# Disk/ATE™ User's Manual

Manual Rev. 1.1

By Jim Rosenberg



# INTRODUCTION

DISKATE is a powerful general purpose program development system for the 8Ø8Ø which can be used as a text editor, a monitor, an assembler, and even an operating system. The text editor and monitor allow powerful content oriented commands. The entire system is programmable through variables, repetitive loops and command macros. The assembler provides great flexibility by allowing its two passes to be invoked separately. The user has full control over all memory allocation, including placement of the symbol table. Source programs, object programs and symbol tables can all be stored on and retrieved from the disk. The disk commands automatically allocate and deallocate disk space, handling compacting of the disk whenver necessary. Although configured for the Disk Jockey disk controller board, all I/O calls are handled through special vectors, so that if you have some other type of hardware this manual will explain how DISKATE can be customized.

The manual is divided into three major parts. Part I, called Usage, explains how to use the various features of DISKATE. You will want to read this carefully before beginning to use DISKATE, but once you are familiar with the program you will probably consult this section only infrequently. The right hand margin of this part has left space for important keywords so that you can quickly locate where concepts are explained. You may also want to use this space for notes of your own. Part I is divided into sections based on how DISKATE is to be used. This is done for greater coherence in explaining how the commands work, but there will inevitably be some overlap. Some of the commands work differently in monitor usage than in text editing usage, for example, and so are discussed in both sections. All DISKATE commands are introduced in Part I, with the single exception of the V command which can only be used in machine language programs, and so is discussed in Part II.

Part II is called Installation and Maintenance. It explains the nitty gritty of how to get this software running on your hardware, and how to make the proper changes in DISKATE to customize your system. It also gives the details of how to interface DISKATE to machine language programs.

Part III is the System Reference Summary. Almost all the material in Part III is all contained in Parts I and II, but appears here in a condensed form so that you can quickly locate something you need. Experienced programmers who are used to learning new languages from tersely written manuals may want to skim lightly over Part I and mainly consult Part III. If you are confused about what a command does by a discussion in Part I, it may help to refer back and forth to both Parts I and III.

This is a large manual. You may be daunted by the task of reading all of it before beginning any work with DISKATE, but your work will be more pleasurable if you do. Although DISKATE may seem complicated, it is one of the most powerful pieces of software in the personal computer marketplace, and this complexity is a reasonable price to pay for the power you get.

TABLE OF CONTENTS

# Part I: USAGE

## I.1: Using DISKATE as a Text Editor

A text editor is one of the most crucial pieces of microcomputer software. It is through the text editor that you will prepare and maintain a good deal if not all of the material your computer will deal with. DISKATE (ATE stands for A Text Editor) has a variety of editing commands which can work with any type of text. The text is entered into and maintained in an area of the computer's memory called the SOURCE AREA. The Source Area is divided into one or more units called FILES. A file is simply a seqeunce of bytes in the memory, bounded by Ø's. Most of the editor's commands work with a file which is established as the CURRENT FILE. If this file is expanded or contracted, the source area is expanded or contracted along with it to preserve all of the files in the source area. There are special commands which serve to make a different file the current file. Actually, you can have several different source areas in the computer's memory, but more about that later. To work with files that reside on the disk, you will need to move them in and out of memory using the DISKATE I/O commands, which are discussed in I.3.

SOURCE AREA

FILES

CURRENT FILE

When you bring up DISKATE it will issue ">" as a prompt character and wait for a command. There may be blanks between the name of the command and an argument if the command takes one, and blanks MUST be present between arguments if the command takes more than one. Usually in this manual we will omit unnecessary blanks, though in your own work you can include them if it will make the DISKATE commands more intelligible. A blank should never occur within an argument, unless that blank is a significant character as part of the argument.

> IS PROMPT CHAR

There are two key concepts which you will need to be familiar with to understand how the DISKATE commands work: The entry pointer and intervals. The ENTRY POINTER is a special 16 bit location pointing to a memory location which serves as a kind of target for where an action performed by many DISKATE commands is to take place. For instance, let's consider one of the simplest DISKATE commands, which serves to enter text into a file. This command has the form:

ENTRY POINTER

E

```
E[text]
```

COMMAND

(The E stands for Enter.) Note that if this is a command you are typing in all by itself, the prompt

character ">" will be the first character on the line. For instance, to enter the string "I have a weak spot for computers", the line on your terminal should look like:

>E[I have a weak spot for computers]

To avoid confusion the prompt character will generally not be shown, since it is not one that you type.

As you type in the characters of the text one by one, here's what DISKATE will do. Everything in the current file, and in fact the whole source area, starting with the location pointed to by the entry pointer, will in effect be pushed up in memory by one byte and the character typed will be entered into the computer's memory at the location pointed to by the entry pointer. The entry pointer is then incremented. (Actually, this doesn't all happen character by character as you enter the E command, but conceptually you can visualize it that way.) The E command thus inserts the typed characters into the text without overwriting. To simplify things, let's think of the entry pointer as pointing to a character in the file. This character will be called the TARGET CHARACTER. When you enter text with the E command, the text is INSERTED into the file BETWEEN THE CHARACTER PRIOR TO THE TARGET CHARACTER AND THE TARGET CHARACTER.

TARGET CHAR IS CHAR POINTED TO BY ENTRY POINTER

Let's look at an example. Suppose the current file consists entirely of the characters:

My computer is well fed

with the target character being the "w" in "well". (We'll leave aside for the moment the question of how it got that way.) Now suppose we give the DISKATE command:

E COMMAND INSERTS BETWEEN TARGET CHAR AND PREVIOUS CHAR

E[very]

The file will now consist of the characters:

My computer is verywell fed

and the "w" in "well" will remain the target character, (though it will be in a different place in the computer's memory.) Because the "w" is the target character, the "very" goes between the "w" and the preceeding character. If I now enter the command:

E[ ]

the file will consist of the characters:

My computer is very well fed

The text entered with an E command can be any number of lines long (so long as it fits into the area of memory allocated for the source area!) Brackets are allowed within the text, but they must be balanced. For instance,

E[Computers, [of course,] are complicated.]

is a perfectly valid DISKATE command. This entire string of characters will be entered into the file, including the brackets. Now, you ask, what will happen if brackets are not balanced? Trouble, dear user! If you type:

E[Computers, [of]

thinking that this will enter the string "Computers, [of" then the DISKATE command processor will never have received the closing right bracket which ends text entry. The result will be that you may think you are typing commands, when in fact what you type is still being entered into the text. This will garbage up your file with what should have gone as commands. When you are typing in a command, check to make sure that the first character on the line is the prompt character -- ">" -- if not you are probably still in text entry.

WONT RESPOND, STILL IN TEXT ENTRY?

The opposite problem, too many right brackets, will cause the text entry to be terminated prematurely. DISKATE will interpret the rest of the text as a command, which probably won't make sense. When DISKATE gets a command that it can't make sense out of, or an error occurs, it will come back with a question mark.

ERROR MESSAGE IS "?"

What if a piece of editing requires that you insert a string of characters with unbalanced brackets? There is a way to do this, which for the moment we'll postpone: the E command can also be used to enter numerical code, as well as text. This is discussed in section I.2. Note that since parentheses are not used at all in DISKATE, there is no problem with entering a string containing unbalanced parentheses.

The text you enter with an E command may include ASCII control characters in addition to the carriage return. Some care should be taken in doing this, however, since such characters may not show on the terminal and could become hidden booby traps. When you type a carriage return during text entry, only the carriage return character is entered into the file, though both a carriage return and line feed are echoed back to the terminal. If you are using DISKATE to

CR ONLY BETWEEN LINES IN MEMORY

prepare text to be used by another program, you may have to make sure that this program issues a line feed when it encounters a carriage return. One way of doing this is to create an "edit macro" which inserts a line feed into the text immediately after every carriage return. We'll see below how to do this.

BACK-SPACE TO IGNORE CHAR JUST TYPED

If you are entering characters for an E command, or any other command for that matter, and see a mistake on the current line, you can type BACKSPACE to back up the cursor to the place where the error is. Each time you type BACKSPACE, DISKATE will ignore the last valid character on the line, back up the cursor and remove from your screen the offending character. You'll then have to retype the rest of the line. Or, you can type ESCAPE (ESC) to tell DISKATE to ignore the entire line. DISKATE will echo the ESC, then a backslash ("\"), then a carriage return and line feed. If you ESC a line which begins a command, DISKATE will expect a new command and will issue the prompt character. If you ESC a line of an E command which is not the first line, the line will be ignored and you will still be in text entry.

ESC TO IGNORE WHOLE LINE

As you type characters to DISKATE, major changes in the memory do not occur until DISKATE has received an entire line. In the meantime what you have typed is placed in a special internal location called a line buffer. The length of this buffer is 130 characters, which means that DISKATE cannot process a longer line. This should normally cause no problems, since most terminals have a maximum line length of 80 or less. It could show up, however, if you forgot to end a line with a carriage return and kept on going to the next line down on the screen. When the DISKATE line buffer is full, the CURSOR WON'T MOVE. If this happens while you're in the middle of the characters for an E command, you should type BACKSPACE and then a right bracket and a carriage return to terminate text entry to use other editing commands to place part of the line which is too long on the next line. Or, you can simply type a carriage return and keep on entering text. The line buffer always keeps an extra place for the carriage return at the end of a line.

CURSOR WON'T MOVE -- LINE TOO LONG TYPE CAR RET

OR BACK-SPACE ] CAR RET

In addition to this limit on the maximum length a line can be, DISKATE has an internal setting for the width of the terminal's line. This setting can be changed by the user at any time. We won't discuss this at the moment -- a full discussion is contained in section I.3 under the WID command.

Now that we have a basic idea of how text is entered using DISKATE, an important but still very basic function is viewing the text to make sure that it is correct. There are a number of DISKATE commands which

serve this purpose. The simplest is the ' command, which will QUOTE ONE LINE. This command will cause the line containing the target character to be printed on the terminal. The symbol "^" (this prints as a carot on some terminals and as an upward arrow on older terminals), which is used in DISKATE to denote the entry pointer, will appear excatly where any text entered with an E command will go. That is, assuming the line has no invisible control characters, the "^" will go between the target character and the preceding character. The ' command has no arguments, and is a simple way to be sure the entry pointer is where you want it to be. It is highly recommended that you use this command often while you are first getting the hang of DISKATE. It will show you excatly where the changes you are about to make will go -- to the nearest line at least. To use the example from above, if the line with the target character is:

' COMMAND PRINTS LINE WITH TARGET CHAR

My computer is well fed

with the target character being "w", giving the command ' will print the line:

My computer is ^well fed

Obviously we'll need to be able to see more than just the line with the target character. The " command is a more general command that will let us view practically anything. The argument of the " command is an INTERVAL in the computer's memory. This concept of interval is probably the most important one in DISKATE, and so we'll spend some time with it. The underlying idea of an interval is extremely simple: an interval is a pair of addresses in the computer's memory. In order to be valid for use by DISKATE commands, the first address of the pair should not be larger than the second address. For instance, if we use the notation ddddH to refer to memory location dddd hex, the pair 2000H, 2A00H is a valid interval, though it is not yet in a form that DISKATE can recognize. It refers to all of the characters in the memory from 2000H through 2A00H inclusive. This is why the first address must be less than or equal to the second address -- to refer to all of the characters from 2A00H to 2000H would not make sense. To return to the " command, this command takes an interval as its argument and prints on the terminal all the characters in the interval, issuing a line feed also whenever it encounters a carriage return. To see some examples we have to consider how intervals are specified.

INTERVAL IS A PAIR OF ADDRESSES

" COMMAND PRINTS INTERVAL

The simplest way to specify an interval is to give

explicitly the pair of addresses. For instance, the command:

"2000H..2A00H

will cause the characters in the interval from 2000H through 2A00H inclusive to be printed on the terminal. To give a complete explanation of the meaning of the .. symbol at this point would be premature, so for the moment let's leave it that an interval can be specified in the form:

explicit adress..explicit address

Normally you will give the addresses explicitly when you are using DISKATE more as a monitor than as an editor, so there will be more about this in section I.2.

A more sophisticated method of specifying an interval is to use an argument that calls for MATCHING. In using matching you present to DISKATE a pattern of characters, and it searches the memory looking for the pattern. If it finds it then the area that matches the pattern becomes the interval. A simple form of matching argument is similar to one we have already seen with the E command. The argument:

MATCHING

[TEXT]
GIVES
1ST
OCCUR-
RENCE
OF CHARS
OF TEXT

[text]

-- where in this case text is NOT longer than one line
-- will match the first occurrence in the current file of the characters comprising text.

For example, suppose the current file consists of the characters:

NAME: JOAN DOE
OCCUPATION: DEER
SEX: FEMALE
ADDRESS: THE WOODS

The argument [NAME] denotes the interval in the memory occupied by the characters N,A,M,E in the first line of the file -- wherever that happens to be. Note that this form of specifying an interval is CONTENT ORIENTED, in contrast to specifying the addresses bounding the interval explicitly. The argument [NAME] will still denote the first occurrence of N,A,M,E in the file even if the file is changed so that something else occupies the memory locations that N,A,M,E once occupied.

An argument like [NAME] will do us little good as an argument for the " command, since the command:

"[NAME]

will simply print NAME on the terminal. Simple arguments of this form are very useful with other commands, however, as we'll see. What will be very useful with the " command, as well as others, is combining simple patterns such as [NAME] using the operations DISKATE recognizes into much more complex ones. One powerful method is to use the .. construct, which we've seen in connection with explicit addresses. The argument:

PATTERN1 .. PATTERN2

[NAME]..[ADDRESS]

GIVES FROM 1ST PATTERN1 TO 1ST PATTERN2 AFTER THE PATTERN1

refers to the characters from the first occurrence in the current source file of the characters N,A,M,E up to the first occurrecne AFTER the N,A,M,E of the characters A,D,D,R,E,S,S -- so that the command:

"[NAME]..[ADDRESS]

would print on the terminal:

```
NAME:  JOAN DOE
OCCUPATION:  DEER
SEX:  FEMALE
ADDRESS
```

Remember that in using .. in this way, if two patterns are combined as in pattern1..pattern2, DISKATE will look for the first occurrence of pattern2 after pattern1 to find the upper limit of the interval. For instance, if the current file is the same as the example above, the command:

"[A]..[A]

would print on the terminal:

```
AME:  JOA
```

In many editing applications it is useful to view the text in units of entire lines. There are several ways to accomplish this. The character "_" -- which prints as an underscore on most terminals and as a left-pointing arrow on older terminals, is a matching argument which will MATCH A CARRIAGE RETURN. For example, if we are using the same current file as in the examples above, the command:

(UNDERSCORE OR LEFT POINTING ARROW

MATCHES CAR RET

"[SEX].._

will print on the terminal:

```
SEX:  FEMALE
```

The matching in this case works just the way it did in the previous uses of .. to combine intervals. The argument [SEX] will match the first occurrence of the characters S,E,X in the current file. The argument _ by itself would match the first occurrence of a carriage return in the current file, so the argument [SEX].._ will match the interval starting with the first occurrence of S,E,X through the first carriage return after the S,E,X.

CAN USE _ AS ARGUMENT OF E COMMAND TO ENTER CARRIAGE RETURN

The symbol _ can also be used as an argument for the E command. The command:

```
E_
```

will enter a carriage return, just as would the command:

```
E[
]
```

In many cases it is advantageous to use the _ symbol in an E command to have the command fit on one line.

OCCURRENCING

Suppose we want to see more than one line. To do this we can use the same symbol _ in connection with another operation supported by DISKATE: OCCURRENCING.

nPATTERN

An argument using occurrencing gives a number n followed by a pattern and matches THE NTH OCCURRENCE OF THE PATTERN.

MATCHES nTH OCURRENCE OF PATTERN

The argument 2_ by itself will match the 2nd occurrence of a carriage return in the current file. The argument 2[A] will match the 2nd occurrence of the character A. To continue with our same example, the command:

```
"[NAME]..4_
```

will print on the terminal:

```
NAME:  JOAN DOE
OCCUPATION:  DEER
SEX:  FEMALE
ADDRESS:  THE WOODS
```

Here the argument [NAME]..4_ specifies the interval from the first occurrence in the current file of the characters N,A,M,E through the fourth occurrence after the NAME of a carriage return. This will give us four lines from the current file beginning with the first occurrence of N,A,M,E. There are a couple of points to be noted here. Using an argument of the form n_ as we have done will not guarantee that first line of the resulting interval is the entire line from which it was taken.

For instance, the command:

"[JOAN]..4_

would print on the terminal:

JOAN DOE
OCCUPATION: DEER
SEX: FEMALE
ADDRESS: THE WOODS

-- there is nothing in the argument [JOAN]..4_ which instructs DISKATE to begin the interval with the beginning of a line. We'll see how to get entire lines in a minute.

