     3           JMP  FIRST
1008  C3 03 10     4           JMP  SECOND    THIS IS A SILLY PROGRAM
```

Notice that the assembler allows us to fill out a line with comments. The listing shows the machine code at the addresses for which it is assembled, not where it is stored. We can check this:

```
>#0D00H..0D0AH      (The number sign # command is called DUMP on most systems.)
0D00   21 34 12 3E 01 C3 00 10 C3 03 10
>#1000H..100AH
0000   C3 5F 1D 31 C4 0E CD 51 14 CD C3
```

We can see that the assembled machine language was stored at 0D00, not at 1000 which still holds the beginning of ATE.

There is another way to tell ATE where to begin the assembly or where to store the object code. (Note: object code = assembled machine language.) We can put instructions to this effect right in our assembly language program:

Example 6

```
>↑.
>E( AORG 1000H
 SORG 0D00H
)
>Z,A,G,
1000               1           AORG 0
1000  0D00         2           SORG 0D00H
1000  21 34 12     3    FIRST  LXI  H,1234H
1003  3E 01        4    SECOND MVI  A,1
1005  C3 00 10     5           JMP  FIRST
1008  C3 03 10     6           JMP  SECOND    THIS IS A SILLY PROGRAM
```

AORG (Assembly ORiGin) tells ATE to assign the following value to the assembly program counter. SORG (Storage ORiGin) tells ATE to assign the following value to the storage pointer. These two assembly language instructions are called pseudo operations, since they are not actual CPU operations.

Another pseudo-op, ORG, affects both & and \$. (It is included mainly for compatability with other systems, which don't have AORG and SORG.) Using ORG 2000H in a program will increment & to 2000H, and then increment \$ by the same amount (not necessarily to 2000H). That is, after the assembler sees ORG 2000H, then $\&_{new} = 2000H$, and $\$_{new} - \$_{old} = \&_{new} - \&_{old}$. One reason for this is to allow you to use ORG to reserve storage space in the middle of your machine language program. For instance, ORG &+100 would reserve 100 bytes.

An easier way to reserve memory space is to use the DS (Define Storage) pseudo-op. DS 100 will reserve 100 bytes. DS, however, does not put any information into the reserved space. To do this, use DB (Define Byte), or DW (Define Word).

Example 7

```
>N,E( LHLD ADDRES
ADDRES DW SECOND,1234H
 DB 12H,34H,'A','B'
)
>&0D00H,$,A,G,
0D00  2A 03 1D      1
0D03  03 1D 34 12   2    ADDRES DW    SECOND,1234H
0D07  12 34 41 42   3           DB    12H,34H,'A','B'
```

There are several things to notice in this example:

(a) In the command line &0D00H,\$,A,G, we didn't give any argument after the \$ command. As always, whenever a command argument is missing, ATE uses the argument from the last command. We could have typed &0D00H,\$0D00H, A,G, but that would have been redundant.

(b) We didn't Zero the symbol table before we gave the A command, so the symbols FIRST and SECOND (from the last example) are left in the table, along with their values of 1000 and 1003. So when the assembler sees DW SECOND, it assembles that correctly.

(c) DW reverses the natural order of a two-byte word, as required by the 8080.

(d) DB can be used to put ASCII characters into the program, as shown above, but an easier way is to use the ASC pseudo-op:

Example 8

```
>N,E( ASC HELLO
 ASC- BY-BY
)
>G
0D0B   48 45 4C 4C 4F  1           ASC  HELLO
0D10   42 59 20 42 59  2           ASC- BY-BY
```

Note that to insert a space (blank) character in the ASCII string, we signal that a dash (or any other non-alphanumeric character) will stand for a space by putting the dash right after the ASC.

Example 9. Another important pseudo-op is EQU.

```
>K.
>E( LXI H,ADDRES
ADDRES EQU 1234H
),Z,A,G,
0D15  21 34 12         1           LXI  H,ADDRES
      1234             2    ADDRES EQU  1234H
```

Notice that in this example, we killed the contents of the current file and put new text into it, rather than leaving the old file in memory and starting a new one as we did before.

The EQU pseudo-op puts the statement label (such as ADDRES above) into the symbol table, and gives it the stated value. We can use the name ADDRES many times in the program, and if we ever want to change its value, we need only change the EQU statement. (However, we cannot change the value of ADDRES from one thing to another within the same program, i.e., we can have at most one EQU statement for each label.)

<u>Example 10</u>. The last pseudo-op is END.