Remember that a construct like pattern..n_ will give you whatever matches pattern through the NEXT n line endings. A common error is to assume it will give you the interval from pattern to the n'th line ending in the current file, which is not in general what will happen.

SUFFIX % EXPANDS TO WHOLE LINE

As we just saw, we will often want to get entire lines. The easiest way to do this is with the % operator. By adding the suffix % to an argument it will normally expand the argument to an entire line. The command:

"[JOAN]%

used with our now weary current file will print on the terminal:

NAME: JOAN DOE

Now suppose we want to see 4 lines beginning with the first line containing JOAN. Here we've got to be careful. The command:

"[JOAN]%..4_

will not give us what we want, but will give an error message! To see why we have to keep at the business of going through exactly how DISKATE will interpret the argument. The argument [JOAN]% is the interval consisting of the first line in the source file containing the characters J,O,A,N. This INCLUDES THE CARRIAGE RETURN at the end of the line. The argument [JOAN]%..4_ is the interval beginning with [JOAN]% through the 4th occurrence of a carriage return AFTER the [JOAN]%. Since [JOAN]% already includes a carriage return, in order for this argument to work there must be at least 5 carriage returns in the file, but altogether there are only 4.

PATTERN% INCLUDES FINAL CAR RET

This explains the error message. The argument gave a pattern which DISKATE couldn't match. Note that in this case, there is nothing syntactically wrong with the command "[JOAN]%..4_ -- the command caused an error because the pattern failed to match. DISKATE does not distinguish in it's error message between syntax errors and "logic errors" -- i.e. for instance errors that arise because a pattern has no match in a particular file, but which might give a perfectly legitimate match in a different file.

Occurrencing together with the % operator gives us a simple way to specify the n'th line in a file. For instance, suppose your terminal is a video terminal with 24 lines, and you want to view your text in groups of 20 lines -- this way there will be a little overlap and room for DISKATE to issue the line with the prompt character. Suppose you want the 3rd such group. The command:

"n_%..m_%

GIVES m+1 LINES STARTING WITH THE nTH LINE

"41_%..19_%

will give the desired result. Remember once again that in a command of the form: "n_%..m_% m will give WIDTH -- that is the number of lines being viewed, and not the number of the final line to be viewed. Again note that even though we want to see 20 lines, we have to use 19_% in the second part of the command because 41_% matches a whole line. If we had said "41_%..20_% this would give 21 lines, not 20. Acutally, the final % in this command is superfluous -- "41_%..19_ will give exactly the same output. (Of course, with the JOAN DOE file we have been using as our standard example, this command would return an error message, since there aren't enough lines in the file to match even 41_%.)

NEGATIVE OCCURRENCE NUMBER COUNTS BACKWARD FROM END OF FILE

Suppose you want to see the last line in the current file but you don't know how many lines in the file there are. This can be easily achieved using occurrencing with a NEGATIVE OCCURRENCE NUMBER. In an instance of occurrencing of the form Npattern, if N is negative DISKATE will look for the Nth occurrence of pattern COUNTING BACKWARDS FROM THE END OF THE CURRENT FILE. The last line of the current file can be viewed by the command:

"-1_%

POS OCCURENCE NUMBER MUST BE <=

At this point we should note that DISKATE considers occurrence numbers to be SIGNED 15 BIT INTEGERS. That means that the high order bit is treated as a sign bit, and negative numbers are represented in two's complement notation. On the other hand, an address is interpreted as a non-negative 16 bit number. When you

give DISKATE a number, the question of whether it will be treated as a signed 15 bit number or a non-negative 16 bit number depends on the context. If you give DISKATE a negative number when it expects an address, the sign bit will be treated as a digit bit, and if you give a large number when it expects a signed number, the highest order bit will automatically be treated as giving the sign bit. This means that an occurrence number greater than 32767 will count as a negative number. This shouldn't pose a problem, since it's very unlikely you would ever need an occurrence number this large.

32767

What if you want to see the last 20 lines in the file? One way to do it would be by the command:

```
"-20_%..19_%
```

This would literally give you 20 lines beginning with the 20th line counting back from the end of the file. However there is an easier way. The command below may throw you for a loop at first, because so far we have presented the symbol .. as an operator which connects intervals into a larger interval. In fact the .. symbol can itself be used as a pattern matching argument. The command we just showed will give the same result as the simpler command:

```
"-20_%..
```

Let's see why this works. If you think of .. as an operator connecting two intervals, then the argument -20_%.. would appear to be incomplete -- the second interval is missing. Instead of viewing .. as an operator, it can be viewed as a MATCHING ARGUMENT WHICH MATCHES ANYTHING. The argument -20_%.. can be read as:

..
AS
MATCHING
ARG
MATCHES
ANYTHING

> the line containing the 20th carriage return back from the end of the file, followed by anything.

Because DISKATE is working with the current file, the term "anything" is restricted in scope to the current file. Notice also there is a construct here which we haven't seen before: a matching argument consisting of one pattern followed by another pattern. This is called CONCATENATION. Writing one pattern directly after another specifies the pattern obtained by joining the two patterns together. The pattern:

CONCAT-
ENATION

```
[JOAN][ DOE]
```

is exactly equivalent to the pattern:

[JOAN DOE]

Concatenating two patterns of the form [textl][text2] is not very useful, because instead you could just use the pattern [textltext2]. Concatenation is very important, however, when the two patterns are specified by different means. A pattern of the form [text]_ specifies the pattern consisting of text followed by a carriage return. There is no way to specify this pattern directly in the form [text] because a pattern of this form cannot extend accross more than one line.

CAN USE CONCAT-ENATION WITH E COMMAND

You can also use concatenation with the E command. For instance, the command:

E_[TABLE:]_

would enter first a carriage return, then the characters T,A,B,L,E,:, and then another carriage return.

If .. is a matching argument, then how should we interpret the command:

".. ".. PRINTS ENTIRE CURRENT FILE

? Literally it would mean, print "anything" on the terminal. Remember, though, that anything is restricted in scope to the current file. Thus the command ".. will PRINT THE ENTIRE CURRENT FILE ON THE TERMINAL. Suppose we want to see everything after the 35th line. The command:

"35_..

will do what we want. Acutally, to be completely precise this is not quite correct, since this will give us the 35th carriage return followed by anything. Seeing this extra carriage return at the beginning will not do us any harm, of course, but an argument error like this can be very serious with other commands. To get exactly what we want we should really use the command:

"36_%..

since the interval consisting of "everything after the 35th line" is the same as "everything starting with the 36th line". It is a frequent but inescapable nuisance in many areas of programming to worry about whether something should be n or n+1.

Similarly to get the first 5 lines of the current file, though the command "_%..4_% would work, a simpler command is: "..n_

"..5_ GIVES

1ST n LINES

Literally the argument ..5_ means "anything, followed by the 5th carriage return".

Now is the time to confess to a little white lie. So far in this manual we have seen the use of the symbol .. both as an operation for combining intervals, and as a matching argument. In fact, instead of .. you can use any number of dots from one on up -- so, for example the argument pattern1.pattern2 has the same meaning as pattern1..pattern2 or pattern1...pattern2. Let's see why this "ambiguity" causes DISKATE no problems. The actual symbol which DISKATE recognizes internally is the single dot. Since . can be interpreted as a matching argument which matches anything, .. can be thought of as a matching argument which means: anything followed by anything. Clearly this has the same meaning as just plain anything, which is why any number (greater than zero) of dots amounts to the same thing. The use of two dots here is simply personal preference. A single dot by itself in an argument is easy to miss, which is why two are being used in this manual, but in your own work you should use either one, two or three dots together as your preference suits.

. .. ... ETC.

ALL HAVE THE SAME MEANING

We've seen so far that specifying an interval which matches a given line number, or a give group of lines identified by line number, can be slightly tricky. DISKATE provides a way to deal with numbered lines directly which eliminates some of this difficulty, but which is not as general in building complex expressions. The symbol ! is similar to % in that it refers to an entire line, but rather than being a general suffix which can be appended to any argument, the symbol ! is only placed after an occurrence number to specify the n'th line in the current file. Specifically the argument:

n!

SPECIFIES THE N'TH LINE DIRECTLY -- NO MATCHING

n!

specifies the n'th line in the current file. This is NOT A MATCHING ARGUMENT. When evaluating n!, DISKATE searches through the current file for the n'th line without regard to anything else. Let's see how this compares to our previous examples.

Above we saw an example of a " command using expressions of the form n_% for showing 20 lines beginning with line 41. To do this using the ! symbol we could use the command:

"41!..60!

Notice here that 60! is not a matching argument, so it definitely does not match the 60th line after the 41st

line. Instead, 60! simply specifies the 60th line -- period, no matter where the expression 60! occurs.

The ! is very useful when you want to see specific numbered lines in a file, but not as useful when you want to specify lines relative to a content oriented argument. We saw for instance, that the command:

"[JOAN]%..3_

will print on the terminal four lines beginning with the first occurrence in the current file of JOAN -- WHEREVER that happens to be. The command:

"[JOAN]%..3!

would give an error message if the first occurrence of JOAN happened to be beyond the 3rd line in the current file. Even though it may seem like more work at first to learn to specify intervals using expressions of the form n_%, because these expressions can be used more widely than expressions like n! it's a good idea to go to the extra effort.

There is another important way in which expressions of the form n! differ from expressions like n_%. We saw that the argument -1_% refers to the last line in the current file, the expression -2_% the next to the last line, and so on. In evaluating expressions of the form n!, DISKATE will TREAT ALL OCCURRENCE NUMBERS AS POSITIVE. If you specify n! and there are not n lines in the file, n! will evaluate to the address of the 0 which forms the upper bound of the file in the computer's memory. So, -2! will not give the next to the last line in the current file, but will give the address of the 0 at the end of the file, as will -3! and so forth, since -2 is the same thing as 65534, which is surely larger than the number of lines in the file. This means that

-1!

is a convenient expression for the 0 at the end of the current file. We'll have occasion to need this.

Since DISKATE provides the facilities for referring to locations in the current file by line number, in more than one way in fact, you might wonder if there is a way for DISKATE to provide you with the numbers of the lines in the current file. There is indeed a way, but we'll postpone a detailed discussion until section I.4 because it is keyed to the assembler. To jump the gun a bit, the P command is similar to the " command and will print the interval given as argument with line numbers. However, if you give this command with a file consisting of ordinary prose text, THE TEXT WILL COME OUT IN A PECUL-

n!
TREATS n
POSITIVE

IF TOO
BIG
GIVES
ADDRESS
OF 0
AT END
OF FILE

-1!

GIVES 0
AT END
OF FILE

P
COMMAND

PRINTS
INTERVAL
WITH
LINE
NUMBERS

IN

IAR FORMAT with all kinds of spaces where they shouldn't be. The P command is designed to print assembler source code, and the spacing is so that the listing will conform to a convenient assembler format. If you're able to recognize lines properly in spite of this format, you can use the P command to obtain line numbers for use in expressions like n_% and n!. The use of line numbers obtained in this way is a very good way of making sure that an occurrence of a pattern is in the right place.

ASSEMBLER FORMAT

(MANY EXTRA SPACES)

There is a way to make the P command easier to use for editing prose by using a command which we'll discuss in more detail later. The command:

```
TAB Ø Ø Ø Ø
```

will suppress most of the format pecularities. It can still make your text look slightly odd, since if there are multiple spaces surrounding the first three words on a line, the multiple spaces will be reduced to only a single space, so if the first line of a paragraph is indented, for instance, this indentation will be suppressed. However, giving this command should allow you to view prose text with line numbers conveniently.

It may seem that we've been spending an undue amount of time with the " command. DISKATE has many more commands than just the ones we've discussed so far. The reason for spending so much time with the " command is to develop the concepts of interval, matching, and many of the ways that intervals can be specified by using for illustration a command which simply shows us the interval in question. Although we are not finished with seeing new operations by which you can combine intervals, perhaps now's the time to begin introducing some more commands. Before doing this I should emphasize as strongly as possible that if you are ever in doubt as to just what interval an argument actually specifies, it's always best to give DISKATE a " command and see.

So far the DISKATE commands we've discussed have been concerned with two kinds of operations: entering text, and viewing text already entered. Another important major operation involved in editing is DELETING text. In DISKATE this is accomplished by the K command, which will KILL, or delete, the interval given as the argument. Let's go back to our standard example:

K COMMAND

KILLS (DELETES) INTERVAL

```
NAME:  JOAN DOE
OCCUPATION:  DEER
SEX:  FEMALE
ADDRESS:  THE WOODS
```

Suppose we want to make the following changes, which

both involve only deletions: the name JOAN is to be changed to JON and FEMALE to MALE. Let's do the second one first. What we want to do is delete the FE from FEMALE. It happens in this case that the only occurrence of FE in the file is in FEMALE, so the command:

K[FE]

will do what we want. But!!! Remember that the argument [FE] specifies the FIRST occurrence of the characters F,E in the file. How do we know for sure that this first occurrence is in the right place? Of course, in this case the whole file is small enough that we can look and see, but even then it is possible to make a mistake. The best thing to do is ALWAYS USE A " COMMAND TO MAKE SURE THE INTERVAL YOU GIVE IS THE ONE YOU WANT. If we were being careful we would have said:

"[FE]%

and DISKATE would have responded with:

SEX: FEMALE

Then we would have the confidence that the K[FE] command would do what we intend. When you give a K command, the only thing that will be printed on the terminal is the prompt character when DISKATE is ready to accept another command -- unless the argument you give involves matching and DISKATE cannot find anything that matches, in which case an error message will be printed. If we're being as cautious as possible, perhaps we ought to take a look after the K command to make sure everything looks the way we want it -- this is after all the goal of text editing. The simplest way to get a quick look is to use the ' command. If we give the command ' right after the K[FE] command, then DISKATE will print on the terminal:

SEX: ^MALE

Let's take a careful look at this. Remember that the ' command prints the line with the target character, putting the symbol "^" between the target character and the previous character -- i.e. exactly where any text entered with an E command would be inserted. It's been a while since we mentioned the target character and its cohort the entry pointer. The " command does not affect the entry pointer, but the K command will SET THE ENTRY POINTER. After a K command, the TARGET CHARACTER WILL BE THE FIRST CHARACTER AFTER THE INTERVAL DELETED. That's why the "^" is in front of MALE in the line shown above. This setting of the entry pointer can be both

K COMMAND SETS TARGET CHAR = 1ST CHAR AFTER INTERVAL DELETED

useful and also cause severe difficulties if not kept in mind.

The difficulties can arise because it frequently happens that you want to enter some text, then make some changes using the K command, and then go on to enter more text. Becuase the K command resets the entry pointer, if you're not careful to put the entry pointer where you want it, the text you enter thinking it is going at the end of the file will actually be going into the middle. The reason DISKATE resets the pointer, however, is that a K command followed by an E command will REPLACE the deleted interval by the text given with the E command, which is extremely useful. These two considerations are very important, but in order to see some more examples of using the K command, we'll postpone a detailed discussion.

Now let's do the second change we had planned: changing JOAN to JON. In this case what we need to do is delete the A from JOAN. Before getting involved, let's give a ".. command and see what the file looks like. In response to ".. DISKATE will print:

```
NAME:  JOAN DOE
OCCUPATION:  DEER
SEX:  MALE
ADDRESS:  THE WOODS
```

Right away we can see that to simply say K[A] would be trouble. The first occurrence of A in the file is in the word NAME, so K[A] would delete the A from NAME, which is not at all what we want. One approach would be to try to count the number of A's in the file before the one we want, so we could specify the right one by using occurrencing. In this case the A in JOAN is the second A in the file, so the command:

```
K2[A]
```

will in fact do what we want. But this method has real difficulties. First of all, we can hardly find out using a "2[A] command whether we've got the right occurrence of A, since "2[A] will only print A -- not telling us WHICH A it's printing. One way to remedy this would be with the command:

```
"..2[A]
```

This would print the whole current file up to the 2nd occurrence of A, which would be a way to make sure that the 2nd A is really what we want. But, a more serious problem with this approach is that in a long file, it would hardly be convenient to have to count occurrences

just to specify the one we want. At this point we are really getting to the whole crux of using DISKATE as a text editor. Because matching is content oriented, it allows us to do some incredibly powerful things, but at the same time we have to take some care to be sure that when an interval is specified by matching it is the one we want. Although this particular change is simple, and we've already seen one way that will work, let's look at several different methods that will be very helpful in a great variety of situations.