```
>E( END
THE REST OF THE FILE CAN HAVE ANYTHING IN IT.  THE ASSEMBLER WILL NOT GO
BEYOND AN "END" PSUEDO-OP.)
>".
 LXI H,ADDRES
ADDRES EQU 1234H
 END
THE REST OF THE FILE CAN HAVE ANYTHING IN IT.  THE ASSEMBLER WILL NOT GO
BEYOND AN "END" PSUEOD-OP.
>Z,A,G
0D18  21 34 12        1          LXI  H,ADDRES
      1234            2   ADDRES EQU  1234H
0D1B                  3          END
```

<u>Assembly language format rules</u>:

The assembly language program (or "source code") is a text file containing lines, or "statements". Each line looks like this:

label opcode argument comment
  <u>or</u>
* comment

(1) Lines may <u>not</u> begin with a line number, as they must in some systems. However, a line number will be supplied with the program listing.
(2) If the <u>first</u> character (not the first non-blank character) is an *, then the rest of the line is taken to be a comment.

(3) The label is optional. If the line does begin with a label, then the first character in the line must be the first letter of the label. If the line has no **label**, then the first character must be a blank. The label must begin with a letter, and can contain only upper case letters and digits.

(4) The opcode can be either a machine instruction or a pseudo-op. It must be preceded by a blank.

(5) An argument is necessary for some opcodes. It must be preceded by a blank, and it cannot contain blanks except within quotes, such as MVI A,' ' . Here, MVI is the opcode and A,' ' is the argument.

(6) Anything after the argument is assumed to be a comment.

(7) The number of blanks (as long as there is at least one) separating the label, opcode, argument, and comment have no effect on the format of the printout. This format may be changed by changing the tab stops (see Initialization Data).

The assembler stops when it reaches an end-of-file marker or an END pseudo-op. (Any byte that is numerically less than a carriage return will be treated as an end-of-file marker.)

Assembly error messages:

A	Argument error. This can be caused by an undefined symbol (i.e., a name not in the symbol table, such as MADD in example 3) or by bad syntax. Arguments are generally not computed during pass 1, so this error message will be printed only during pass 2. Exception: arguments of pseudo-ops are computed during pass 1, and may cause this error message.
M	Missing label. This occurs only if you use an EQU pseudo-op without a label. This error message, along with the offending line, will be printed during pass 1 and pass 2.
D	Doubly-defined label. The label is already in the symbol table, and you are attempting to change its value. The old value is retained. This can happen if you have just assembled

a program, and you are trying to re-assemble it without first zeroing the symbol table. Pass 1 and 2.

L Label error--bad character in label. This can happen only if the first character in the line is neither alphabetic, nor blank, nor *. (In particular, it can happen if the line begins with a number.) The assembler gives up on the offending line, and 3 NOPS (zero-bytes) are generated in place of whatever machine instruction was intended. Pass 1 and 2.

0 Opcode error. The opcode is not any recognizable operation or pseudo-operation. 3 NOPS (zero-bytes) are generated. Pass 1 and 2.

The following summary of the ATE assembler contains some information not covered in this introduction.

ATE Assembler summary

ATE contains an assembler based on the Processor Technology assembly language format. However, lines must not begin with a line number (although one will be supplied on the output listing). Old line numbers can be removed with a simple edit macro. Each line must begin with a label, if it has one, or else with a blank. Labels can be any length; the assembler will recognize all characters. But to keep the listing neat, labels should be $\leqslant$ 6 characters. The label, opcode, argument, and comment must be separated by blanks, and the argument cannot contain blanks except within literals.

All instruction opcodes are standard. The pseudo-ops are:

ORG Sets the assembly program counter (&) to the given value, and increments the code storage pointer ($) by a like amount. $\&_{new} - \&_{old} = \$_{new} - \$_{old}$. The ORG statement may be labeled, in which case the lable will have the new & as its value.

AORG	"Address Origin." Sets & to the given value without changing $. If this statement is labeled, the label receives the new & value.
SORG	"Storage Origin." Sets $ to the given value without changing &. A label receives the current value of &, not $.
DB	"Define Byte." Standard, except that multiple bytes may be defined, separated by commas.
DW	"Define Word." Standard, except that multiple words may be defined, separated by commas.
DS	"Define storage." Standard.
ASC	"ASCII." Not standard. The ASCII string must be delimited by blanks (but may end with a carriage return). To embed blanks in the ASCII string: `LABEL  ASC-  HELLO-WORLD-    COMMENT` Any non-alphanumeric character may be used in place of the -. Finally, the character ↑ has a special significance within the ASCII string. It sets bit 7 of the preceding character. This is useful in constructing tables. See the section on The User Command Table for an example.
EQU	"Equals." Standard. May occur at most once for each label.
END	Standard.

Assembly errors: A--Argument error. Zero is used in place of the bad argument. Pass 2 only.

M--Missing lable. Pass 1 and 2.

D--Doubly-defined label. The old value of the label is retained. Pass 1 and 2.

L--Label error, bad character. 3 NOPS (zeros) are generated. Pass 1 and 2.

O--Opcode error. 3 NOPS are generated. Pass 1 and 2.

Assembly Commands

&	Set the assembly program counter (which can be referenced by an & character in opcode arguments) to the given address. For example, & 1000H. This command is superceded by an AORG or ORG statement in the source code.

$ Set the code storage pointer (which can be referenced by a $ character in opcode arguments) to the given address. For example, $ 0D000H. This command is superceded by a SORG statement in the source code.

A "Assemble the symbol table." This performs pass 1 over the current file. The two passes of the assembler can be commanded separately in ATE. This allows you to treat many different files as one program. You can have a library of subroutines in source code on tape, for example, and incorporate selected ones into a new program by doing pass 1 over the desired files and then going back and doing pass 2 over the same files. The total amount of source code can be larger than memory, and there is no need to physically cocatenate all the files before assembling them. Of course, both passes can be commanded together by typing A,G.

Note that A does not take an argument. The assembly program counter (&) and the code storage pointer ($) can be set before the first pass over the first file either by the & and $ commands above (e.g., &1000H,$ sets them both to 1000H), or by AORG, SORG, or ORG statements in the source code. $ does not need to be set for pass 1 unless it is referenced in the program.

If an error is detected, an error code (M,D,L, or O for pass 1) is printed, followed by the offending line. Otherwise, pass 1 produces no listing.

G "Generate object code." This performs pass 2 over the current file, storing object code in memory and producing a full listing. If & and $ were set previously, they do not need to be reset for pass 2--ATE does this automatically. The object code listing is produced in the current operating base (see the B command).

Example: (This uses the tape commands Identify and Load, which will be covered later.) Suppose we have 11 consecutive source files on tape, which together would be too large to fit in memory. But we do have room to fit them in one at a time, and in addition we have room to store the 4K of object code they will produce.

(We could also put the code out onto tape--an example will be given later.) We can assemble these files as one program by typing the following command lines:

```
>&0, $0D000H, R11, I, L, A, K..      Then we rewind the tape and type
>R11, I, L, G, K..                   We kill each file after we are
                                        through with it to make room
                                        for the next one.
```

H "Hold the presses." This is the same as G except that it suppresses the listing of everything except the error lines. Note that even without a listing we can look at and edit the object code. Suppose that we want to look at the code for a routine called INIT, which ends just before a line labeled READ. We can type # INIT..READ-1, since these symbols are now in the table. If we had assembled our code at one address and stored it at another, we could type

```
F=$-&, # INIT+F..READ-1+F
```

Z "Zero the symbol table." Initializes a new symbol table containing only the 8080 register symbols and their values. (A=7,B=0,C=1,D=2,E=3,H=4,L=5,M=6,SP=6,PSW=6). After initialization, these symbols have no special status; they can be removed (using Zsymbol) or redefined (using =) just like any other symbol. Note: if the table was Moved (as described under the Move command) then Z will initialize the new table at the new address.

Zsymbol Zero the given symbol. This removes the symbol from the table and compacts the table. For instance, Z INIT removes all traces of INIT (and its value) from the table and compacts the table, freeing 6 bytes of table space.

Z>symbol Zero after the given symbol. Removes all symbols from the table that were created chronologically after the given symbol. Before assembling a program, you can use this to remove conflicting symbols from the table (from a former assembly of the same program, say) without destroying previously created variables that you want to save. There is usually no need to completely Zero the table. For instance, suppose you have saved some names for your often-used machine language routines (instead of putting these in the user command table). If the last such name to be saved was DOS, they typing Z>DOS before an assembly will preserve these names while giving you an otherwise clean slate.

Tape Handling Commands

These commands are fairly simple--they were designed with the realities of audio cassette recording in mind. But in combination with ATE's multi-command line and programming capability, they are quite powerful. See the examples at the end of this section.

I "Identify." Identifies the next record on the tape (i.e., reads the record header) and prints information at the terminal. For example, if after loading ATE from cassette, you rewind the cassette to the lead-in tone:

```
>I
1000 1FFF ATE OBJECT CODE COPYRIGHT 4/15/77 G.FITTS
```

This gives the addresses to which the record will load (unless you specify otherwise), and the record title. The tape is now stopped between the header and the record body, waiting for an L, J, or V command.

Note: Every record on the tape consists of (a) a 5 second lead-in tone, (b) the record header--256 bytes, approx. 9 seconds, (c) another 5 second tone, and (d) the record body. When a tape is first mounted, or after it is rewound, you must position it manually to the first lead-in tone. After this, ATE will automatically start and stop the tape at the correct positions with no further need for manual intervention.

I(TITLE) Searches the tape (forward) for a record whose title begins with the given string. The entire title need not be given. For example, I(ATE) would find the record mentioned above, as would I(ATE OBJECT), etc. Header info from other records encountered during the search is printed, so I(any non-existent title) will catalog the tape. (The tape will run for about 1 minute beyond the end of recorded material before ATE will stop it, issue an error sign ?, and return control to the terminal.)

Note: Control-A does not function while the tape is running. But stopping the computer and restarting ATE at address SYS1 will stop the tape. The tape must then be repositioned to a header lead-in tone.

L "Load." If used without an argument (there is no default argument in this case), the record is loaded at the address that was printed in response to the I command. If it is a source file, then this address is the top of the source area. In this case, the file is loaded, the source area is expanded to include the new file, and the new file becomes current with ↑ at its end. (That is, unless a checksum error occurs. See below.)

If an address is given with the L command, then the file (source or not) is simply loaded to that address. Even if it was a source file, it is not made current or incorporated into the source area.

After loading, a checksum is computed across the loaded record. An error will cause a ?, and control will return to the terminal. If the bad record was a source file, it will not be incorporated into the source area or made current.

Of course, multiple tape commands can be included in the command line or in a program, as for any ATE commands. I,L will identify and load the next record. I(BASIC),L,X<R> will find, load, and execute that record (as long as its entry point is the first byte). Note than an I command must precede an L, although other non-tape commands can intervene.

J "Jump over." Moves the tape past the previously Identified record and stops it.

V "Verify." Checks the record byte-for-byte against memory, issuing a ? at the end if there is any difference. To use this command, first Save the record (see below), rewind the tape to the lead-in tone for the record body, and type V. (Or you can rewind to the header lead-in tone and type I,V)

S "Save." Takes an argument, and saves the addressed interval on tape. For instance, S.. saves the current file. S<S> saves all files. S1000H..1FFFH creates a new copy of ATE.

(A) If no title was given (see the T command below), then a default title is used. For source files, this is the first line of the file. For object code, this is the first 8 bytes. (ATE labels a record "source" if it begins in the source area.)

(B) If no write-address was given (see the W command below), then by default the write-address that is saved with the record is the same as the address from which the record is saved. (This address can always be changed at load time. It is irrelevant for source files, which are always loaded onto the source area.)

(C) Records saved with the S command (including ATE itself) can be loaded and executed by the ROM bootstrap loader on the Morrow interface board. Simply set the tape to the lead-in tone and execute the bootstrap (address 815FH=201:137Q). The sense switches play the following role:

If all switches are off, the program loads and executes (as long as there is no checksum error). If switch 0 is on, the tape will stop after the record header has been loaded. You can then change the load address from the front panel. It is stored at address 8277H=202:167Q, low byte first. Then turn switch 0 off and restart the computer from where you stopped it.

After the load is complete, a checksum is computed, and if in error, the computer enters a jump-self loop (C2 0C 82, or 302 014 202). Otherwise, switch 7 is checked. If off, the record is executed. If on, the computer loops, reading switch 7. At this point, you can go into the record from the front panel and change its IO, or whatever.

And one more feature: after changing the I/O or whatever, you can create an updated tape of the same record by setting your recorder to record and executing address 823EH=202:076Q. (Of course it would probably be easier to load ATE and use it to edit and save the new version of the record.)

(D) George Morrow's Speakeasy board can control up to three recorders. ATE always reads from machine #1, and at first it also writes to machine #1. But editing a tape is ten times easier with two recorders. To make ATE write to recorder #2, type ↑QEF5H, E#85#. For further details, see the sections entitled "Initialization" and "Initialization Values".

T "Title." If used before the Save command (with no intervening tape commands), this titles the record about to be saved with the given text. The given text must be enclosed in parantheses. For example,

```
T(ATE OBJECT CODE COPYRIGHT 4/15/77 G. FITTS), S 1000H..1FFFH
```

will create the record mentioned under the I command.

Note that there is really no need to title source code, since the default title (the first line) is the conventional title for the file once it is loaded into memory.

W "Write address." If used before the Save command (with no intervening tape commands), this sets the write address (load address) of the record about to be saved to the given value. For instance, if you copied ATE to 0D000H (say), and then installed some of your own custom patches, you could save the new version by typing:

```
T(PERSONALIZED ATE), W 1000H, S 0D000H..0DFFFH
```

RS "Resave." If used after a load (with no intervening tape commands), this resaves the the record using the original title and load address. For instance, if you had just loaded ATE at 0D000H, then typing RS would create a copy from this address that still had its original title and load address 1000H

But if you typed S 0D000H..0DFFFH, then the load address of the new copy would be 0D000H, and the title would be the first 8 bytes of code.

Tape examples

Here is a command line to search the tape for a record containing STRING.

```
 R999, I, L, QS<R>/(STRING), K<R>
```

Here is a program (called *EDTAPE) to read through a tape, changing every occurrence of STRING1 to STRING2 and creating an updated tape. Note that ATE must write to recorder #2 (as described under Save) to make this feasible.

```
*LOOP, QF(STRING1), K, E(STRING2), >(*LOOP)
*EDTAPE, I, L, D(*LOOP),  RS,  K<R>, >(*EDTAPE)
```

If you want to assemble 10 files into a single program and you don't have enough memory to store the assembled object code, you could type the following lines:

```
 &0, R10, $0D000H, I, L, A, K..      (Then rewind the tape)
 &0, R10, $0D000H, I, L, W&, G, S 0D000H..$-1, K..
```

Gory details

Hardward Requirements

To run ATE with no modifications, you need at least 8K of memory beginning at address 0 and Morrow's IO board connected to recorder #1 and to a teletype (or other 110 baud serial device). More memory is desireable, as is a second recorder.

If you meet all the above requirements except for the baud rate, then you can patch in the new rate by changing one byte. See below.

If your terminal is not connected through Morrow's IO board, you can patch your own IO routines into ATE by changing three jump instructions. The procedure is described below.

If your tape recorder is not connected through Morrow's board, then (assuming you can load ATE--see below) you can still use the editor and assembler parts of ATE, but you won't be able to use the tape commands, unless you patch in a new tape driver (see below).

Loading ATE

Assuming that you have the standard hardware described above, proceed as follows: mount the ATE cassette in recorder #I and position the tape to the beginning of the first lead-in tone (about 25 seconds into the tape). Then execute address 815FH = 201:137Q with all sense switches off. The ROM bootstrap loader on the Morrow IO board will read in a loader from the tape, which will then load the ATE object code to addresses 1000H..1FFFH. This takes about 2-3/4 minutes. Then, unless a checksum error is detected, ATE will begin executing, printing a prompt > character at the terminal. (If a checksum error is detected, the loader enters a jump-self loop: C2 0C 82 or 302 014 202).

If you can load ATE as above, but you want to patch the baud rate or change the IO before ATE starts executing, then turn sense switch 7 on before the load is complete. This will prevent the loader from passing control to ATE. When the tape stops, you can make the patches as described below and then execute ATE from address 1000H.

If you want to load ATE through some other cassette interface: ATE is recorded at 300 baud Kansas City Standard. The tape consists of (a) a 5 second lead-in tone, (b) a 256 byte header (approx. 9 seconds), (c) a I/2 second gap, (d) another 5 second lead-in tone, and (d) 4096 bytes of core image ATE object code, which should be loaded at address 1000H. The 2 byte checksum for these 4096 bytes is stored in the header. It is the 43rd and 44th bytes of the header, low byte first.

Initialization

ATE is written so that it can be stored in ROM. Consequently, you can write-protect the 2nd 4K block of memory after loading ATE, if you want. ATE keeps its variables, stack, etc., in RAM beginning at address 0E60H = 016:140Q. When ATE is executed from address 1000H, initial values for many of these variables are copied into RAM. These initial values are stored together in a list within ATE.

To make a permanent change in any of these values, you can make changes within this list. (See the addresses appendix for the details of this list.) To do this, you can load ATE with sense switch 7 on, make changes from the front panel, and then execute from address 1000H. Or you can let ATE begin executing, make the changes using the Enter command, and then type X1000H. In either case, you will want to make an updated copy of ATE by typing T(PERSONALIZED ATE), S 1000H..1FFFH

If you don't want your changes to be permanent, you can alter the desired data at its new location in RAM after initialization. (Again, see the addresses appendix for these locations.)

IO Patching

If you are using the serial port on the MMS IO board and you simply want to patch in a new speed constant, enter it at one of the addresses given in the appendix (either the pre or post-initialization address, **IBAUD** or **SCON**, depending on whether you want a permanent or temporary change).

Three routines are required for terminal IO: (a) a character input-echo routine, (b) a character output routine (which can be the echo part of the first routine), and (c) a panic detect routine. ATE accesses each routine through a single jump instruction, and a new routine can be patched in simply by changing the jump address. See the addresses appendix for the locations of these jumps.

The requirements for these routines are as follows: In every case, the only CPU register that must be maintained is SP. You can use up to 20 bytes of stack. (a) The character input-echo routine should get a character from the terminal, strip off the parity bit if necessary, echo the character (possibly by falling into the character output routine), and return with the character in register A. (b) The character output routine should output the character in register A. (c) The panic detect routine should RETurn to continue the current process, or jump to SYS1 to abort it (see addresses appendix).

Suggestions:

The character output routine can drive a different device than the echo routine. For instance, commands could be echoed to your CRT, while printouts (which come from the character output routine) could appear on your hardcopy device.

Of course, ATE can change IO devices under program control by entering a jump to the new driver at OTPAD (see appendix). For instance, suppose that your hardcopy driver is located at 0D000H, while your CRT driver is at 0E000H. You could create the following edit macros:

```
*HARDCOPY, X=↑, ↑ OTPAD, E#C3 00 D0#, ↑X
*SOFTCOPY, X=↑, ↑ OTPAD, E#C3 00 E0#, ↑X
```

Now, D(*HARDCOPY) can be used in a command line or in a program to route output to the hardcopy device, while D(*SOFTCOPY) will return output to the CRT. (In each case, the entry pointer is saved and restored.)

ATE can be used with a half-duplex terminal by eliminating the echo part of the character input routine.

```
                         *SAMPLE IO ROUTINES FOR THE 3P+S
        000:000          STATUS EQU  0        STATUS PORT
        000:001          DATA   EQU  1        DATA PORT
        000:100          DATAREADY EQU 40H
        000:200          PRINTERREADY EQU 80H
        000:001          ABORT  EQU  1        CONTROL-A
        020:003          SYS1   EQU  1003H    ATE RE ENTRY POINT
                         *
000:000 333 000          INECHO IN   STATUS
000:002 346 100                 ANI  DATAREADY
000:004 312 000 000             JZ   INECHO
000:007 333 001                 IN   DATA
000:011 346 177                 ANI  7FH
000:013 107              OUTCHR MOV  B,A
000:014 333 000          OUTLOP IN   STATUS
000:016 346 200                 ANI  PRINTERREADY
000:020 312 014 000             JZ   OUTLOP
000:023 170                     MOV  A,B
000:024 323 001                 OUT  DATA
000:026 311                     RET
                         *
000:027 333 000          PANDET IN   STATUS
000:031 346 100                 ANI  DATAREADY
000:033 310                     RZ
000:034 333 001                 IN   DATA
000:036 315 000 000             CALL INECHO
000:041 376 001                 CPI  ABORT
000:043 300                     RNZ
000:044 303 003 020             JMP  SYS1
>
```