Let's suppose for the moment that we can be confident that the JOAN in which we want to delete the A is the first occurrence of JOAN in the file. Here, of course, we can be because the file is so simple. The thing that we want to do is: delete the A in JOAN. DISKATE has a construct which reflects exactly this meaning. We have said repeatedly that in determining intervals using matching arguments, DISKATE works with the current file. Actually this has been an oversimplification which normally does no harm, but at this point we need to be more precise. Whenever an argument involves matching, before the argument is evaluated DISKATE restricts the scope of the argument to an INITIAL REFERENCE INTERVAL. For most commands this initial reference interval is the current file, which is why our previous oversimplification is normally not misleading. DISKATE has a special operation which allows you to in effect set the initial reference interval to something else. Whenever DISKATE encounters a | in evaluating an argument, THE INITIAL REFERENCE INTERVAL FOR THE REST OF THE ARGUMENT IS THE INTERVAL CORRESPONDING TO THAT PART OF THE ARGUMENT ALREADY EVALUATED. The easiest way to understand this will be by example.

INITIAL REFERENCE INTERVAL

GIVES THE SCOPE OF MATCHING

(USUALLY CURRENT FILE)

|

SETS INITIAL REF. INTERVAL FOR REST OF ARG TO PART OF ARG TO LEFT OF |

The argument [A] will, to be precise, match the first occurrence of A in the initial reference interval, which is normally the current file. The argument [JOAN] will match the first occurrence of J,O,A,N in the initial reference interval, which is normally the current file. The argument [JOAN]|[A] will match the first occurrence of A WITHIN the first occurence of J,O,A,N in the initial reference interval, normally the current file. The command:

```
K[JOAN]|[A]
```

will achieve exactly what we want -- provided, of course, as we mentioned, that we are sure that the JOAN in question is the first occurrence of JOAN in the current file.

In a larger file, however, we would still have the general problem of making sure that we were in the right

place for a change. In the same way that we saw before, issuing "[JOAN]|[A] or even "[JOAN] will not tell us we are making the change in the right place, because it will not tell us which occurrence of JOAN we are looking at. Let's suppose that we have a file in which we want to make exactly this same change -- deleting the A from an occurrence of JOAN, but there are many occurrences of JOAN in the file and we want to be very careful to delete only the right one. The command:

"..[JOAN]

will print everything in the current file up to the first occurrence of JOAN. Suppose this is not the right one, and we want to print the rest of the file up to the next JOAN. One way of doing it would be:

"[JOAN]..2[JOAN]

There's nothing wrong with this, but if this is not the right JOAN either, we would have to be keeping track of the occurrence number. Here is a command that you can keep on issuing successive times to see the intervals of text between successive JOAN's:

">..[JOAN]

>

This command uses a feature we haven't seen before. So far the only time we have encountered the character ">" is as the prompt character. DISKATE recognizes this symbol as a special argument which does not involve matching. > denotes the upper address of the LAST INTERVAL COMPUTED BY DISKATE. Since we just got through with the command "..[JOAN], in this case > will be the address of the N in the first JOAN in the file. The argument >.. then will match everything in the current file from the N in the first JOAN onward.

GIVES UPPER ADDRESS IN LAST INTERVAL COMPUTED

There is an important but subtle point here that we should note before going on. In all the cases we have seen, DISKATE arguments are intervals -- i. e. a pair of addresses. But here we've just said that > denotes the upper address in the last interval computed by DISKATE. This implies that > denotes not an interval but a single address. In fact, any symbol or expression in DISKATE that determines a single address is also interpreted as AN INTERVAL WHOSE UPPER AND LOWER ADDRESSES ARE EQUAL. We said that an interval was a pair of addresses in which the first address was not greater than the second, but they certainly may be equal. For example, in evaluating arguments DISKATE takes the expression:

SINGLE ADDRESS EQUIVALENT TO INTERVAL WITH UPPER AND LOWER ADDRESS THE SAME

2000H

to be completely equivalent to the expression:

2000H..2000H

Note that this last makes perfect intuitive sense.

Now let's return to the example. We started with the command "..[JOAN], and could see from it that the first occurrence of JOAN was not the right one. What we said we wanted to do, then, was view the rest of the file to the next JOAN. Our command ">..[JOAN] will do exactly that, with a minor bit of nuisance. This command will print the interval from the N in the JOAN we just saw to the next occurrence of JOAN. If this JOAN is not the right one either, we can type exactly the same command, ">..[JOAN], and keep on typing it, to see the consecutive intervals of text prior to successive JOAN's. Actually, as we'll see at the end of section I.1, there is a special command called DEF which is very useful in exactly this kind of situation where we want to keep giving the same command repeatedly. The DEF command will allow just carriage return to invoke a special default command.

The minor bit of nuisance referred to is the fact that the N from the last JOAN we looked at will begin the printout. (We'll see in a minute how to get rid of that.) The second time we type ">..[JOAN] the symbol > will be the upper address from the last interval computed, which in this case will be the N from the second occurrence of JOAN. Each time we type the command, a new interval will have been computed, so the value of > will change. Note that to use this method, you must not use any other command between the instances of ">..[JOAN], since an intervening command will probably cause an interval to be computed, so that > will not have the value we want it to.

After having typed in the command ">..[JOAN] enough times, we should come to the JOAN we want. But now there is a problem. We want to kill the A in the JOAN in the interval we just saw -- but how do we specify it? There are different possible approaches, but here is a simple one which will show another feature of DISKATE we haven't seen before. In the interval that we just computed, we can be absolutely sure that there is only one occurrence of JOAN, since we set things up so that the command ">..[JOAN] is showing us the interval of text until the NEXT occurrence of JOAN. What we want to do is delete the A in the JOAN -- and we know there is only one -- in the interval just computed. The following command will do exactly that:

K|[JOAN]|A

Let's take this carefully. The first thing we see in this command is K, which is the symbol for the Kill command. The very next character in the command is a |. We have said that | causes the part of the argument already given to become the initial reference interval. But here we haven't given any argument yet at all! What we're seeing is an example of an extremely useful DISKATE feature: IF A COMMAND REQUIRES AN INTERVAL AS ARGUMENT, AND NO ARGUMENT AT ALL IS GIVEN, THE ARGUMENT USED IS THE PREVIOUS INTERVAL COMPUTED.

MISSING ARGUMENT

USES PREVIOUS INTERVAL COMPUTED

So. The letter K says that this is a kill command, and what follows is to be the argument. The | says to take that part of the argument already computed and use it as the initial reference interval. Since no argument thus far has been given, the | immediately after the K will cause the initial reference interval to be the previous interval computed. That interval, remember, went from the N in the JOAN before the one we want, to the next JOAN, which is the one from which we want to delete the A. So, the argument up through |[JOAN] specifies the first occurrence of JOAN in the previous interval computed, which is just the JOAN we want. The next | sets this, that is the JOAN, as the initial reference interval, so that the total argument |[JOAN]|[A] specifies:

> the A within the first JOAN within the previous interval computed before this command is executed.

This is exactly what we want, so the K|[JOAN]|[A] does the trick. Note that this is an example of using more than one | in a single command. You can use the | repeatedly in this way to keep narrowing down the scope of matching to get what you want.

We saw that the symbol > can be used to give the upper address of the last interval computed by DISKATE. There is a corresponding symbol which denotes THE LOWER ADDRESS OF THE LAST INTERVAL COMPUTED: < . Here are some ways this can be used. Suppose you just saw an interval using a " command, and it didn't amount to a whole line, and you want to see the line containing this interval. The command:

<

GIVES THE LOWER ADDRESS OF THE LAST INTERVAL COMPUTED

```
"<%
```

will do the trick. Notice that the command:

```
"|.._
```

will not work. The | right after the " sets the initial

reference interval to the last interval computed, so the whole argument |.._ means the first line of the last interval computed. But the last interval computed didn't contain an entire line, so this command will give an error. The | operation allows you to specify an interval with a given interval, but there can't be any overlap. With < and > you can see intervals that do overlap the last interval computed. The command:

"<..3_

will show you three lines beginning with the first character of the last interval computed. (If that character happpens to be a carriage return you will appear to see only two lines.) It doesn't matter one way or the other whether the previous interval computed contained three lines or not.

Care must be taken in using < in complex arguments, since its value will change as the argument is evaluated. < is ASSIGNED A VALUE AS SOON AS THE LOWER ADDRESS OF THE INTERVAL IS DETERMINED. So, if you wanted to see everything in the current file up to the first character of the last interval computed, you could NOT use the command:

< GIVEN VALUE AS SOON AS LOWER ADDRESS OF INTERVAL IS DETERMINED

"..<

since as soon as DISKATE sees the .. it knows the lower address of the interval to be computed will be the beginning of the current file. < is then assigned this value, so the argument ..< specifes only the first character of the current file. The value of < can change several times as a single argument is evlauated. For instance, in the argument:

[JOAN]|[A]

< is first set to the address of the J in the first occurrence of JOAN in the file, and then set to the address of the A within that. An argument with several instances of | may cause < to be reevaluated several times during a single argument.

A value is not assigned to > until either the whole argument is evaluated or a | is encountered. So, the argument:

> GIVEN VALUE WHEN WHOLE ARG EVALUATED OR AT |

[JOAN]|>

specifies the N in the first occurrence of JOAN in the current file. If you are in doubt about what < or > should mean, give DISKATE a simple test case using a " command to see the result.

Above we saw how we could see successive intervals of text ending in the next occurrence of JOAN, but mentioned that by using the command ">..[JOAN] the first character printed would be the N in JOAN. It certainly does no harm to keep having this N printed, since this is only a " command, but it would be nice to know how to get rid of it. It's always good practise to be able to specify precisely the interval we want, because a slight nuisance such as this initial N could turn into a significant error with other commands. The interval >..[JOAN] includes everything we want to see, but what we want is the characters from the interval >..[JOAN] starting with the first character after the initial N. The command below will be an effective replacement for the ">..[JOAN] eliminating the initial N problem. It introduces a new matching symbol which we haven't seen before.

```
">..[JOAN]|2@..                                          @

     The symbol @ will match ANY SINGLE CHARACTER. As    MATCHES
an argument all by itself it matches the first character ANY
in the initial reference interval. In the command        SINGLE
above, the argument to the left of the | is exactly what CHAR
we used before, so let's look at the part to the right
of the |. 2@ matches the 2nd occurrence of the pattern
@, or the 2nd occurrence of any character. In other
words, 2@ matches THE 2ND CHARACTER in the initial       n@
reference interval. 2@.. therefore matches everything    MATCHES
in the initial reference interval starting with the 2nd  THE
character. If we're using the command above as a         nTH CHAR
replacement for the ">..[JOAN], before the command above
is given, the last interval computed will presumably be
the interval of text starting with the first character
after an occurrence of JOAN, or the first character in
the current file if we are just starting out. So,
>..[JOAN] will match everything from the N in the cur-
rent JOAN to the next JOAN. The | sets this as the ini-
tial reference interval, so the argument >..[JOAN]|2@..
becomes everything starting with the second character
WITHIN: the interval from the N in the current JOAN
through the next entire JOAN. This is exactly what we
want -- the same interval we were getting before, but
with the first character trimmed off.
     Here's a different kind of example that shows how
you can use the | operation together with the @ matching
symbol. Suppose the file you are editing is a program,
and you want an argument which will match the first
occurrence of a variable name, let's say BUFFER, which
occurs at the BEGINNING OF A LINE. To say that it
occurs at the beginning of a line means either that it
will be the very first thing in the file, or else it
```

will be immediately preceded by a carriage return. To simplify things let´s suppose we can rule out the first possibility. Here´s the argument that will work:

_PATTERN
|2@..

MATCHES PATTERN AT BEGINNING OF LINE

```
_[BUFFER]|2@..
```

The argument _[BUFFER] matches the first occurrence of a carriage return followed by the characters B,U,F,F,E,R, so the whole argument matches everything from the 2nd character on within the first occurrence of a carriage return followed by BUFFER. That is, the argument matches the everything in _[BUFFER] after the carriage return, which is what we want.

Let´s return now to our example of deleting the A in JOAN and see another way it can be done. Recall that the last way we did it, we ended up marching through the whole current file using " commands until we found the right JOAN. But suppose we could be sure we had the right JOAN just by seeing the line containing it. Here´s an approach where we only view a line at a time, which will introduce a new DISKATE command that is very important to keep in mind in conjunction with the K command. We recall that the ´ command is a simple command with no argument that will show us the line with the target character, and the position of the target character in the line. What we´re going to do is set the entry pointer to successive JOAN´s and use the ´ command to view the line with the target character, until we get the right JOAN. Then we can go ahead and use the K command. The first step is to set the entry pointer to the first JOAN in the file. This is accomplished with the command:

^

COMMAND

```
^[JOAN]
```

We encountered the symbol "^" before -- DISKATE prints this symbol between the target character and the preceding character in response to a ´ command. When used as a command, ^ SETS THE ENTRY POINTER TO THE LOWER ADDRESS IN THE INTERVAL GIVEN AS THE ARGUMENT. That is, the ^ command sets the target character as the first character in the interval given as the argument. In this case, [JOAN] matches the first occurrence of JOAN in the file, so the ^[JOAN] command sets the entry pointer so that the J in this JOAN becomes the target character. Now if we give a ´ command and the target character is the J in the wrong JOAN, the command:

SETS TARGET CHAR TO FIRST CHAR OF INTERVAL GIVEN AS ARGUMENT

```
^>..|[JOAN]
```

will set the J in the next JOAN to be the target character. Again we can give the ´ command to see the line

with the target character to check to see whether we´ve got the right spot. Note that because the ´ command does not take an argument, the symbol ">" has the same value after the ´ command as before it. What command shall we use once we have the right JOAN? There are several different commands we could use that would all have the desired effect. As before, K|[JOAN]|[A] would do. Or, since we know that the target character is the J in the JOAN we want, we could use the command:

K^..|[A]

^ IN ARGUMENT

In this command we see still another use of the symbol "^". The command is a Kill command, so "^" is not used here as a command. Rather, "^" is used here as part of an argument. The symbol "^" used as an argument gives AN INTERVAL BOTH OF WHOSE ADDRESSES ARE THE VALUE OF THE ENTRY POINTER. Or, you can think of "^" as a SYMBOL WHICH MATCHES THE TARGET CHARACTER. So, the argument in the command above, translated literally, means the first A within everything in the current file starting with the target character. Since we already arranged for the target character to be the J in the JOAN from which we want to delete an A, this command will work.

MATCHES TARGET CHAR

In fact there is a command for deleting the A in JOAN in this situation which is even simpler. When we used the ^ command to set the target character to be the J in successive JOAN´s, we also set the last interval computed to be the whole JOAN. So, by the time we´ve found the right JOAN we will know that the last interval computed will be this JOAN. All we need to say, then, is:

K|[A]

The | before any argument is given sets the initial reference interval to the last interval computed, which is the correct JOAN, and within this interval the first A -- the only A in fact -- is just what we want to delete. Note that if we use this command for the deletion, the main function of having used the ^ command was to be able to see where we were using the ´ command. It is often not a bad idea to use the ^ command this way, even when you are not sure whether you will need the target character to be set to the first character of the argument in question.

MULTIPLE COMMANDS ON ONE LINE SEPARATED BY

Perhaps this is the time to mention an important feature of DISKATE. So far we have been giving examples of DISKATE commands one by one. Actually, you can give DISKATE an ENTIRE LINE OF COMMANDS at once SEPARATED BY COMMAS. Rather that giving two separate

commands:

COMMAS ARE ALLOWED

^>..|[JOAN]
'

it's easier to put both commands on a single line:

^>..|[JOAN],'

By now it should be clear that there will almost always be many different ways of accomplishing a given thing in DISKATE, and after using it for a while you will surely develop your own style. The examples presented here reflect the style of the author of this manual -- a style which you may not care for once DISKATE becomes familiar. One of the things that has been stressed above, and will be stressed below, is that you should always look before you leap -- give DISKATE a " command or ' command which will preview any change you are about to make. Constantly giving such viewing commands may seem tedious, but they will save an enormous amount of grief that can result from giving an erroneous command.

We've seen many examples of various simple deletions -- deleting an A from an occurrence of JOAN to change it to JON. The deletion itself was usually easy, but the work came in specifying where it was to take place. Now let's suppose we wanted to change the JOAN instead to JOHN -- deleting an A from JOAN and inserting an H. Without repeating all of the previous examples, let's suppose we have set things up, by various " commands perhaps, so that in the last interval computed the first JOAN is the one we want to change. We saw that the command:

K|[JOAN]|A

will delete the A from this JOAN. Now we want to add in the H. As mentioned above, the K command sets the target character to be the first character after the interval deleted. After the command K|[JOAN]|A, then, the target character will be the N in the JON. (We just deleted the A.) To insert the H we will want to use an E command, and remember the E command inserts the characters of its operand between the target character and the preceding character. Since the target character is now the N in JON, anything entered with an E command immediately after the K|[JOAN]|[A] will go between the N in JON and the previous character, which will be the O in JON. This is exactly where we want to put the H. The command:

K FOLLOWED BY E

REPLACES KILLED INTERVAL

```
K|[JOAN]|[A],E[H]
```

WITH ENTERED TEXT

will replace the A in the JOAN with an H in a single command. In general, Kinterval,E[text] will REPLACE the characters in the interval with text.

TO RESUME TEXT ENTRY AFTER MAKING CHANGES,

At this point we come to what is possibly the greatest single potential source of error in ordinary DISKATE usage. Regardless of whether you want to use an E command to replace with new text an interval which has been deleted with a K command, the K command will set the entry pointer so that the target character is the first character after the deleted interval. It frequently happens in using a text editor that you will enter some text, discover an error perhaps, go back and make a change, and then resume entering more text. Here we have a problem. Any change in the file made using the K command will reset the entry pointer. If we don´t somehow save the value of the entry pointer before the change is made, and then restore it after making the change, when we go back to entering text we will be entering it in the wrong place. How do we get the entry pointer back to where it was before we made the change? As usual there are several methods.

ENTRY POINTER MUST BE RESTORED !!!!!!!!

VARIABLE NAMES:

LETTERS & DIGITS STARTING WITH LETTER

Perhaps the safest method is to save the value of the entry pointer and then restore it. This can be done using a DISKATE feature we haven´t seen before: variables. A DISKATE VARIABLE is named by a string of characters beginning with a letter, and containing only upper case letters and digits. Variable names can be of any length. When you bring up DISKATE certain variables used by the assembler will already have values, and should not be changed unless you are certain you will not be using the assembler. These variables are:

VARIABLES USED BY ASSEMBLER

```
A, B, C, D, E, H, L, M, SP, PSW
```

A variable can be assigned a value using the = command, which has the form:

=

COMMAND

```
variable=argument
```

VAR=ARG SETS VARIABLE TO LOWER ADDRESS OF ARG INTERVAL

In this case there MUST NOT BE A BLANK between the = sign and the variable name, since if there were DISKATE would interpret the variable name as the name of a command. There can, however, be blanks between the = sign and the argument if you like. The argument can be any DISKATE argument, and the variable is given the value of the LOWER ADDRESS of the interval specified by the argument. So, if you are going to make some changes in text you have just entered with the E command, and want to save the entry pointer, you can use the command

ENTRY=^

and then when you want to restore the value of the entry pointer give the command:

^ENTRY

Note that the ^ command does not use an =, and automatically assigns the value of the lower address of the argument to the entry pointer. (In fact ^=ENTRY would give an error message, since ^ is not an ordinary variable.) The choice of "ENTRY" as the name of the variable is of course arbitrary, and you may want to use shorter names. Be careful, though, about using single-letter names. If we used the variable E instead of ENTRY, this could cause real problems since E is one of the variables used by the assembler. Whenever you are ready to begin entering text after having made some changes, it´s always a good idea to give a ´ or " command to make sure the entry pointer is where you want it to be. If you make a change, fail to restore the entry pointer, and then resume text entry, you may find your text looking much more botched up than it was before you made the change.

While a simple way to save the contents of the entry pointer is to set a variable = ^ and then vice versa, as we just saw, you must be extremely careful when you do this. Here is the problem. In many cases you will want to enter new text at a given RELATIVE LOCATION within the source file. If you save the contents of the entry pointer in a variable, make a change in the text, and then restore the entry pointer to the original address, the entry pointer may not be where you want it to be if the change you made in the text changes its size. In this case the entry pointer must be restored by invoking a command which sets the entry pointer to a pattern matching argument. We can see how to do this using commands that we already know. Suppose there is a simple pattern that you can be sure does not occur anywhere in your text, such as: !!!!! Before changing the entry pointer to make the changes in your file, give the command:

TO RESTORE ENTRY POINTER TO RELATIVE LOCATION

ENTER UNUSED PATTERN BEFORE MAKING CHANGE,

E[!!!!!]

then after the changes have been made, the command:

K[!!!!!]

THEN KILL IT

will remove the exclamation points and restore the entry pointer in a single step. Of course, the entry pointer will end up in the wrong place if this pattern should

actually occur in the file higher up.

We'll have a great deal to say below about the special case of setting the entry pointer to the end of the file.

Variables can also be used as occurrence numbers. We saw above examples of how to march through your file seeing successive instances of a given text pattern. Another way to do it is using variables for occurrence numbers. Suppose for instance that your file is a list of lines, each one having a similar format to:

```
NAME:  JOAN DOE
OCCUPATION:  DEER
SEX:  FEMALE
ADDRESS:  THE WOODS
```

and you wanted to page through successive entries. To do it using variable occurrence numbers, begin with the command:

```
X=1
```

Then to see the next entry you can use the command:

```
"X[NAME]..4_,X=X+1
```

Note once again that we must use the pattern matching symbol _ here rather than !, which specifies an absolute line number. One slight drawback to this method is that in evaluating the argument X[NAME]..4_ DISKATE must search the entire current file from the beginning each time, so it will tend to be a little slow.

The + we haven't encountered before, so it's worth taking a minute to understand what it does. On the surface a command like X=X+1 appears straightforward -- it simply adds one to the value of X. However, DISKATE arguments are generally intervals, and a single number is simply an interval where both addresses are the same. + can be used with any argument, so we should clarify what it means with a general interval. An argument of the form:

```
arg+value
```

is evaluated by adding value to BOTH addresses of arg. In our standard example, for instance, the command:

```
"[NAME]+1
```

will print A,M,E,space on the terminal. Arithmetic operations you can use are +, -, *, and / with their usual meaning. Division will give only the quotient

| ARG1 |
|---|
| + |
| - |
| * |
| / |
| ARG2 |
| |
| PERFORMS OPERATION ON BOTH ADDRESSES OF ARG1 WITH LOWER ADR OF ARG2 |

discarding the remainder, and multiplication will give only the low order 16 bits of the (possibly) 32 bit product. However, these operations are carried out left to right with the same priority, so that for instance

ARITH. OPS EVALU- ATED LEFT TO RIGHT SAME PRIORITY

```
1+2*3
```

evaluates as 9 rather than 7 as it would in a typical BASIC. As with +, all of the arithmetic operations operate on both values of an interval argument.

Let's see some ways these operations can be used. Suppose you want to see the first 10 characters of the last interval computed by DISKATE. One way to do it would be by the command:

```
"|..10@
```

This would say, print everything up to the 10th character within the last interval computed. Another way would be:

```
"<..<+9
```

This method does not actually call for any matching. It says to print the interval from the lower address of the last interval computed through 9 more characters.

^^+n

Suppose you wanted to advance the entry pointer by one character. The command:

ADVANCES ENTRY POINTER n CHARS

```
^^+1
```

will do the job. Likewise ^^-1 will back it up by one character. Many other uses of the arithmetic operations will show up when using DISKATE as a monitor which we'll see in Section I.2.

TO ADD TEXT TO THE END OF THE FILE

Let's return to the problem of restoring the entry pointer after changes have been made so that text entry can continue in the proper place. There is one problem with the method of saving the value of ^ with a variable and then restoring it. What if you have made some changes, with a K command say, but suddenly don't remember whether you saved the value of ^ or not? It would be useful to be able to put the entry pointer in the right place regardless of what you did before. Fortunately it's easy to specify where this place will usually be. If you want to go on entering new text after changes have been made in what has already been entered, the new text should go at the END of the current file. Because the E command places text between the target character and the preceding character, to add text to the end of the file you want the target character to be THE CHARACTER AFTER THE LAST CHARACTER IN THE FILE. How

TARGET CHAR MUST BE CHAR AFTER LAST CHAR IN THE FILE

can we specify this? The argument .. specifies the current file. The argument ..|> specifies the upper address of the current file, since the | causes a new value to be assigned to >. The address we want is one higher than ..|>, so the command:

^..|>+1

will put the entry pointer so that any text entered with the E command will go at the end of the file. Future shipments of DISKATE will have a special symbol, F^, which will denote the position where text should be entered to be appended to the end of the file. This will make a command to do the job above much simpler. We'll also see other methods for doing this same thing below.

There is a subtle point here which may cause some confusion. In all of the previous examples we've stressed that | allows us to specify in essence an argument WITHIN an argument. Here we seem to have violated this. In the argument ..|>+1, the part >+1 specifies a location which is outside of the part .. to the left of the |. How can this work? The answer is that > is evaluated directly by consulting an internal DISKATE location WITHOUT ANY MATCHING. When DISKATE has evaluated the argument as far as ..|, here is what has happened, among other things:

1. The initial reference interval is still the current file.

2. < has been assigned the address of the first character in the file.

3. > has been assigned the address of the last character in the file.

After these have taken place, the value of >+1 can be calculated directly. It doesn't matter that this value lies outside the initial reference interval, any more than it matters that an explicit address like 2A00H may lie outside the initial reference interval.

There is an alternative symbol which can be substituted for .. in this example. The symbol <F> is an argument which calls for no matching and denotes the current file. The command we just saw can be replaced by the command:

^<F>|>+1

There is an important difference. .. does not really specify the current file, but rather the initial

ARG1|
ARG2

WITH
ARG2 NOT
INSIDE
ARG1

WILL
WORK IF
ARG2 HAS
NO
MATCHING

<F>

DENOTES
CURRENT
FILE

NO
MATCHING
WHATEVER
INIT REF
INTERVAL

^<F>|>+1

reference interval, as we have discussed. There is a special DISKATE command which we will discuss below which can be used to manipulate the initial reference interval, thus changing the meaning of .. -- but <F> always refers to the current file, no matter what the initial reference interval is.

SAFE COMMAND TO PUT ENTRY PTR AT END OF FILE

The command that we've just seen, ^<F>|>+1, is probably the command that comes the closest to specifying the end of the file within the DISKATE language in a straightforward way. The argument <F>|>+1 may look confusing, but it "says" the end of the file quite directly. However, there is a simpler command which will work by a side effect. We recall from a previous discussion that -1! will also specify the end of the current file. The simplest command for setting the entry pointer to the end of the file which is reliable is probably the command:

^-1! SIMPLEST COMMAND TO PUT ^ AT END OF FILE

```
^-1!
```

If you use this command in preference to ^<F>|>+1, be very sure you don't confuse ! for %.

There are simpler commands which will set the entry pointer to the end of the current file, and we'll see them later, but they require you to be certain that particular circumstances obtain and so can lead to errors. Probably the safest command to use is either ^<F>|>+1 or ^-1!.

Once again, it cannot be emphasized strongly enough that you MUST absolutely restore the entry pointer before resuming text entry if you make changes in text already entered. Failing to do this will mean the new text you enter will be going in the wrong place!!!

M COMMAND MOVES INTERVAL GIVEN AS ARGUMENT AND INSERTS PRIOR TO TARGET CHAR

But suppose the worst does happen, and you find that you have a large block of text in the wrong place. Of course, this can happen just because you change your mind about how you want the text to look even if you've made no mistakes in using DISKATE. Moving text is one of the key components of editing, and DISKATE has a simple move command. The M command, M for move, will move the characters in the interval specified by the argument and insert them the same place text entered with an E command would go: between the target character and the preceding character. Let's look at an example. Suppose the current file consists of:

```
NAME:  JOHN DOE
OCCUPATION:  DEER
SEX:  MALE
ADDRESS:  THE WOODS
```

and suppose we want to move the line with the address so

that it is the third line instead of the fourth line. The first step is to set the entry pointer to the right place. In trying to determine where to set the entry pointer, the key is always to determine which two characters you want an insertion to go between. Since we want the line with the address to become the third line, we want it to go between the second line and what is currently the third line. In other words, we want to move what is now the fourth line to the location between the first character of what is now the third line and the preceding character. This is achieved by:

```
^3_%
```

or alternately

```
^3!
```

Now to actually execute the move we use the command:

```
M4_%
```

or

```
M4!
```

The file will now look like:

```
NAME:  JOHN DOE
OCCUPATION:  DEER
ADDRESS:  THE WOODS
SEX:  MALE
```

Notice that the whole job could have been done with a single command, say:

```
^3!,M4!
```

However, while you may find yourself using such commands often when you are accustomed to DISKATE, it's always wise at first to make sure the entry pointer is in the right place before making any change. A more cautious sequence would be ^3!,' to see the position of the entry pointer, followed by M4!.

The move command is one of the most powerful in the DISKATE text editor, and there is no limit to the size of the block than can be moved. After a move command has been executed the target character will be the same character within the text as it was before, but the address of that character will be changed by the insertion. In the example above, after ^3! the target character is the S in SEX, and after the move command the target character remains the S in SEX, even though this is now in a different place. To be specific, after an M command the target character is the character after the group of characters inserted by the move.

AFTER M COMMAND

TARGET CHAR IS 1ST CHAR AFTER CHARS INSERTED

EXAMPLE OF USING M COMMAND TO RECOVER FROM POINTER IN THE WRONG PLACE

Let's look at another example of using the Move command -- one that hopefully you won't have to use. Over and over we've mentioned that to resume text entry after having made some changes, the entry pointer must somehow be restored. Several methods for doing this have been shown. But suppose the worst happens, and you forget to do this. How do you repair the text? Typically it will be with a Move command, and what follows is a "realistic" example. To make it seem natural and easy to understand, we'll assume in the example that we're creating and editing a BASIC program -- even though you may not be using DISKATE to edit BASIC. We're just at the stage of writing the body of comments that will go at the beginning of the program. Because this example will deal with several DISKATE commands in succession, in this case we will SHOW THE PROMPT CHARACTER so that the example listings will look just like they would on your terminal. Here is how the current file looks:

```
>"..
100 REM BASIC PROGRAM FOR BENCHMARKING CHARACTER HANDLING
200 REM
300 REM PROGRAM WILL ENTER INTO A STRING VARIABLE A
400 REM SEQUENCE OF RANDOM LENGTH INITIAL SEGMENTS OF
450 REM THE ALPHABET TERMINATED BY A CARRIAGE RETURN.
460 REM
500 REM ON A SECOND PASS  THE SEGMENTS WILL BE
600 REM SEPARATED AND PRINTED OUT.
>
```

We notice that there is an extra space following the word PASS in statement 500, and we want to get rid of it.

```
>^[500]%|[  ],'
500 REM ON A SECOND PASS^  THE SEGMENTS WILL BE
>K^,"..
100 REM BASIC PROGRAM FOR BENCHMARKING CHARACTER HANDLING
200 REM
300 REM PROGRAM WILL ENTER INTO A STRING VARIABLE A
400 REM SEQUENCE OF RANDOM LENGTH INITIAL SEGMENTS OF
450 REM THE ALPHABET TERMINATED BY A CARRIAGE RETURN.
460 REM
500 REM ON A SECOND PASS THE SEGMENTS WILL BE
600 REM SEPARATED AND PRINTED OUT.
>
```

Everything looks good. (Remember that in a command like K^, * serves as an interval consisting of only one character, the target character.) Forgetting to reset the entry pointer to the end of the file, we go blithely

along entering additional comments:

```
>E[700 REM
800 REM PROGRAM WRITTEN IN VIRTUAL BASIC VERSION 1.0
900 REM
]
>"..
100 REM BASIC PROGRAM FOR BENCHMARKING CHARACTER HANDLING
200 REM
300 REM PROGRAM WILL ENTER INTO A STRING VARIABLE A
400 REM SEQUENCE OF RANDOM LENGTH INITIAL SEGMENTS OF
450 REM THE ALPHABET TERMINATED BY A CARRIAGE RETURN.
460 REM
500 REM ON A SECOND PASS700 REM
800 REM PROGRAM WRITTEN IN VIRTUAL BASIC VERSION 1.0
900 REM
 THE SEGMENTS WILL BE
600 REM SEPARATED AND PRINTED OUT.
>
```

Help!! This is not the way the text was supposed to look! Now the question is how to put it back together. In this case we´re somewhat lucky because the statement numbers provide an easy way to figure out where things should go. If this were an assembler program things might be tougher. After staring at this garbaged up text for a minute we can see that the last thing we entered, statements 700, 800, and 900, are out of place. They should have gone at the end of the file, but instead have wound up in the middle of statement 500. Here is one way of correcting the damage. We´ll use the entry pointer to be able to see just where things are by means of the ´ command. Clearly the problem starts on what is now line 500.

```
>^[500]%|[700],´
500 REM ON A SECOND PASS^700 REM
>X=^
>^[900]%|>,´
900 REM^
>Y=^
>"X..Y
700 REM
800 REM PROGRAM WRITTEN IN VIRTUAL BASIC VERSION 1.0
900 REM
>^<F>|>+1
>MX..Y,"..
100 REM BASIC PROGRAM FOR BENCHMARKING CHARACTER HANDLING
200 REM
300 REM PROGRAM WILL ENTER INTO A STRING VARIABLE A
400 REM SEQUENCE OF RANDOM LENGTH INITIAL SEGMENTS OF
450 REM THE ALPHABET TERMINATED BY A CARRIAGE RETURN.
```

```
460 REM
500 REM ON A SECOND PASS THE SEGMENTS WILL BE
600 REM SEPARATED AND PRINTED OUT.
700 REM
800 REM PROGRAM WRITTEN IN VIRTUAL BASIC VERSION 1.0
900 REM
>
```

Success! Let's go over this carefully. First a word about the general method. The problem is that we have a piece of text in the wrong place, and to correct that it has to be moved with the Move command. To specify the Move command requires we determine three things: the lower address of the interval to be moved, the upper address of the interval to be moved, and the target location to which we want the interval moved. The method to be used here, which is a fairly cautious use of DISKATE, is to determine each of these separately. First the lower address of the interval to be moved. We decided that the thing which is out of place is statements 700 through 900, and statement 700 now begins in the middle of line 500. The first thing we do, then, is set the entry pointer to the beginning of where statement 700 now is, and use the ' command to make sure we've got it. The value of the pointer is then saved in X, which will be the lower address in the interval to be moved. The use of the pointer here is not absolutely necessary, but setting the pointer and then viewing its location with the ' command is so convenient as a way of seeing what we're doing that it's worth the extra bother. Remember that the ^ command sets the entry pointer to the LOWER address of the interval which is its argument.