```
                       1    *SAMPLE IO ROUTINES FOR THE 3P+S
        0000           2    STATUS EQU  0       STATUS PORT
        0001           3    DATA   EQU  1       DATA PORT
        0040           4    DATAREADY EQU 40H
        0080           5    PRINTERREADY EQU 80H
        0001           6    ABORT  EQU  1       CONTROL-A
        1003           7    SYS1   EQU  1003H   ATE RE ENTRY POINT
                       8    *
0000    DB 00          9    INECHO IN   STATUS
0002    E6 40          10          ANI  DATAREADY
0004    CA 00 00       11          JZ   INECHO
0007    DB 01          12          IN   DATA
0009    E6 7F          13          ANI  7FH
000B    47             14   OUTCHR MOV  B,A
000C    DB 00          15   OUTLOP IN   STATUS
000E    E6 80          16          ANI  PRINTERREADY
0010    CA 0C 00       17          JZ   OUTLOP
0013    78             18          MOV  A,B
0014    D3 01          19          OUT  DATA
0016    C9             20          RET
                       21   *
0017    DB 00          22   PANDET IN   STATUS
0019    E6 40          23          ANI  DATAREADY
001B    C8             24          RZ
001C    DB 01          25          IN   DATA
001E    CD 00 00       26          CALL INECHO
0021    FE 01          27          CPI  ABORT
0023    C0             28          RNZ
0024    C3 03 10       29          JMP  SYS1
>
```

The Tape Driver

If you don't have George Morrow's Interface board, you will have to duplicate some of its onboard ROM software with your own tape driver, and you will have to provide 512 bytes of memory at 8200H = 202:000Q. (This data buffer cannot be relocated without reassembling ATE, since ATE contains code that executes within this address space.) ATE accesses its tape driver thru a single call instruction at TAPCAL (see appendix and (5) below), and uses these conventions:

(1) If bit 0 of the A register is 1, then a <u>write</u> operation is required:
  (a) The HL register pair contains the beginning address of the data to be written.
  (b) The DE register pair contains the number of bytes to be written.

(2) If bit 0 of the A register is 0, then a <u>read</u> operation is required:
  (a) The HL register pair contains the beginning address of the buffer where the data should be stored.
  (b) The DE register pair contains the number of bytes to be read and stored.
  (c) If the C register equals 0, then the data should simply be read and stored. However, if the C register equals 1, the data should be read but <u>not</u> stored (i.e., the tape should be advanced over DE bytes). If the C register equals 40H, then the data should be read and <u>compared</u> to memory beginning at address HL. If a discrepancy is found, a non-zero byte should be stored at address DERR (see appendix). (In addition, you could store the address of the offending byte at ERSAV. Then one of the commands "?, #?, or ?? would give information about this address).

(3) If your tape interface is capable of detecting any physical error conditions in your tape hardware, you can signal this to ATE by storing a non-zero byte at SERR and returning.

(This is what allows ATE to signal an error after one minute of listening to a blank or motionless tape.)

(4) If your tape interface has motion control, it should stop the tape after each read or write operation.

(5) You will also have to provide a checksum computing routine. ATE uses the routine CHECK on Morrow's I/O board, calling this routine twice at CHECK1 and CHECK2 (see appendix). If you want, you can duplicate the code for CHECK, which follows. In any case, you will have to use the same conventions.

(6) Finally, if you want the bootstrapping capability and the reproductive capability provided by each ATE record header, you will have to keep your tape driver and checksum computer in ROM, and you will have to provide a bootstrap loader in ROM that can read the 256 byte header into TAPRAM and then branch there.

COMPUTE CHECK-SUM ROUTINE

Calling conventions:

(A) The register pair H-L is loaded with the starting address of the data block on which the check-sum is to be computed.

(B) The register pair D-E is loaded with the word count of the data block.

(C) The computed check is returned in the register pair H-L.

Including the return address of the calling program, the routine uses four levels of the stack.

```
814D  E5          CHECK  PUSH  H        SAVE ADDRESS POINTER
814E  21 00 00           LXI   H,0      INITIALIZE CHECK SUM
8151  44                 MOV   B,H
8152  E3          GDATA  XTHL           SAVE AND EXCHANGE/ADDR POINTER
8153  4E                 MOV   C,M      GET DATA
8154  23                 INX   H        INCREMENT ADDRESS POINTER
8155  E3                 XTHL           SAVE & GET PARTIAL CHECK SUM
8156  09                 DAD   B        ADD NEW DATA
8157  1B                 DCX   D        DECREMENT WORD COUNT
8158  7A                 MOV   A,D      TEST FOR
8159  B3                 ORA   E          WORD COUNT
815A  C2 52 81           JNZ   GDATA      EQUAL ZERO
815D  D1                 POP   D        RESTORE STACK
815E  C9                 RET
```

The user command table

The user command table is initially located at IUSRCT (see appendix), but this may be changed at any time (see below). Command names may be any length, and may contain any printing ASCII characters. The only restrictions are:

(a) The last byte of each command name in the table must have its high order bit set to 1. (Since the ASCII code only requires the low order 7 bits, this does not restrict your choice of characters.)

(b) The command name must be followed by the command address, low byte first.

(c) The table must end with a zero byte.

You can create a user command table with the Enter command, but the easiest way is to assemble it in place, as in the example on the next page. Once you have created a table, you can save it on tape (along with the object code for its routines), and reload it at any time. See the Save command.

When ATE is initialized, it writes a zero (= end-of-table byte) at IUSRCT, and writes the address IUSRCT into RAM at USRCT. Thereafter, whenever ATE is given a command, it begins searching at USRCT first before searching its own internal command tables. Thus user commands can supercede ATE's. For instance, if you create a command called PUNCH, ATE will not interpret this as Print UNCH.

The user command table can be relocated in several ways. You can always change IUSRCT before initialization, or change USRCT after initialization. If you already have a table occupying addresses 0E00H..0E0CH, for instance, and you want to move it to 3000H, simply type ↑3000H, M 0E00H..0E0CH . ATE will realize that you have moved the user command table, and will remember the new location.

```
>
>
>
>
>
>
>
>
>
>
>
>
>N,E( AORG 0E00H
 SORG 0E00H START OF USER COMMAND TABLE
 ASC PNCH^
 DW 0D000H ADDRESS OF PNCH
 ASC PAPR^
 DW 0E000H ADDRESS OF PAPR
 DB 0 END OF TABLE)
>
>G
016:000                 1           AORG 0E00H
016:000 016:000         2           SORG 0E00H    START OF USER COMMAND
                                                   TABLE
016:000 120 116 103 310 3           ASC  PNCH^
016:004 000 320         4           DW   0D000H   ADDRESS OF PNCH
016:006 120 101 120 322 5           ASC  PAPR^
016:012 000 340         6           DW   0E000H   ADDRESS OF PAPR
016:014 000             7           DB   0        END OF TABLE
>
>
>B16,G
0E00                    1           AORG 0E00H
0E00    0E00            2           SORG 0E00H    START OF USER COMMAND
                                                   TABLE
0E00    50 4E 43 C8     3           ASC  PNCH^
0E04    00 D0           4           DW   0D000H   ADDRESS OF PNCH
0E06    50 41 50 D2     5           ASC  PAPR^
0E0A    00 E0           6           DW   0E000H   ADDRESS OF PAPR
0E0C    00              7           DB   0        END OF TABLE
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
```

ATE addresses -- functional descriptions (numerical values follow)

IBOSA	Pointer to initial beginning of source file area
ICODE	Initial value of & and $
IBAUD	Initial speed constant for Morrow's interface board
ISYMTB	Pointer to initial beginning of symbol table
IBASE	Initial base for numerical input and output
IWCHNL	Initial write channel -- ie, reg A constant for WRITE calls to COPE (the ROM tape driver on Morrow's IO board). 83H = 203Q for recorder #1, 85H = 205Q for recorder #2, or 89H = 211Q for recorder #3.
IUSRCT	Pointer to the beginning of the initial user command table. (ATE writes a zero there during initialization.)
IINPAD	Contains a jump to the initial character input-echo routine.
IOTPAD	Contains a jump to the initial character output routine.
IPNPAD	Contains a jump to the initial panic-detect routine.
IWIDTH	Initial terminal width
ITAB1	Initial TAB1; column number for labels
ITAB2	Initial TAB2; column number for opcodes
ITAB3	Initial TAB3; column number for arguments
ITAB4	Initial TAB4; column number for comments
IALOFF	Initial assembly source-listing offset; column number for error flag (if any). The source listing follows to the right of this column, with TABS 1-4 interpreted relative to this column.

BASE	Current base for numerical input and output.
WCHNL	Current write channel (see IWCHNL above)
USRCT	Pointer to the beginning of the current user command table
INPAD	Contains a jump to the current input-echo routine.
OTPAD	Contains a jump to the current character output routine.
PNPAD	Contains a jump to the current panic-detect routine.

WIDTH	Current terminal width
TAB1	Current tab 1; column number for labels
TAB2	Current tab 2; column number for opcodes
TAB3	Current tab 3; column number for arguments
TAB4	Current tab 4; column number for comments
ALOFF	Current assembly source-listing offset. The source code is listed to the right of the object code, with tabs 1-4 interpreted relative to this offset.

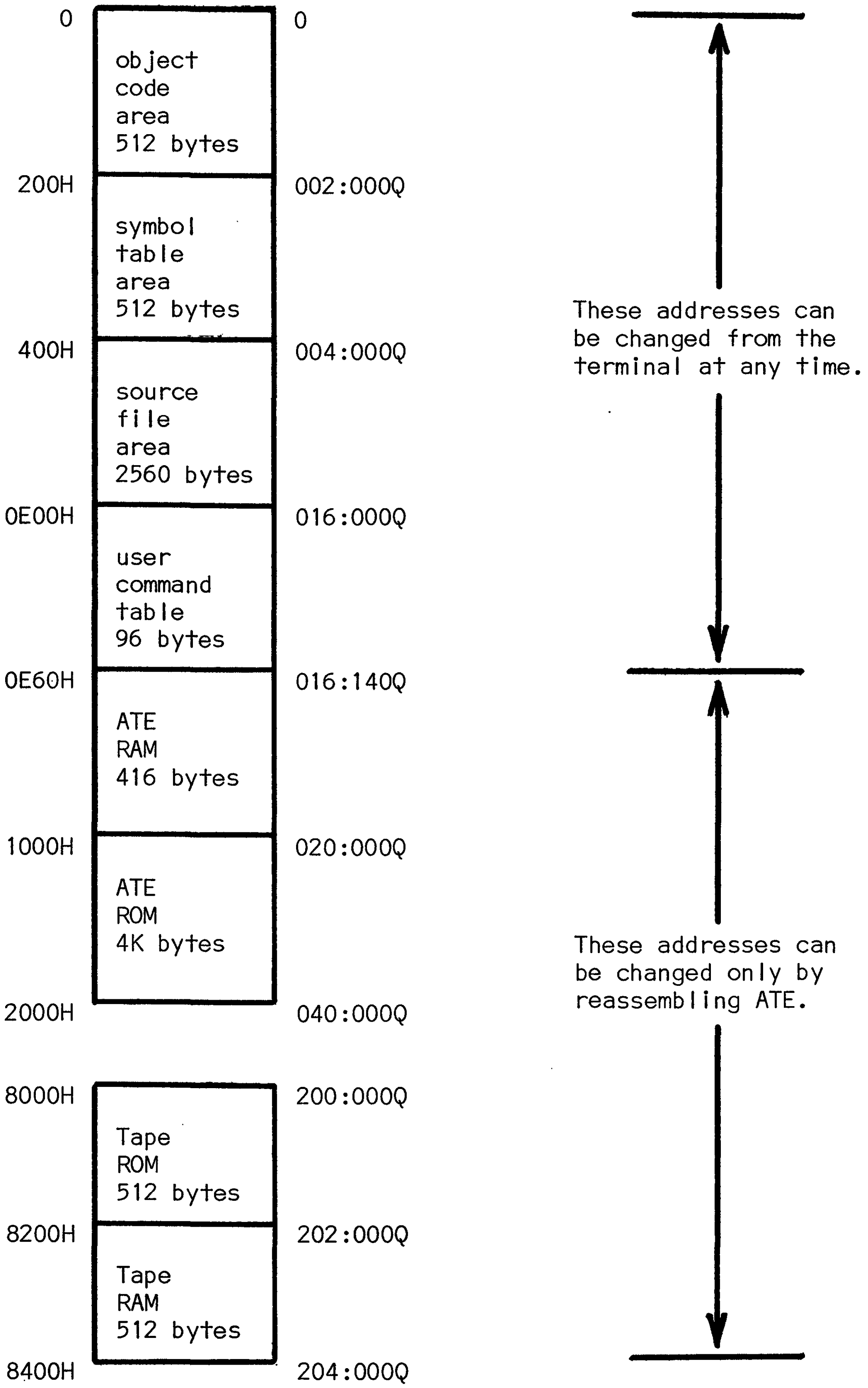
ATERAM	The beginning address for storage of ATE's variables and stack
BOSAP	Pointer to the beginning of the current source file area
EOSAP	Pointer to the end of the current source file area
BOFP	Pointer to the beginning of the current file
EOFP	Pointer to the end of the current file
ASPC	Assembly program counter (&)
STCTR	Assembly storage pointer ($)
SYMTB	Pointer to the beginning of the current symbol table
TABA	Pointer to the end of the current symbol table
CHPTR	The entry pointer (↑)
PNTR	The command interpretation pointer
P1	The beginning value of the argument (<)
P2	The ending value of the argument (>)
RECAD	Pointer to the beginning of the record read in from tape
RECND	Pointer to the end ot the record read in from tape
ERSAV	Pointer to the character that caused a command interpretation error (?)
PHD	The column in which the print head is waiting

SYS0	This is the beginning of ATE, ie, the entry point that initializes everything. Jump here after power up. Typing X followed by this address will re-initialize ATE.
SYS1	This is the re-entry point to ATE that avoids re-initialization.

VCHK	This is useful with user-written machine language routines called from ATE (via the user command table, or via an X command). VCHK will return with the Z flag *off* if there *is* an argument following the command, or with the Z flag on if there is no argument. (See the X command for more info.)
CVALS	This routine returns the values of any ATE argument that follows a user command. The beginning value is returned in HL, while the ending value is returned in DE. If there is no argument, the values computed for the last command are returned. Any error encountered will cause a ?-output and will return control to the terminal. If the user has supplied several arguments (separated by blanks), these can be detected by VCHK and evaluated by repeated calls to CVALS. CVALS will not proceed beyond a comma, carriage return, or end-of-file zero byte. The reference string for any matching operands is the current file.
VALUS	This routine is the same as CVALS with two exceptions: you must provide the beginning and ending addresses of any reference string in HL and DE; and in case of an error, VALUS simply returns with the Z flag off.
LISTR MARGN	A terminal width of less than approximately 50 (depending on your tab settings) will not accomodate an assembly listing properly. To remedy this, replace the CALL TAB at LISTR with a CALL MARGN.
CHECK1 CHECK2	If you don't have Morrow's IO board and you are supplying your own tape driver, you will also have to supply a checksum computing routine as described earlier in the TAPE DRIVER section. Replace the CALL CHECK at CHECK1 and CHECK2 with a call to your own checksum routine.

TAPCAL	This is ATE's only call to its tape driver, so you can patch in your own driver by changing this call. See also CHECK1 and CHECK2 above and the "tape driver" section of the manual.
TAPRAM	Each I command, and each bootstrap load, reads a 256 byte record header into this location.
SCON	Speed constant for the serial interface on Morrow's IO board
DERR	DATA ERROR and STATUS ERROR. Before each tape driver call, ATE sets these bytes to zero. On return from the tape driver, ATE checks both bytes, and if either one is non-zero, ATE signals an error and returns control to the terminal. See the "tape driver" section for more info. Note that ATE's checksum logic is independent of these bytes.
SERR	
CHKSM	The record checksum
LODAD	The record load-address
LNGTH	The record length
TYPE	The record type: an ASCII 'S' for source, 'B' for binary
WUNIT	The recorder (see WCHNL) that will be used if the reproductive capability of the record header is invoked. See the Save command for more info.
TITYP	The type of the record title. See TYPE above.
TITLE	The title that was given at Save time, if any, or else the first 128 bytes of the record.

<u>ATE memory map</u>: standard initialization, minimal 8K system



These addresses can be changed from the terminal at any time.

These addresses can be changed only by reassembling ATE.