The next step is to set the upper address of the interval to be moved, which we'll do by setting the pointer, verifying that it's in the right place, and then saving it in the variable Y. Right here is another important possible source of error. The ^ command, as well as a command of the form: variable=argument, will take the LOWER address of the argument. What we want here is the UPPER address of the interval to be moved. That's why we had to give the command ^[900]%|> instead of simply ^[900]%. ^[900]%|> sets the entry pointer to the upper address of the interval [900]%.

Finally we set the entry pointer to the place where we want the interval moved, which is the end of the file. The command for doing this we've already seen. As a final precaution, the command "X..Y will show us the interval we are about to move, so that we can have one last check that it is specified correctly. At last we are ready for the Move command itself, which is simply MX..Y since we have set everything else up.

This may seem like a lot of trouble to have gone to. In fact, the whole thing could have been accomplished by the single command:

^<F>|>+1,M[700]..[900]%

Why on earth should you go through all the work that we did when such a simple command will work?? The answer is that the simple command will work because we happen to know it is correct. When you are working on editing in actual usage, you may be just as likely to make a mistake in a command such as the one just above as you were to make the mistake which this command is supposed to correct. The simple command above gives you no feedback at all to see if the operands are specified correctly before tha actual move takes place. There is nothing more frustrating in text editing than to make a mistake in entering a command which is simply aimed at correcting a still previous mistake. Of course, the more experience you get with DISKATE, the more you will be able to use such simplified commands with no trouble.

By now we have presented all of the most basic DISKATE text editing commands. The editing commands that are presented below are on a more advanced level, and allow you to perform editing tasks repetitively and create your own "edit macros" -- programs that accomplish complex editing functions. You should probably attempt to familiarize yourself with the use of the commands discussed so far before going on to try the techniques that are going to be presented now.

Above we saw that you can replace a portion of text with something else by giving a K command followed by an E command. This is fine if you want to change a single instance, but what if you want to make the change in several places? It frequently happens in text editing that you have to change EVERY occurrence of a part of the text to something else. For instance, you might discover that you had systematically misspelled a word every time you used it, or in editing a program you might be forced to change the name of a variable, which would mean changing it everywhere it occurred. DISKATE provides very powerful commands for performing this kind of task. Of course, this power if used erroneously can leave you with a monster job of cleaning up, so great care must be used to make sure these powerful commands do not themselves contain errors.

Let's start with a simple example. Suppose that we have a file which is a list of entries each one having the same format as our dog-eared JOAN DOE file. After

numerous complaints from the clientele we are pursuaded that it isn't good form at all to keep maintaining a line in each entry for the sex of the creature described. This is the seventies, after all. So, every line in the entire file containing the pattern [SEX:] must go. Before showing how it all can be done with a single command, let's see how we would get rid of these lines one at a time. The first such line in the file could be eliminated by the command:

```
K[SEX:]%
```

Now what about the next such line? Remember that a pattern matching argument such as [SEX:]% will call for a search through the entire initial reference interval, in this case the current file, starting at the beginning. We have already axed what was the first occurrence of [SEX:]%, so the pattern [SEX:]% will NOW match what originally was the SECOND ocurrence of [SEX:]%, since we deleted the original first occurrence. This means that to kill the next occurrence of the pattern we can give the identical command K[SEX:]%, and keep on giving the same command any number of times to kill off the offending lines one by one. Suppose for the moment that we happen to know that the file has 47 entries, so that the command will have to be given 47 times. It's obviously a ridiculous nuisance to have to actually type in the command this many times. Fortunately there is a DISKATE command which will cause other commands to be REPEATED a given number of times. This is the R command. In a multi-command command line, the sequence:

```
Rn,othercommands
```

will cause othercommands to be repeated n times. We are now able to purge our notorious file of its baser instincts by the single command:

```
R47,K[SEX:]%
```

R COMMAND

Rn

CAUSES REST OF COMMAND LINE TO BE REPEATED n TIMES

A word of warning. In giving such commands as this you have to be extremely careful that a pattern in the part of the command to be repeated does not match more than you intend. Suppose for instance that instead of using the pattern [SEX:]% we had used [SEX]%. This pattern would also match a line containing the name THOMAS WESSEX, for instance, which is not one of the lines we want to delete. "Boundary characters", such as spaces, punctuation and carriage returns, are often very important in specifying patterns that match the right thing.

An obvious question here is what to do when you don't know how many times the command will have to be repeated. Actually, you can program DISKATE to count the number of times a pattern occurs, and we'll see below how to do that. But we don't really need to go to all that work. The simplest approach in giving a repeat command which you want to apply to every occurrence of a pattern is to give a repetition number which you can be sure is greater than the number of times the pattern will occur. If we know for instance that there are not more than a few hundred entries in the file but don't know exactly how many, we could use the command:

```
R9999,K[SEX:]%
```

Assuming we're correct that there are not nearly this many occurrences of the pattern [SEX:]% in the file, the command will be repeated untill all of the occurrences of the pattern have been deleted. Then what? The pattern [SEX:]% will fail to match anything, so an error message will occur.

Though this method will work perfectly well, the obvious problem is that we have to assume that the error message is given because DISKATE ran out of occurrences of the pattern to match, rather than because of some error that should really concern us. It's better not to have to rely on an error to terminate a process, so we'll see in a minute how to avoid this. One note. Repetition numbers are treated as POSITIVE. That means that -1 is the same thing as 65535. So, if you want to use the "large number" method for the R command, the simplest number to use for the number of repetitions is -1, since this is the largest repetition number DISKATE will allow. (65535 is obviously large enough for anything you would want to do, since files must fit in the memory and 65K is the limit to the size of the memory.)

IN Rn COMMAND, n IS TREATED AS POSITIVE

R-1 IS THE SAME AS R65535

The problem we just encountered can be phrased as follows: How can we break a repetition loop when we've run out of matches without terminating the loop with an error? There is a DISKATE command especially for this purpose. The command QF, which stands for Quit on Failure, will terminate a repeat loop without an error if in evaluating the argument of the QF command a match fails. So, we can do what we did above without having to end in an error by the command:

QF COMMAND TERMINATES LOOP WITHOUT ERROR IF ARGUMENT GIVES MATCHING FAILURE

```
R-1,QF[SEX:]%,K[SEX:]%
```

The R-1 will cause the rest of the command to be repeated 65535 times, and the QF[SEX:]% command will terminate the loop without an error if there is nothing which matches [SEX:]%. In fact we can make the command much

simpler. Recall that if a command takes an argument but none is given, DISKATE uses the last argument computed. In the command line above the argument of the QF command is identical to the argument of the K command, so the command can be simplified to:

DELETE EVERY OCCUR-RENCE OF A PATTERN

```
R-1,QF[SEX:]%,K
```

This is an example, then, of a general form of command which will delete every occurrence of a pattern in a file. In general you would use the command:

```
R-1,QFpattern,K
```

to delete from the current file every occurrence of pattern. Suppose we want to REPLACE EVERY OCCURRENCE of one pattern by a piece of text. This can be done exactly the same way, remembering that a K command followed by an E command replaces the argument of the K command with the entered text. Thus the command:

REPLACE EVERY OCCUR-RENCE OF A PATTERN WITH TEXT

```
R-1,QFpattern,K,E[text]
```

will replace every occurrence of pattern with text. (In word processing systems this operation is often called "global search and replace".)

In determining whether a matching failure has taken place for an argument of a QF command, even if there are no matching symbols in the argument, a "matching failure" -- perhaps argument failure is a better term -- will still occur if the lower address of an interval turns out to be larger than the upper address. This can acutally be used for comparison operations. We'll see an example of this in I.2.

In effect a command line using the R command constitutes a miniature program. Becuase it so often happens that you will want to execute a programmed sequence of DISKATE commands, DISKATE provides the facility to execute a sequence of commands that are stored in the memory as all or part of a file. Our next major goal is to discuss this facility, but before we do it's time to present the commands that enable you to use multiple files. Often you may only work with a single source file at a time, but when entering a sequence of commands which is to be executed by DISKATE, more than likely they will want to go into a file separate from the material they operate on.

As we mentioned at the beginning of the manual, a file is a sequence of bytes bounded by Ø's. The current file is included in the source area, which may contain several files. Suppose you want to begin editing a new file, while retaining the current file in the source

N COMMAND CREATES NEW

area for use later. The command to be used is the N command, (N for New file,) which takes no argument. This command creates a new empty current file at the end of the source area, and expands the source area to include it. Any editing you do on this new current file will leave the old file intact. (That's assuming, of course, that in editing the new file you don't make an error in an argument causing it to specify an address outside of the current file.) If you want to edit several files, the N command can be given any number of times, as long as you don't run out of memory.

EMPTY FILE AT TOP OF SOURCE AREA & MAKES NEW FILE CURRENT

By using the N command you can create several files in the memory. Now the question is, suppose you want to go back and look at one of the previous files? To do this you use the F command, (F for File.) The F command is different from all of the commands we have described so far in that THE INITIAL REFERENCE INTERVAL FOR THE F COMMAND IS THE WHOLE SOURCE AREA. The actual operation of the F command is somewhat detailed, but the basic idea is that it TAKES THE LOWER ADDRESS of the argument given and MAKES THE FILE CONTAINING THAT ADDRESS CURRENT. Remember that a file in the memory is bounded by Ø's. When DISKATE is given an F command it evaluates the argument and uses only the lower address. First it looks to see if this location contains a Ø -- if so it assumes that this is the beginning location of the file. If not, it begins searching backward from this location looking for a Ø, and when it finds it, sets this as the beginning of the file. Having found the beginning of the file, DISKATE searches forward from the argument location looking for another Ø which will be the end of the file. When it has determined the file boundaries, the file is made current and THE TARGET CHARACTER IS SET TO BE THE Ø WHICH ENDS THE FILE.

F COMMAND

SETS CURRENT FILE TO THE FILE CONTAIN-ING THE LOWER ADDRESS OF ARG

INITIAL REF INTERVAL IS WHOLE SOURCE AREA

TARGET CHAR IS END OF FILE

The F command also ADJUSTS THE SOURCE AREA. If the file boundaries determined place the file entirely within the current source area, the source area stays unchanged. If the file boundaries place the file entirely outside of the current source area, the source area is redefined to be the file determined by the F command. Nothing happens to the old source area in the memory -- the only thing that is changed is that the internal pointers DISKATE uses to define the limits of the source area are reset. If one of the file boundaries lies inside the current source area but the other doesn't, the source area is expanded to include the file determined by the F command.

SOURCE AREA ADJUSTED TO INCLUDE THE FILE

Let's see how the F command would work with a simple example. Let's suppose we're using DISKATE to edit a letter to George Morrow expressing how much we like his memory boards. Somewhere in the first few lines will be a line containing the pattern [George].

Now suppose we use the N command to set up a new empty file, and in this file we put a string of DISKATE commands used to format the letter. We´ll see below some examples of such "edit macros". Before we execute these we want to look over the file with the letter one more time to make sure it is O.K. At this point the source area consists of two files: the file with the letter, followed by the file with the DISKATE commands. The file with the commands is current. The command:

F[George]

will make the file with the letter current, and set the target character to the end of the file. Remember that the initial reference interval for the F command is the whole source area, so in executing the F command DISKATE will begin searching for the pattern [George] at the beginning of the source area, which is the file with the letter.

Suppose we had written three letters, one each to Tom, Dick, and Harry, and following that had a file with commands. If we want to see the file with the letter to Harry we can use the command:

F[Dear Harry]

(It might not be a good idea to use the command F[Harry], since the letter to Tom might talk about Harry.)

There is also a simple way you can "page through" the files without knowing what any of their contents are. Suppose you want the current file to be the first file in the source area, whatever that is. The command:

F<S>

<S>

ARGUMENT DENOTING SOURCE AREA

(NO MATCHING)

will do the trick. <S> is a symbol we haven´t seen before. It is analogous to <F> and denotes the source area. It does not call for matching. Since the F command only uses the lower address of its argument, F<S> will make the first file in the source area current.

Now suppose we want to see the next file in the source area. As usual with DISKATE there are many ways of doing this. We saw above that the command ^<F>|>+1 will set the entry pointer so that the target character is the Ø at the end of the file. That means that the argument <F>|>+1 will specify the Ø at the end of current file. But, if another file follows the current file in the source area, this Ø will also be the Ø which forms the lower bound of the file which follows. So, if we give the command:

F<F>|>+1

it will set the current file to be the file following what had been the current file. (Don't be confused about "F" appearing in both the command and the argument -- this is just the same as the principle behind a statement like X=X+1.) Or, we could just as well have given the command:

F-1!

There is just one problem with this method. Suppose you just don't remember which is the last file in the source area, so that you aren't able to recognize the last file when you see it. The command F<F>|>+1 calls for no matching, and neither does F-1!, so that if the current file happens to be the last one in the source area and you execute this command anyway, it may not return an error message, but instead will give you a file consisting of garbage. This could give you a significant surprise! What we would like is a method of setting the current file to be the next file in the source area but which would give an error message if we happened to already be at the last file. Here is a command that will do the trick:

COMMAND TO MAKE NEXT FILE IN SOURCE AREA CURRENT

^<F>|>+1,F^..|2@

Let's see how this works. ^<F>|>+1 we have already seen as a command which will set the pointer to the end of the file. This may not be necessary if you have done no editing on the current file, but we'll include it to be safe. Now let's look at the argument in the second command on the command line above. Becuase it is the argument of an F command, the initial reference interval is the entire source area, so ^.. specifies everything in the source area following and including the Ø at the end of the current file. ^..|2@ specifies the second character within the interval we just mentioned. If there is a file following the current file, then, the argument ^..|2@ will give the first character of that file. (^..|@ would give the Ø forming the lower bound of the file.) However, if there is no such file then after ^<F>|>+1, ^ will point to the highest address in the source area, so the argument ^..|2@ will produce a matching failure and hence an error. Thus the F^..|2@ will give an error if there is no file following the current file, or else if there is such a file will make it current. Of course, the command:

^-1!,F^..|2@

would work just as well.

At this point a digression is in order to discuss one of the "side effects" of the F command. Above we gave a great deal of attention to the problem of having as close to a foolproof method as possible for setting the entry pointer to the end of the file after corrections have been made on entered text. We also saw an example of how unpleasant it can be to forget to do this. We've also mentioned that the F command will position the entry pointer to the end of the file that it makes current. Perhaps this is a clue to how we might be able to have a shorter command which will set the entry pointer to the end of the current file. One possibility, which is certainly a short command, is:

```
F^
```

While this will work in many cases, it is somewhat dangerous to rely on it habitually. The reason is that there is no guarantee that the target character is actually within the current file. In all of the examples we have seen so far, the target character did stay within the current file, but in those cases where you are using DISKATE as a monitor, which is discussed in the next section, you will frequently execute commands which place the target character outside of the source area altogether.

A much safer alternative is the command:

```
F<F>
```

This command says, take the current file and make it current. A bit redundant, of course, but it will position the entry pointer reliably to the end of the current file, and is shorter and simpler than the command ^<F>|>+1. It's more straightforward than ^-1!, but you can choose whichever is easier to remember.

F<F>

ANOTHER SIMPLE COMMAND TO PUT ENTRY POINTER AT END OF CURRENT FILE

Now we can get to the real goal of the present discussion: how to write sequences of DISKATE commands which can be executed as "edit macros". The flexibility which this feature of DISKATE allows provides for a virtually unlimited range of applications. We'll begin with examples which are very simple. One simple application for edit macros which could arise very often is substituting for an abbreviation the text the abbreviation stands for. If there is a long phrase that occurs over and over in your text, for instance, it would be nice to have an abbreviation for it and then use the DISKATE edit macro facility to replace the abbreviations with the expanded form. To use an example that occurs frequently in this manual, suppose we use the symbol I.R.I. to stand for initial reference interval. Without

using any edit macros, we could instruct DISKATE to replace every occurrence of the I.R.I. by the full text using the command:

```
R-1,QF[I.R.I.],K,E[initial reference interval]
```

If we found ourselves wanting to issue this command frequently, it's a long one to keep having to type in every time. Let's see how to set it up as a macro, so that it can be executed with a very short command line. First we have to enter it into the memory. We can do this with the commands:

```
N,E[R-1,QF[I.R.I.],K,E[initial reference interval]]
F<S>
```

For the moment we'll assume that there will be only one file with the text, and then a second file with the macro. The F<S> command will restore the text file as the current file. Now the question is, how do we execute the macro? Macros are executed using the DISKATE command D, which stands for Do. Like the F command, THE INITIAL REFERENCE INTERVAL FOR THE D COMMAND IS THE ENTIRE SOURCE AREA, and the D command only uses the lower address of its argument. To execute the macro all we have to do is figure out an argument which will specify where the macro is in the source area and give a D command with this argument.

The current file is the text file on which the macro will operate. <F> is an argument which will specify this file, and the upper address in <F> is the last character of the file. <F>|>+1 would specify the Ø forming the upper bound of the file, so since the macro immediately follows the text file, <F>|>+2 will specify the first character of the macro. So, every time we want to execute the macro we can type the command:

```
D<F>|>+2
```

There is something very unsatisfying about this. The command D<F>|>+2 is one that is difficult to remember and it would be easy to make a mistake in typing it. It would be better if we could give the macro a name and then execute it by calling it by name. DISKATE has just such a provision. To name a macro you use the * command. The * command does nothing and is ignored. This means that the characters following the * in a * command can be used as a label. Let's see how this would work. Let's say we want to give our macro the name IRI. The macro originally looked like:

```
R-1,QF[I.R.I.],K,E[initial reference interval]
```

D COMMAND EXECUTES STRING OF DISKATE COMMANDS STARTING WITH LOWER ADDRESS OF ARG

INITIAL REF INTERVAL IS SOURCE AREA

* COMMAND DOES NOTHING, IS IGNORED

CAN BE USED AS LABEL TO NAME MACROS

To add the name to the macro it should look like:

```
*IRI
R-1,QF[I.R.I.],K,E[initial reference interval]
```

When the macro is executed the command *IRI will simply be ignored. Now the question is, how do we execute the macro by name? Remember that the initial reference interval for the D command is the whole source area. The command:

COMMAND
D[*name]

```
D[*IRI]
```

EXECUTES MACRO WHICH BEGINS WITH *name

will take as its argument the beginning address of the first occurrence within the whole source area of the characters *IRI. Assuming the text file does not contain such a pattern, since the initial reference interval is the whole source area this argument will evaluate as exactly the beginning of our macro. Of course, in giving a macro a name in this way you have to be careful that the pattern *macname -- where macname is the name of the macro -- does not occur anywhere in the source area ahead of the beginning of the macro. Note that using this method of naming a macro using the * command and executing it by D[*macname], you can have as many macros as you want, and they can all be in a single file after your text file(s) or they can be in several separate files. It doesn't matter where they are within the source area, as long as the name is properly unique.

The commands within a macro end in a carriage return, just as if they were typed in a command line, and of course there can be several on a line separated by commas. When a macro is being executed, when the end of the file is reached this is automatically treated as a return. If the macro was invoked from the terminal by a command line, control will be returned to the terminal. However, a macro can contain a D command, so that macros can call other macros. In this case when the macro returns, control is passed to the next command in the calling macro. Thus the D command works like a GOSUB in BASIC. We saw above that the command QF can be used to break out of a repeat loop. It can also be used in a macro to return the to caller. QF will quit whatever is the innermost process in which it occurs, whether this is a repeat loop or a macro, if its argument produces a matching failure.

There is a special argument that can be used in connection with macros. As a macro is executed, a special "command interpretation pointer" is maintained internally by DISKATE. If the macro aborts in an error, the value of this pointer is saved in a special loca-

?

AS ARG SYMBOL

tion. The contents of this location are denoted by the symbol ?, which does not call for matching. In other words, ? is an argument which gives the address of the character of a macro that caused an error. You can use this to help debug a macro which produces errors. For instance, if we had a macro in the source area which began with the command *IRI, and this macro was producing errors, we could find the point where the error occurred by the command:

GIVES CHAR OF MACRO THAT CAUSED AN ERROR

```
"<S>|[*IRI]..?
```

This would print the macro on the terminal up to the point that caused the error.

When the R command was introduced we said that it would cause the rest of the command to be repeated. This is correct if the command is entered from the terminal, but if an R command occurs as part of a macro it will cause THE ENTIRE REST OF THE MACRO to be repeated until either an end of file or a quit command is encountered.

R COMMAND IN MACRO REPEATS TO END OF FILE OR QS,

Analagous to the QF command is the QS command, which stands for Quit on Success. If its argument does not produce a matching failure it will quit the innermost process be it a repeat loop or a macro. The QS command can be used in repeat loops in commands entered from the terminal, just like the QF command. If the QS command is given with no argument, the argument used will be the previous interval computed, as with any other DISKATE command. This argument can be presumed not to have given a matching failure, so that QS WITH NO ARGUMENT CAN BE USED AS A RETURN COMMAND. This is especially useful if you want to have several macros in a single file.

QF CAUSING QUIT QS COMMAND QUITS INNER-MOST PROCESS IF NO MATCHING FAILURE IN ARGUMENT QS WITH NO ARG ACTS AS RETURN

Let's put some of these pieces together and look at a macro which performs a function which occurs frequently in text editing: formatting paragraphs. After making editing changes in prose text it will often happen that the margins of the lines will have to be readjusted. The macro we will look at will have three parameters, which are given by setting the values for the variables X, Y, and Z. Z will give the number of characters that we want as the maximum line length, and X..Y will serve as an interval forming the boundaries of the paragraph. Once X, Y, and Z are set we want the command D[*PARAGRAPH] to adjust the right-hand margins.

The macro will work in two stages. First we will delete all the carriage returns in the paragraph, replacing them with blanks. This will in effect turn the paragraph into one enormous line. The second stage will chop it up into lines of the proper length. Each of the two stages will have its own "submacro", which

G COMMAND TRANS-

PARAGRAPH will call. The macro will contain a new command, the G command which stands for Goto. The G command causes execution of a DISKATE macro to be transferred to the lower address of the argument, and works just like a GOTO in BASIC. Like the D command, the initial reference interval for the G command is the whole source area. And now for the macro:

FERS EXECU- TION OF MACRO TO LOWER ADDRESS OF ARG

INITIAL REF INTERVAL IS SOURCE AREA

```
*FUSE
QFX..Y|_,K,E[ ],G[*FUSE]
*CHOP
QFX..Y|^..Z@,K|-1[ ],E_,G[*CHOP]
*PARAGRAPH
D[*FUSE]
^X
D[*CHOP]
QS
```

A few notes. The final QS can be done away with if this macro ends the file. If as shown the entire macro is going to go in one file, the lines beginning with *PARAGRAPH must NOT begin the file. The reason is that if they do, the pattern [*FUSE] will match not the name of the submacro, as we want it to, but rather the characters *FUSE within the D[*FUSE]. This could cause all kinds of problems. Note that in the macro CHOP, the K followed by the E command will leave the target character as the character following the carriage return just entered, setting it properly for the next time around the G loop.

There is one difficulty with this method: invoking a macro or using G commands with an argument like [*name] will work perfectly well but could end up being extremely slow, since each time the command is executed DISKATE has to search through the whole source area looking for the argument. One way around this is to use variables. For instance, the above example could also be done this way:

```
*FUSE
QFX..Y|_,K,E[ ],G FUSE
*CHOP
QFX..Y|^..Z@,K|-1[ ],E_,G CHOP
*PARAGRAPH
FUSE=<S>|[*FUSE]
D FUSE
CHOP=<S>|[*CHOP]
^X
D CHOP
QS
```

Here the macro assigns the DISKATE variables FUSE and CHOP to the respective arguments. This way the G com-

mands do not call for any matching, and so execute much faster. Note that whereas a D command could be given simply in the form D[*FUSE], in the = statement above the form <S>|[*FUSE] had to be given. The reason is that the D command is one of the few commands for which the initial reference interval is the entire source area, so the prefix <S>| was not needed. For the = statement the initial reference interval is the current file, so the source area must be explicitly set as the initial reference interval.

There is one VERY IMPORTANT CAUTION when using the G command with an argument which is a variable rather than a pattern matching argument. If the macro is located inside the source area, and follows the text to be edited, editing commands within the macro can cause the size of the text to change, and thus cause the entire macro to be moved about in the memory as the source area behind the change is shifted up or down. In this case, the value of the variables used as arguments of G commands will NOT be updated as the source area is shifted. It just happens that in the macro above, no net change in the length of the text is caused, so this problem does not occur. One way of preventing this difficulty is to locate edit macros outside of the source area completely. This can be done, for instance, by editing the macros in the source area, storing them on disk for future use, then loading them into a vacant part of the memory which will not be overwritten by the source area. Or, as we'll see below, it's possible to have more than one source area, though of course only one at a time can be current. Another way of getting around this problem is to put the macros at the BEGINNING OF THE SOURCE AREA. That way, as changes are made within the text, addresses within the macros will not change. Of course, when using only pattern matching arguments the problem does not occur at all.

EDIT MACROS WHICH CHANGE THE LENGTH OF TEXT MUST NOT FOLLOW TEXT IN SOURCE AREA

CAN BE AHEAD OF TEXT OR OUTSIDE SOURCE AREA

An alternative method to using G commands would be to use the R command, but this would require that FUSE, CHOP and PARAGRAPH be in separate files, since the R command will repeat to the end of the entire file until a QF or QS causes a quit. To enter the macros using this method, you could use the following sequence of commands:

```
N
E[*FUSE
R-1,QFX..Y|_,K,E[ ]
],N
E[*CHOP
R-1,QFX..Y|^..Z@,K|-1[ ],E_
],N
E[*PARAGRAPH
```

```
D[*FUSE]
^X
D[*CHOP]
]
```

A special command which can be used in macros or command lines is the PAUSE command. The PAUSE command, which has no argument, will cause the process in which it occurs to stop in a "panic detect" state. You can stop to examine the printout on the terminal, and then resume execution of the process by typing any character other than S or ESC. The "panic detect" is discussed more fully in Section I.3. The message PAUSE is printed on the terminal to notify you that you're in the pause state. This command is especially useful for such things as changing diskettes.

PAUSE COMMAND

STOPS PROCESS IN PANIC DETECT STATE

We conclude the discussion of using DISKATE as a text editor with two special commands which can be used as a convenience to save extra typing. The command DEF takes as its argument a string of characters up to 24 in length enclosed in brackets, as you would for an E command. DEF stands for Default. When DISKATE is given a DEF command, the string forming its argument is saved in a special internal location. Thereafter, whenever DISKATE expects a command, if you simply type a carriage return with an empty command line, the string given as the argument to the DEF command will be executed as a default command. For instance, above we had examples of using the same command repetitively to see successive intervals of text, such as ">..[JOAN]|2@.. By giving the command:

DEF[cmd]

CAUSES cmd TO BE EXECUTED FOR CAR RET ONLY AS COMMAND

```
DEF[">..[JOAN]|2@..]
```

DEF WITHOUT ARG SHOWS cmd

whenever you typed a carriage return in response to the prompt character, DISKATE would execute the command ">..[JOAN]|2@..

If you don't remember what the default command is, typing DEF with no argument will show it to you. If you have given a default command and then want to stop using this feature, the command:

DEF[]

```
DEF[]
```

DISCONTINUES USE OF cmd

(DEF with an empty argument in brackets) will discontinue using the default command feature until another DEF command is given.

A similar command to DEF is REF. This command also takes as argument a character string up to 24 characters in length enclosed in brackets. This string is treated as a DISKATE argument and is evaluated prior to any

REF[arg]

CAUSES INITIAL

subsequent command which normally takes the current file as the initial reference interval. This argument then becomes the initial reference interval for every such command. Those commands which use the whole source area for their initial reference interval, i.e. G, D, or F, are not affected, and a missing argument, <, and > also are not affected. The best way to see how this works is by an example. Suppose you give the command:

REF INTERVAL FOR ALL COMMANDS NORMALLY TAKING IT AS <F> TO BE arg

```
REF[^%]
```

Normally the argument .. by itself would match the current file, but after the REF command it would match the line with the pointer. Note that the argument of the REF command is kept as a character string and reevaluated each time a command would normally take the current file as its initial reference interval. With the command above, the initial reference interval changes every time the entry pointer is moved to a new line. As with DEF, REF with no argument shows the string that is being saved as the argument of the last REF command executed, and REF[] discontinues the use of this feature until the next REF command.

REF NO ARG SHOWS LAST ARG GIVEN

REF[] DISCONTINUES

REF and DEF can be used in combination to work quickly through blocks of text. For instance, suppose you are editing a letter in which paragraphs are separated by two carriage returns in a row. The commands:

```
REF[^..__]
DEF[^..|>+1]
```

can be used together to edit paragraph by paragraph. You would begin by giving the command ^<F>. This sets the pointer at the beginning of the file. Now the symbol .. matches the CURRENT PARAGRAPH. To get to the next paragraph simply type carriage return in response to the prompt character. Of course, if you make some changes, resetting the entry pointer, .. will then match the portion of the current paragraph from the entry pointer onwards. You must be very careful when using this editing method, since the REF command CHANGES THE MEANING of the symbol .. In fact virtually every matching argument may have a different meaning if the REF command feature is being used.

NOTE: REF CHANGES THE MEANING OF .. AND OTHER MATCHING SYMBOLS

In exactly the same way, the pair of commands:

```
REF[^%]
DEF[^_+1,"..]
```

can be used to edit line by line. The REF command establishes the "current line" -- i.e. the line containing the entry pointer, as the initial reference inter-

COMMANDS TO EDIT

val. By giving the default command, (carriage return with an empty command line,) the entry pointer will be set to the next line and that line will be printed on the terminal. Note that when giving the default command, the part of the default command ^_+1 sets the entry pointer to the address of the byte beyond the first carriage return in the initial reference interval, which because of the REF command is the line containing the entry pointer, rather than the whole source area.

LINE BY LINE

## I.2 Using DISKATE as a Monitor

As mentioned in the introduction, there is considerable overlap between this section and the previous one. Most of the commands we will discuss have already been introduced. In using DISKATE as a text editor, the commands you give will usually apply to the current file within the source area, or sometimes the whole source area. In using DISKATE as a monitor, the commands you give may apply anywhere in the computer's memory. Some of these commands WILL FUNCTION DIFFERENTLY IF THE TARGET IS OUTSIDE THE SOURCE AREA. Naturally all such differences will be pointed out.

CURRENT BASE

In addition to using ASCII codes, when using DISKATE as a monitor you will also want to deal with the codes directly, probably in hexadecimal or octal. When you bring up DISKATE there will be an initially established CURRENT BASE. The current base represents the base in which numbers from DISKATE are OUTPUT. When you input numbers to DISKATE, THE BASE MUST ALWAYS BE GIVEN EXPLICITLY. If no base is specified, DISKATE assumes that the base of any number typed in is DECIMAL. There are several options for specifying numbers to DISKATE. First of all, a number may either be SPLIT or not. A number in split form is specified by giving two numbers separated only by a colon. The first number gives the value of the high order byte of a 16 bit value, and the second number gives the value of the low order byte.

IS BASE FOR OUTPUT OF NUMBERS

FOR NUMBER INPUT BASE MUST BE GIVEN -- DEFAULT IS DECIMAL

For instance, suppose you are reserving the first "page" -- i.e. the first 256 byte segment in the computer's memory, for special drivers or interrupt handling or the like. The first address that you might have a program occupy, then, could be specified as:

```
1:Ø
```

n:m

This means the number = 1*256 + Ø. Note that since no base was specified, DISKATE assumes decimal. Of course numbers in hexadecimal do not need to be split, since the hexadecimal form is "naturally" split already. Numbers in hex are indicated by adding the character "H" as a suffix to the number. The number 1:Ø is equivalent to:

GIVES n*256+m (SPLIT FORM)

```
1ØØH
```

in hex.

SUFFIX H FOR HEX,

To specify a number in octal, or base 8, the suffix "Q" is added to the end of the number. So, for example the numbers:

Q FOR

OCTAL

```
2000H
40:000Q
32:0
```

HEX NUMBER MUST BEGIN WITH DECIMAL DIGIT

all specify the same number, which is 32*256. There is one important point about hex numbers. A HEXADECIMAL NUMBER MUST BEGIN WITH A DECIMAL DIGIT. Otherwise there would be difficulty distinguishing it from a variable name. Thus FFFFH is not a valid way to specify a number for DISKATE -- you would have to use 0FFFFH.