```
>
                                *ATE ADDRESSES -- NUMERICAL VALUES
                                *
                                *
                                *THE FOLLOWING ARE COPIED INTO RAM
                                *AT INITIALIZATION TIME
                                *
035:224 000 004                 IBOSA  DW   400H
035:226 000 000                 ICODE  DW   0
035:230 254 000                 IBAUD  DW   0ACH
035:232 000 002                 ISYMTB DW   200H
035:234 010                     IBASE  DB   8
035:235 203                     IWCHNL DB   83H
035:236 000 016                 IUSRCT DW   0E00H
035:240 303 345 027             IINPAD JMP  MINPT
035:243 303 263 201             IOTPAD JMP  SROUT
035:246 303 202 027             IPNPAD JMP  PANIC
035:251 110                     IWIDTH DB   72
035:252 010                     ITAB1  DB   8
035:253 017                     ITAB2  DB   15
035:254 024                     ITAB3  DB   20
035:255 035                     ITAB4  DB   29
035:256 025                     IALOFF DB   21
                                *
                                *STARTING WITH IBASE, THE ABOVE
                                *VALUES ARE COPIED INTO THE
                                *FOLLOWING RAM LOCATIONS
                                *
016:364 000:001                 BASE   DS   1
016:365 000:001                 WCHNL  DS   1
016:366 000:002                 USRCT  DS   2
016:370 000:003                 INPAD  DS   3
016:373 000:003                 OTPAD  DS   3
016:376 000:003                 PNPAD  DS   3
017:001 000:001                 WIDTH  DS   1
017:002 000:001                 TAB1   DS   1
017:003 000:001                 TAB2   DS   1
017:004 000:001                 TAB3   DS   1
017:005 000:001                 TAB4   DS   1
017:006 000:001                 ALOFF  DS   1
                                *
                                *OTHER LOCATIONS IN ATE RAM
                                *
016:140 001:240                 ATERAM DS   200+BUFLN+BUFLN+BUFLN
016:322 000:002                 BOSAP  DS   2
016:326 000:002                 EOSAP  DS   2
016:332 000:002                 EOFP   DS   2
016:336 000:002                 EOFP   DS   2
016:320 000:002                 ASPC   DS   2
016:324 000:002                 STCTR  DS   2
016:362 000:002                 SYMTB  DS   2
016:356 000:002                 TABA   DS   2
016:342 000:002                 CHPTR  DS   2
016:306 000:002                 PNTR   DS   2
016:312 000:002                 P1     DS   2
016:316 000:002                 P2     DS   2
016:346 000:002                 RECAD  DS   2
016:352 000:002                 RECND  DS   2
```

```
016:354 000:002          58   ERSAV  DS   2
017:032 000:002          59   PHD    DS   2
                         60   *
                         61   *ADDRESSES INSIDE ATE
                         62   *
020:000 303 137 035      63   SYS0   JMP  INIT
020:003 061 304 016      64   SYS1   LXI  SP,STACK
021:014 315 005 030      65   VCHK   CALL SBLK
021:025 315 067 021      66   CVALS  CALL FVALS
021:103 257              67   VALUS  XRA  A
033:075 315 222 031      68   LISTR  CALL TAB
027:260 257              69   MARGN  XRA  A
034:061 315 115 201      70   CHECK1 CALL CHECK
034:122 315 115 201      71   CHECK2 CALL CHECK
034:164 315 012 201      72   TAPCAL CALL COPE
                         73   *
                         74   *ADDRESSES IN THE TAPE RAM
                         75   *
202:000 002:000          76   TAPRAM DS   512
203:363 000:002          77   SCON   DS   2
203:365 000:001          78   DERR   DS   1
203:366 000:001          79   SERR   DS   1
                         80   *
                         81   *THE FOLLOWING ARE LOADED
                         82   *WITH EACH RECORD HEADER
                         83   *
202:052 000:002          84   CHKSM  DS   2
202:167 000:002          85   LODAD  DS   2
202:171 000:002          86   LNGTH  DS   2
202:173 000:001          87   TYPE   DS   1
202:174 000:001          88   WUNIT  DS   1
202:175 000:001          89   TITYP  DS   1
202:176 000:202          90   TITLE  DS   TAPRAM+256-&
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
```

```
>
>
                        *ATE ADDRESSES -- NUMERICAL VALUES
                        *
                        *
                        *THE FOLLOWING ARE COPIED INTO RAM
                        *AT INITIALIZATION TIME
                        *
1D94    00 04           IBOSA  DW   400H
1D96    00 00           ICODE  DW   0
1D98    AC 00           IBAUD  DW   0ACH
1D9A    00 02           ISYMTB DW   200H
1D9C    08              IBASE  DB   8
1D9D    83              IWCHNL DB   83H
1D9E    00 0E           IUSRCT DW   0E00H
1DA0    C3 E5 17        IINPAD JMP  MINPT
1DA3    C3 B3 81        IOTPAD JMP  SKOUT
1DA6    C3 82 17        IPNPAD JMP  PANIC
1DA9    48              IWIDTH DB   72
1DAA    08              ITAB1  DB   8
1DAB    0F              ITAB2  DB   15
1DAC    14              ITAB3  DB   20
1DAD    1D              ITAB4  DB   29
1DAE    15              IALOFF DB   21
                        *
                        *STARTING WITH IBASE, THE ABOVE
                        *VALUES ARE COPIED INTO THE
                        *FOLLOWING RAM LOCATIONS
                        *
0EF4    0001            BASE   DS   1
0EF5    0001            WCHNL  DS   1
0EF6    0002            USRCT  DS   2
0EF8    0003            INPAD  DS   3
0EFB    0003            OTPAD  DS   3
0EFE    0003            PNPAD  DS   3
0F01    0001            WIDTH  DS   1
0F02    0001            TAB1   DS   1
0F03    0001            TAB2   DS   1
0F04    0001            TAB3   DS   1
0F05    0001            TAB4   DS   1
0F06    0001            ALOFF  DS   1
                        *
                        *OTHER LOCATIONS IN ATE RAM
                        *
0E60    01A0            ATERAM DS   200+BUFLN+BUFLN+BUFLN
0ED2    0002            BOSAP  DS   2
0ED6    0002            EOSAP  DS   2
0EDA    0002            BOFP   DS   2
0EDE    0002            EOFP   DS   2
0ED0    0002            ASPC   DS   2
0ED4    0002            STCTR  DS   2
0EF2    0002            SYMTB  DS   2
0EEE    0002            TABA   DS   2
0EE2    0002            CHPTR  DS   2
0EC6    0002            PNTR   DS   2
0ECA    0002            P1     DS   2
0ECE    0002            P2     DS   2
0EE6    0002            RECAD  DS   2
```

```
0EEA     0002           RECND  DS   2
0EEC     0002           ERSAV  DS   2
0F1A     0002           PHD    DS   2
                        *
                        *ADDRESSES INSIDE ATE
                        *
1000     C3 5F 1D       SYS0   JMP  INIT
1003     31 C4 0E       SYS1   LXI  SP,STACK
110C     CD 05 18       VCHK   CALL SBLK
1115     CD 37 11       CVALS  CALL FVALS
1143     AF             VALUS  XRA  A
1B3D     CD 92 19       LISTR  CALL TAB
17B0     AF             MARGN  XRA  A
1C31     CD 4D 81       CHECK1 CALL CHECK
1C52     CD 4D 81       CHECK2 CALL CHECK
1C74     CD 0A 81       TAPCAL CALL COPE
                        *
                        *ADDRESSES IN THE TAPE RAM
                        *
8200     0200           TAPRAM DS   512
83F3     0002           SCON   DS   2
83F5     0001           DERR   DS   1
83F6     0001           SERR   DS   1
                        *
                        *THE FOLLOWING ARE LOADED
                        *WITH EACH RECORD HEADER
                        *
822A     0002           CHKSM  DS   2
8277     0002           LODAD  DS   2
8279     0002           LNGTH  DS   2
827B     0001           TYPE   DS   1
827C     0001           WUNIT  DS   1
827D     0001           TITYP  DS   1
827E     0082           TITLE  DS   TAPRAM+256-&
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
```

Bugs

#FF#@ doesn't work properly as a command argument. Using the any-character matching operand @ cocatenated with numerically expressed bytes will conflict with an FF byte, if there is one within the # signs. (It will cause the FF byte to match anything.) Rule: don't use FF and @ together.

OPCODE COMMENT does not print properly when the opcode requires no argument. The print routine does not know which opcodes require arguments and which don't, so in this case the comment will be printed in the argument field. Rule: if you want to comment a line where the opcode doesn't require an argument, use some visually inoffensive character (such as a period) as the 'argument'. This won't affect the assembly, and the line will list correctly.

ASC TOO-MANY-CHARACTERS will assemble correctly but will not list correctly. Instead of keeping the object code in its proper columns, the listing will allow the object code to run over into the source code columns, displacing the source listing of that line to the right. Rule: to keep a hex assembly listing neat, use 5 or fewer ASC characters per line. For an octal listing, use 4 or fewer. Or, more characters can be accomodated per line by increasing ALOFF (see appendix).

DB too many bytes, and DW too many words: same comments as for ASC above.

		Takes an argument?	Reference string	See page
Basic editing				
↑	Set the pointer	yes	current file	19
E	Enter	no		20
K	Kill	yes	current file	22
M	Move	yes	current file	22
C	Copy	yes	current file	23
Printing				
"	Quote	yes	current file	24
'	Quote one line	no		24
P	Print	yes	current file	24
B	Base	yes	(current file)	25
#	Quote numbers	yes	current file	25
?	Where	yes	current file	26
Memory files				
F	File	yes	source area	27
N	New	no		27
O	Originate	yes	current file	28
Programming				
D	Do	yes	source area	28
>	Goto	yes	source area	29
R	Repeat	yes	(current file)	30
*	Label	no		30
QF	Quit on failure	yes	current file	31
QS	Quit on success	yes	current file	32
=	Equals	yes	current file	32
X	Execute	yes	(current file)	32
Assembling				
&	Set &	yes	current file	47
$	Set $	yes	current file	48
A	Assemble the table	no		48
G	Generate object code	no		48
H	Hold the presses	no		49
Z	Zero the table	no		49
Zlabel	Zero the label	no		49
Z label	Zero after	no		49
Tape handling				
I	Identify	no		50
I(Title)		no		50
L	Load	optional	current file	51
J	Jump over	no		51
V	Verify	no		52
S	Save	yes	current file	52
T	Title	no		53
W	Write address	yes	current file	53
RS	Resave	no		53