The current base can be changed at any time by using the B command. For instance, to change from hexadecimal to octal you would use the command:

B COMMAND

Bn SETS n AS CURRENT BASE

```
B8
```

When the current base is not hexadecimal, outputs of numeric codes from DISKATE will always be in split form.

As mentioned briefly in the previous section, numbers can be used to specify an interval explicitly. For instance, the first 8K bytes of memory can be specified by the intervals:

```
0..1FFFH
0..37:377Q
0..31:255
```

# COMMAND

DUMPS INTERVAL TO TERMINAL BYTE BY BYTE IN CURRENT BASE

To view an interval in memory in numeric, rather than ASCII form, the # command may be given. It works like the " command, except that interval given as the argument is output byte by byte in numeric form in the current base. For each line of output there will also be a field at the left of the printout showing the address of the first byte being printed on that line. The # command will thus perform a CORE DUMP to your terminal of the interval given as the argument.

Arguments given with the # command may call for matching, just like so many of the ones we saw in the previous section. The # command is especially useful for viewing non-printing control characters if these must be used. Suppose, for instance, you suspect that there is a non-printing character in the line in the current file containing the target character. The command:

```
#^%
```

will dump this line on the terminal, and you can inspect the codes to try to find out where there may be a trouble spot.

E#codes#

You can also use the E command to enter numeric

ENTERS

codes as well as ASCII characters. When entering text using the E command, the ASCII characters comprising the text are surrounded by brackets. To enter numeric codes using the E command, the codes are surrounded by the character "#" and separated by blanks. In this case the base must NOT be given -- the base is automatically assumed to be the current base. A string of codes surrounded by the # symbol can also be concatenated with other valid arguments of an E command to form a composite argument. The E command is one of those that works differently if the target character is outside of the source area. When the E command was introduced it was explained that it will INSERT the entered text into the current file between the target character and the previous character. However, if an E command is given and THE TARGET CHARACTER IS OUTSIDE THE SOURCE AREA, the bytes entered WILL OVERWRITE whatever is there and there will be no insertion.

NUMERIC CODES

CODES MUST BE IN CURRENT BASE WITHOUT Q OR H SUFFIX SEPARATED BY BLANKS

Let's look at an example. Suppose location 200DH contains a jump to an output routine for your terminal, but you want to change the jump to a jump to location 307AH. First we can examine the place we want to change using the # command:

IF ^ OUTSIDE SOURCE AREA

```
#200DH..<+2
```

E COMMAND WILL OVERWRITE NOT INSERT

This will show us three bytes beginning with 200DH. (We want to look at 3 bytes because the jump instruction is three bytes long.) We'll assume that the current base is 16. In response to the # command, DISKATE would print on the terminal something like:

```
200D    C3 03 29
```

The C3 is the opcode for the JMP instruction, and the jump is to location 2903H. (Remember that 8080 instructions put the low order byte of an address at the low order location.) The C3 we want to leave alone, but the two bytes 03 29 we want to change to 7A 30. Just as we did when entering text, to use the E command to enter numbers we first have to set the entry pointer. The command to do the whole job is:

```
^200EH,E#7A 30#
```

Note that we say ^200EH instead of ^200DH because at 200DH is the opcode C3 which we don't want to change.

Obviously it should go without saying that if you use DISKATE as a monitor as in this example, and then go back to editing a source file having failed to restore the entry pointer to the right place back inside the source area, a total disaster could result. Entering

text following a monitor usage such as we just saw without having restored the entry pointer could very well dump "garbage" on top of part of the resident software, which could certainly result not in merely garbaging up your source file but in a total crash which could wipe out the contents of the entire memory!!! Once again, always be sure the entry pointer is in the right place before beginning to edit.

You can use the E command with numeric codes rather than text in editing source files also. Once upon a time we promised to show a method for entering into a text file unbalanced brackets. This can be done by using an E command with the numeric code for the bracket rather than using the E command with ASCII text. The command:

```
E#133#    (if the current base is 8)
E#5B#     (if the current base is 16)
```

for instance will enter a left bracket.

#codes#

ARGUMENT THAT MATCHES SEQUENCE OF CHARS WITH VALUES MATCHING codes

You can also use a sequence of numeric codes enclosed in #'s, with each number separated by a blank, as part of arguments calling for matching. The pattern:

```
#numeric code sequence#
```

where numeric code sequence is a sequence of numeric codes in the current base separated by blanks, will match a sequence of characters having the same value in the current base as given codes. There are many ways this can be used. Suppose for instance you have a piece of text which was created using another piece of software and each line concludes with a carriage return followed by a line feed. To delete the line feeds you can use the command:

```
B16,R-1,QF#D A#|>,K
```

Here the pattern #D A# matches the characters carriage return, line feed since we made sure the current base was 16. Thus #D A#|> matches a line feed at the end of a line, and this command will get rid of all of them. Of course, such commands can be easily turned into edit macros if you are going to use them frequently.

Using a matching argument of the form #codes# together with the # command can be a powerful way to use the DISKATE monitor. Suppose for instance there is a jump somewhere between 2000H and 2100H with the following hex code:

```
C3 03 29
```

and we want to patch this so the Ø3 29 is changed to 7A 3Ø. However, we don´t remember just where this jump is located. Let´s assume that the current base is 16. The command:

```
#2ØØØH..21ØØH|#C3 Ø3 29#
```

will print on the screen the address where the jump is located, and then the codes. The patch can then be put in and we can see the result with the command:

```
^<+1,E#7A 3Ø#,#
```

(Note that the last # in the command line above is a # command using the default argument.)

One of the main differences between using DISKATE as a monitor and using it as an editor is that in using it as a monitor you will tend more to use numeric codes and explicit addresses. In this regard it is important to be able to know the explicit address for constructs that we specified before by matching and such symbols as <F>, <, etc. The DISKATE ? command will print on the terminal in the current base the lower and upper addresses of the interval given. For instance, if you want to know where the source area is in the memory, type the command:

```
?<S>
```

? COMMAND

PRINTS UPPER AND LOWER ADDRESS OF INTERVAL GIVEN AS ARGUMENT

Likewise ?<F> will give you the location of the current file. It is strongly recommended that while you are editing you give such commands from time to time to make sure the source area is not getting too big. There is no built-in protection to prevent the source area from overflowing the actual amount of RAM you have, but by giving a ?<S> command from time to time you can find out how big the source area is getting and save your file on the disk before things get out of hand. If you determine that a file is in danger of getting too big, you can save part of it on the disk, kill that part and then go on editing the rest. The commands for working with the disk are discussed in the next section.

The ? command can also be used to find out the value of a variable. Remember that a single address is interpreted by DISKATE as an interval where both addresses are the same. So for instance if X is a variable, the command:

```
?X
```

will print on the terminal the value of X twice, since it interprets X as an interval where both addresses

equal the value of X.

COUNTING NUMBER OF TIMES A PATTERN OCCURS

Note that this can be used to count the number of occurrences of a pattern in a text file. Suppose for instance, that we are writing a piece of prose and we get the sneaking suspicion that we are using the word "however" too often. Let's say the current base is 16. The commands:

```
B10,X=0,^<F>,R-1,QF^..|[however],^>,X=X+1
?X,B16
```

will restore the current base after printing on the terminal (in split decimal) the number of times the word however occurs in the current file.

O COMMAND ORIGINATES NEW SOURCE AREA AT INTERVAL GIVEN

Of course, not only will you need to know where various things are in the memory, you will need to be able to put things where you want them. When you bring up DISKATE it will have an initial value for where the source area is to go in the computer's memory. Suppose you want the source area somewhere else? This can be achieved with the O command, which stands for Originate new source area. When the O command is given with an argument, the interval comprising the argument is established as the source area. A 0 is written at the upper and lower address of the interval to form the boundaries of the source area, unless the upper and lower addresses are equal. In this case two consecutive 0's are written at this address to establish an empty source area. Nothing is changed in the old source area unless the argument of the O command overlaps it, in which case the 0's written will overwrite something there. What is changed are DISKATE's internal pointers to where the source area is. When the O command is executed, the LAST file in the new source area is made current, and the target character is set to the 0 giving the end of the file. Of course, this file might be empty.

WITH NO ARGUMENT CREATES EMPTY SOURCE AREA STARTING SAME PLACE AS CURRENT <S>

The O command can also be given without any argument. In this case, rather than following the standard procedure of using the last argument computed, an empty source area will be established beginning at the same place that the current source area begins. This is very useful for "scrubbing" the work you've already done and starting with a clean slate to bring in a new file from the disk.

Using the O command you can maintain several different source areas in the memory. Suppose for instance you want to create a new source area at a special location which will be used by a piece of software that requires a file in a particular place. If you want to be able to go back to the current source area, you can save its location by:

OLDSOURCE1=<S>,OLDSOURCE2=>

The new source area can be set up using the O command, and if you want to go back to editing the old source area the command:

O OLDSOURCE1..OLDSOURCE2

will reestablish it as the source area.

The O command is what you want if you want to create a fresh source area somewhere in memory other than where DISKATE wants <S> to be, or if you want to make a given interval into the source area. But what if you want the whole source area as it already exists to be moved somewhere else? The DISKATE M command, which we´ve already introduced, is actually an extremely flexible and powerful command and can very easily accomplish moving whole blocks of memory -- not only moving the actual bytes but also changing the internal DISKATE pointers so the change will be properly kept track of. Suppose for instance that your source area begins at 2A00H and you have done some work editing a file. Now you decide you want to consult a BASIC program, which needs this area of memory. Of course your file could be saved on the disk and then reloaded, but perhaps the BASIC program is able to examine the file and you want the file still to be in memory for speed reasons. Let´s say that a block beginning at 7000H is free and big enough to hold the file. The single command:

M COMMAND UPDATES POINTERS

E.G. M<S> DOES MOVE & UPDATES <S>, <F>

M COMMAND WITH ^ OUTSIDE SOURCE AREA SIMPLY COPIES INTERVAL TO BYTE STARTING WITH TARGET CHAR

^7000H,M<S>

will move the entire source area to 7000H, and update the pointers delimiting both <S> and <F>. The M command is another which behaves differently if the entry pointer lies outside the source area, which here it certainly does. In such a case the interval to be moved is NOT deleted from anything, but is simply COPIED to the area of memory beginning with the target character. There is no insertion -- whatever used to be at the interval to which the argument is copied is simply overwritten.

There is a special case of using the M command with an interval outside the source area which may save you some trouble. When you are using the editor, deleting text using the K command actually works internally by calling the routine invoked by the M command. The INTERVAL TO BE DELETED IS MOVED IMMEDIATELY OUTSIDE THE SOURCE AREA, and, in the event that you change your mind and want back the interval deleted, it can be recovered if the source area has not been enlarged. However, the deleted text is NOT surrounded by 0´s, so it may cause

K COMMAND MOVES DELETED INTERVAL JUST OUTSIDE SOURCE AREA

some trouble in general to determine just what the boundaries are for this piece of text.

M COMMAND WITH NO ARG JUST AFTER K COMMAND RESTORES DELETED TEXT

However, if you act fast enough you may be able to recover deleted text with very little trouble. The M command updates all internal DISKATE pointers that it knows about, which includes < and >. After a Move command, these two pointers will give the location TO WHICH THE INTERVAL HAS BEEN MOVED. Since the K command issues an internal call to the M command routine, immediately after a K command < and > give the lower and upper addresses respectively of the area in memory to which the killed text has been moved. Thus, if you give a K command, and then IMMEDIATELY afterward realize that it was a mistake, GIVING THE M COMMAND WITH NO ARGUMENT JUST AFTER THE K COMMAND WILL RESTORE THE DELETED TEXT.

Suppose you delete an interval with a K command, and then decide you want the deleted text back but only after you've already given a command after the K command, say a ".. The text comprising the interval will still be just outside the source area, unless the command(s) you gave after the K command entered text. The deleted interval can be moved back to where it should go if a way can be found to specify the right interval. The first thing to do is to SAVE THE ENTRY POINTER by giving a variable=^ command. Unless you have given a command after the K command which changes the entry pointer, it will point exactly to the place where the text is to be put back. Suppose you know that you deleted 3 lines. In this case the interval of the deleted text can be specified by the argument:

S^ GIVES FIRST BYTE BEYOND SOURCE AREA

```
S^..-1|..3_
```

S^ is a symbol which we haven't seen before, and it denotes the address of the FIRST BYTE BEYOND THE SOURCE AREA. What about the -1? The -1 occurs in a position where DISKATE expects an interval argument of some kind, which means it is looking for something it can turn into an address (or actually a pair of addresses to be correct.) Remember that addresses are treated as 16 bit non-negative numbers. This means that the sign bit in -1 is treated as a digit bit, so -1 is the same as 65535, which is the highest address in the memory. So, the argument S^..-1|..3_ means: everything up to the 3rd carriage return within the interval starting just beyond the source area and extending to the end of memory. The deleted text can be restored by:

COMMANDS TO RECOVER 3 DELETED LINES

```
"S^..-1|..3_
M
```

In this case the " is given as a precaution, to

make sure we´re getting what we want, and saving the entry pointer was a precaution which wasn´t really necessary. However, in general you may not be sure how to specify the deleted text. In this case you may want to give " or # commands to try to pin down exactly where the upper bound is of the material that was deleted, and being able to set the entry pointer might be useful. In general, if you can figure out a pattern which will match the ending of the interval you want back, the commands:

```
"S^..-1|..pattern
M
```

will put you back on the track. Note however that you must set the initial reference interval by an argument such as S^..-1, since matching will always fail outside the source area unless the initial reference interval has been set.

C COMMAND

COPIES INTERVAL GIVEN AS ARGUMENT TO BYTES STARTING WITH TARGET CHAR

NO POINTERS UPDATED

There may be instances where you want to perform a block move but you don´t want DISKATE to know about it. For instance, it may be useful to have a separate copy in the memory of all the files in the source area. This can be achieved by using the C command, C for Copy, which works similarly to an M command with the target character is outside the source area. The C command copies the interval given as argument to the memory locations beginning with the target character. Whatever used to be at the block beginning with the target character is simply overwritten. The C command works the same whether the target character is in the source area or not, and does not update any of the DISKATE internal pointers as does the M command. Thus if you give the command ^1000H, and this location is outside the current source area, the difference between:

```
M<S>
    and
C<S>
```

is that after the M<S> the current source area will begin at 1000H, the pointers to <S> having been updated, whereas after the C<S> command the source area will be exactly where it was before the command. You should be fairly careful in using this command, since it would be easy to overwrite a part of the memory DISKATE is using for something else.

K COMMAND WITH ARGUMENT OUTSIDE SOURCE AREA WILL

As you might suspect by now, the K command also works differently if its argument is outside the source area. In this case, rather than removing the interval given as the argument, the K command will ZERO this interval. Zeroing a block of memory can be especially

ZERO THE INTERVAL GIVEN AS ARGUMENT

useful for machine language programming, where you may want to be sure that a program which is still being debugged is only modifying those areas of the memory that it's supposed to. By zeroing a large block of memory before you begin work, you can easily see with the # command whether a program has dumped garbage in the wrong place. This method also makes it especially easy to see how much stack is being used.

The DISKATE commands discussed in this section make DISKATE the most powerful monitor available for personal computers. They can be used in macros, just as the editing commands discussed in the previous section. For instance, suppose you wanted to fill a block of memory with repetitions of the pattern consisting of 80 ASCII blanks followed by a carriage return and a line feed. Let's say you want to set up 100 such lines, beginning at location 1000H. The commands:

```
N,E[*LINE
R80
E[ ]
],N,E[*FILL
^1000H
B16
R100
D[*LINE]
E#D A#
]
```

will set up a macro which will do this job and can be invoked by the command D[*FILL].

Suppose we want to do the same thing, but we want to fill all of the memory available up to 2000H, but without overwriting anything beginning at 2000H. Of course, we could calculate the number of lines that would be needed, figuring 82 bytes per line, and substitute that for the 100 in the macro above. Another way is to set up the macro to keep entering the lines until it determines that no more can be added. We can do this with the QF command. There is only one way that QF can produce a matching failure if the interval is given in the form expression1..expression2 where the two expressions do not involve matching: in the case that the value of expression1 is greater than the value of expression2. Using this fact, we can modify the commands above as follows:

```
N,E[*LINE
R80
E[ ]
],N,E[*FILL
```

```
^1000H
B16
R-1
QF^+82..2000H
D[*LINE]
E#D A#
]
```

This macro will keep on entering the lines until the entry pointer comes within 82 bytes of the area we want to leave alone, in which case it will quit.

Using techniques such as this there is no limit to the flexibility that DISKATE will allow. Again, as so often in this manual, it must be emphasized very emphatically that after using DISKATE as a monitor, if you want to resume editing text already established in the source area you must be absolutely sure the entry pointer has been restored in the proper place.

X COMMAND

To conclude this section we introduce a command by which you can call machine language subroutines from DISKATE. The X command, for eXecute, will issue a CALL to the lower address of the argument given. This can of course be either an explicit address or a more complex expression. If the machine language subroutine is properly written and exits with one of the 8080 return instructions, a call to it by an X command can be included in a command line or macro and the rest of the command line or macro will be executed upon return. If the address given is a program which goes into its own loop and does not return, such as a BASIC or DOS, then the X command will serve as a jump command, since such a program will certainly intialize its own stack. The details of how to write machine language programs for interfacing to DISKATE are discussed in Part II. Here's an example of using the X command. If you're using DISKATE with software that locates a DOS entry point at 2028H, you can get back to the DOS from DISKATE with the command:

ISSUES CALL TO MACHINE LANGUAGE SUBROU-TINE AT LOWER ADDRESS OF INTERVAL GIVEN AS ARGUMENT

```
X2028H
```

If you want to do this frequently you can set a variable, say DOS, equal to 2028H and then use the command:

```
XDOS
```

This method is highly recommended, since (as with any jump command) if you make a mistake typing an explicit address for the X command a system crash could result.

## 1.3 Using DISKATE input/output

In this section we'll deal first with terminal I/O handling and then talk about how to use the disk commands. All of the I/O in DISKATE is handled through calls to locations in a jump table, so that any type of terminal can be interfaced to DISKATE. The details of how to do this are discussed in Part II of this manual. For the moment all we need to know is the general structure of how DISKATE handles the I/O calls. The terminal I/O routines involve calls to the following basic drivers:

1. A routine to initialize the terminal.

2. A routine get one character of input, presumably from the keyboard, and return it in the A register.

3. A routine to output one character from the B register to the terminal.

4. A "panic detect" routine. This routine queries the terminal to see if a key has been pressed. If not it returns, with a flag set to indicate nothing is doing. Otherwise, if a key has been pressed it inspects the byte corresponding to the key, and if it meets the conditions set by the user to indicate that a process should be stopped, it returns with a flag set indicating this. Otherwise it returns with the flag set as before indicating nothing doing.

The basic terminal input and output routines can handle multiple I/O devices. When the routines are called, a device number is supplied in the A register. The routine can use this number to vector the call to one of several devices. Normally calls are made to device Ø -- i.e. Ø is in the A register when the terminal routines are called. The device number can be changed by giving an IO command. The command:

IO COMMAND

IO n

CAUSES ARGUMENT TO BE SUPPLIED AS I/O DEVICE NUMBER

will cause n to be henceforth passed in the A register to the terminal I/O routines, so that all of the terminal I/O from the time the IO command is executed will go to device n instead of device Ø. Also, when this command is executed, a carriage return and line feed will be printed on the device -- i.e. the output routine will be called with the new device number in the A register first for carriage return, and then for line feed.

CR, LF ISSUED TO

Here is how this can be used. Suppose you have

both a video terminal and a printer, and your drivers are set up so that when the output routine is given a call to device Ø it prints on the video terminal, but device 1 prints on the printer. To get a hard-copy listing of the current file you could give the command:

DEVICE

```
IO 1,"..,IO Ø
```

This would cause the ".. command to send its output to device 1, following which printout is restored to the video terminal by the IO Ø command.

The IO command applies to both subsequent input and output. If your printer does not have a keyboard, your input driver should be set up so that it gets the call no matter what the device number is. Otherwise, if you forget to give an IO Ø command at the end of a command string containing an IO 1, for instance, DISKATE will be unable to get any input. The IO command can also be given an argument other than a constant. The command:

```
IO DEVICE
```

would supply the value of the variable DEVICE as the I/O device number.

Y COMMAND EXECUTES TERMINAL INITIAL-IZATION ROUTINE (NO ARG)

If you don't have multiple input or output devices, your terminal routines can simply ignore the device number in the A register, and this discussion won't apply.

You can execute the terminal initialization routine at any time by giving the Y command, (Y for "Wipe",) which takes no argument. A device number is presented to the terminal initialization routine as for any of the other terminal I/O routines. If you have multiple devices, you may want your terminal initialization routine to initialize all of them at once, or you may want it to intialize only the device whose number is passed to it. If you do it this second way, then whenever you use a new device by means of an IO command, you will have to remember to give a Y command unless the device needs no initializing. All this is discussed more fully in II.1.

WID COMMAND SETS TERMINAL WIDTH TO ARG WITH NO ARGUMENT PRINTS TERMINAL WIDTH

Another command pertaining to the terminal allows you to set the terminal width. This is the WID command. If you give the WID command with no argument, it will print the current terminal width, while if you supply an argument, the low order byte of the value of the argument will be set as the terminal width. The widest line which the line buffer can hold is 130 characters, though if a WID command is given for a value greater than this, the value will be set as the width of the terminal for output purposes. This width determines how many characters are output before DISKATE automatically inserts a carriage return and line feed for an overflow line.

There is a simple command for printing a message on the terminal. The command:

ECHO COMMAND

ECHO[one-line character string]

PRINTS ARGUMENT ON TERMINAL

will simply cause the string within the brackets to be printed on the terminal. This differs from the " command, in that to print a message on the terminal using a " command, the message must be part of the current file. The echo command will print its argument without any matching, so that the contents of the current file don't matter.

PANIC DETECT PROVIDED RESPONDS BEST TO BREAK

The Panic Detect State is important so we'll spend a minute discussing it in detail. The panic detect routine is called periodically by DISKATE, including PRIOR TO THE OUTPUT OF EVERY BYTE TO THE TERMINAL. In the version supplied with DISKATE which is written for the serial I/O port on the disk controller, the panic detect routine will respond to ANY KEY ON THE KEYBOARD BEING PRESSED. Because there are timing considerations involved, the best way to register the panic state to DISKATE if you're using the panic detect routine provided is to press the Break key if your terminal has one. (These details are discussed more fully in Part II.) If you are writing your own panic detect, you may wish it to respond to only a certain character, such as a Control-C. In any event, if the panic detect routine returns to DISKATE signalling the condition that you desire a process to be stopped, DISKATE will STOP AND WAIT FOR THE NEXT CHARACTER TO BE INPUT FROM THE KEYBOARD.

UPON PANIC DETECT, PROCESS WILL STOP & WAIT FOR INPUT:

ESC = ABORT,

S = DO ONE CHAR & STAY IN PANIC STATE,

OTHER = RESUME

If this character is ESC, (1BH) then the ongoing DISKATE process will be TERMINATED. A question mark will be printed on the screen to echo the abort. If you type S instead of ESC, (S for Single-step,) the next character of the output will be printed but the panic state will remain in effect. Thus a single character will be printed, but prior to the output of the next character DISKATE will automatically go into the panic state as if you had again interrupted it. Thus if you want to slow down a printout, interrupt it by invoking the panic detect routine, and then repeatedly type S, or if your terminal will transmit a character repeatedly, transmit the S. This way the characters will be printed slowly enough for you to be able to read them as they are being printed. If you are in a panic state and you type any character other than S or ESC, the process will continue full steam. Recall that you can force entry to the panic state by executing the PAUSE command.

? AS ARG WILL

If a printout from a " command is interrupted by invoking the panic detect and then aborted, there is a way to continue the printout where you left off. The

symbol "?" can be used as an argument and will give the address within the file where the output was halted. For instance, if you abort a ".. command and then decide you want it to go ahead and finish, the command:

GIVE ADDRESS IN THE FILE WHERE PRINTOUT FROM " COMMAND WAS ABORTED

```
"?..
```

will work. In general, if you've given no commands since aborting the printout, you can use the command:

```
"?..>
```

to resume the printout and have it continue to whatever upper bound you orignally specified. This feature is implemented only for the " command, and will not work if a # command is similarly interrupted.

BACK-SPACE WIPES OUT LAST CHAR TYPED, ESC WIPES OUT CURRENT LINE

As we mentioned at the beginning of the manual, DISKATE scans its input and if the back-space key, 08H or Control-H, is pressed, the pointer to the line buffer holding the line currently being typed is backed up one character, and the following is echoed to the terminal: a back-space, a space, and another back-space. If you are using a video terminal this will erase the last character typed from the screen. You can do this any number of times to wipe out several characters that have been typed that contain mistakes. If you type ESC then the entire line is ignored. A backslash, "\", is printed on the terminal to echo this.

The details for creating the actual drivers to handle these I/O calls are discussed in Part II. At this point we are ready to begin discussing the disk file handling commands. Internally DISKATE is divided into two parts: a program called IO, which has all of the disk handling routines and other input/output, and a program called ATE which is independent of any particular I/O requirements. DISKATE comes with drivers for handling both the Disk Jockey controller and the North Star Disk. However, if you have another controller, it is possible to interface that controller to DISKATE by rewriting part of the module IO.

DISK FILE SPECI-FIED BY FILENAME COLON DRIVE

A disk file is named by a combination of the file name and a designator which indicates which drive the file is on. DISKATE maintains a CURRENT DRIVE, and the disk drive designator can be omitted when specifying a file, in which case the file will be assumed to be on the current drive. File names may be up to 8 characters long and can contain ANY PRINTING CHARACTERS EXCEPT: a comma or colon. In addition, the file name must not begin with " or @. The disk drive designator is a single letter, A through H. In the current version, drives A through D are vectored to routines for the Disk

DRIVE OMITTED ASSUMES CURRENT DRIVE

DRIVES

DESIGNATED BY LETTER

A-D DISK JOCKEY
E-G NORTH STAR

Jockey controlled drives 1 through 4, drives E, F, G are vectored to routines for North Star Disk drives 1, 2, 3, and drive H is undefined. To specify the disk drive along with the file name, the file name is given followed immediately by a colon followed by the drive designator. Thus a file named CONTRACT on Disk Jockey drive 2 could be specified:

```
CONTRACT:B
```

In your version of DISKATE as supplied, the current drive on power-up will be drive A. (In Part II we'll see how this can be changed.)

Each file has associated with it the following information:

1. The file name.
2. The location on the disk where the file resides.
3. The length of the file.
4. The file type.
5. The address in memory from which the file was saved.

BLOCK = 256 BYTES IS SMALLEST UNIT OF DISK STORAGE

File names we've already discussed. DISKATE assumes that each disk is divided into units called BLOCKS which are 256 bytes long. The block is the smallest unit of information for dealing with the disk. The actual details of how characters are stored on the disk are irrelevant here: the DISKATE block structure can be thought of as a logical construct which does not depend on the physical details of the acutal disk controller organization. The Disk Jockey controller, for example, will read and write information in units of 128 bytes. The location or address of the file, #2 above, is the BLOCK NUMBER of the file on the disk. The length is the number of blocks. Thus the smallest file possible in DISKATE is 256 bytes long. The length of a file is a one-byte number, so that DISKATE files are LIMITED TO A LENGTH OF 65K, or 256 blocks to be exact. DISKATE recognizes only two file types: source files and non-source files. A source file has a type of Ø, and a non-source file may have any other type. The file type is given by a one-byte number. When DISKATE saves a file on the disk it automatically sets the file type. Any file saved FROM WITHIN THE SOURCE AREA will be assumed to be a source file and will be given type Ø, while any other file will be given type 1.

FILE LENGTH LIMITED TO 256 BLOCKS

SOURCE FILE IS FILE SAVED FROM SOURCE AREA

All of this information is stored in a special location on the disk called the DIRECTORY. The directory occupies the first 4 blocks on each disk, and each file on the disk has an entry in the directory which is 16 bytes long. The format of the directory is:

DIRECTORY USES 1ST 4 BLOCKS,

| byte | contents | 16 BYTES PER FILE |
|---|---|---|
| Ø-7 | file name | |
| 8-9 | disk address | |
| 1Ø | file length | |
| 11 | ignored | |
| 12 | file type | |
| 13-14 | memory address | |
| 15 | unused | |

North Star Disk users will note that this format is completely compatible with North Star DOS. In this table bytes are numbered with low order bytes getting low numbers, and addresses follow the 8Ø8Ø convention of low order byte in the low order address. Because each entry in the directory is 16 bytes long, the directory has space for 64 files. Thus each disk is limited to 64 DISKATE files.

THOU SHALT MAKE BACKUPS!

PLEASE!

As with any floppy disk system, a program which goes out of control and manages to dump garbage on the directory of a disk MAY RENDER THE ENTIRE DISKETTE UNREADABLE!! For this reason you are emphatically advised to make backups of all disks containing important information. If you have a dual disk system, backups can be made by copying entire diskettes. DISKATE has a command for doing this, which we'll see below. If you have only a single drive, you can keep "consecutive backups" by STORING A FILE ON A BACKUP DISKETTE EVERY TIME YOU STORE IT ON THE WORKING DISKETTE. If you consider this to be too much of a nuisance to be worth the bother, it is a safe bet you will change your mind the first time you lose a whole diskette worth of information.

S COMMAND

SAVES INTERVAL GIVEN AS SECOND ARGUMENT ON FILE WHOSE NAME IS FIRST ARGUMENT

Now let's begin the discussion of the file handling commands. Suppose you are working on a file which is brand new -- you started with an empty source area and have created the file for the first time using the editor commands, and now it must be saved on the disk. Let's say you want to save it under the name SPECS. This can be done with the command:

```
S SPECS <F>
```

The DISKATE file handling commands will do a lot of work automatically, so we had better be careful to understand just how the commands work. First of all, since only

SPECS was given as the file name, DISKATE assumes that SPECS is to be saved on the current drive. Before the command above was given, presumably there is no such file as SPECS. No matter, DISKATE will create the file. The information saved on the file is the interval which is the second argument of the S command. In this case we've given the entire current file as the argument. Because <F> is surely within the source area, SPECS will automatically be given type Ø, which makes it a source file. In the memory address section of the directory, the current address of the beginning of the current file will be written, though for source files this information is generally not consulted.

IF NO SUCH FILE EXISTS IT IS CREATED

Now suppose you do some more work editing the file, and want to save the newer version. The identical command: S SPECS <F> will save the current file as the disk file SPECS. Note that the old contents of SPECS on the disk are overwritten by the S command. So: the S command will create a new file if there is no file corresponding to the file name given as the first argument; if there is such a file its old contents are overwritten. Note that this is a great convenience, as long as you haven't made a mistake in typing the S command. However, it does make it possible to get into the following tangle. If you make a mistake entering the file name in the S command and specify a file which doesn't exist, when you meant to specify a file which does exist, the file will be stored under the incorrect name. This will have two consequences: there will be an extra file on the disk under an erroneous name, and the file with the correct name will not contain the latest version of the file. Below we'll see some ways to prevent this type of situation.

One other caution. To specify the current file as the interval to be saved, we used the argument <F>. We could have used .. instead of <F>, since like most DISKATE commands, the initial reference interval for the S command is the current file. Remember, though, that if you are using the REF command to specify a different initial reference interval, this will also apply to the S command. We used <F> to specify the current file because it gives the current file without matching, regardless of the initial reference interval.

Here's one way of avoiding the problem of saving a file under the wrong file name. Whenever a DISKATE command is to be given a file name, the symbol " can be given in place of the file name and refers to THE MOST RECENTLY REFERENCED FILE. Thus if you create a brand new file and save it by a command S SPECS <F>, as above, and then do some more work on the file and want to save the updated version as the same file, you can use the command:

SYMBOL "

CAN BE USED IN PLACE OF FILE NAME FOR MOST

RECENTLY REFERENCED FILE

S " <F>

PROVIDED you have not given any disk commands with other file names or changed the current drive in the meantime. Even immediately after power-up, " will have a meaning, since as we'll see below, when DISKATE powers up it references a special file called STARTUP.

?" COMMAND

If you want to see what the most recent file is, you can give the command:

SHOWS MOST RECENT FILE

?"

This will print the file name and drive on the terminal. Use of the " file name provision is highly recommended.

So far we've said nothing about the length of the file after an S command. When the S command is given, the length of the file will be set at just enough blocks to hold the interval saved. Suppose the file already exists and the length is not long enough to hold the interval? If there is a contiguous block on the disk large enough for the interval that is free, DISKATE will save the file in this block and release the block where the file existed before. If there is enough free space on the disk to hold the interval but not in one block, DISKATE will compact the disk and make the file just long enough to hold the interval. In this case a message of the form:

FILE LENGTH SET AT JUST LONG ENOUGH TO HOLD INTERVAL

IF FILE TOO SHORT DISK WILL BE COMPACTED

COMPACTING ON DRIVE (drive designator)

will be printed while the compacting is going on. In this way DISKATE handles all disk space allocation for you, and you needn't be concerned about allocating lengths for files. Of course, the situation can always arise that you run out of space on a diskette. In this case DISKATE will print an error message to this effect on the terminal. It can also happen that the directory becomes full if there are 64 files on the diskette. An S command for a new file will also cause an error in this situation. The error messages will tell you that either the disk or the directory is too full, and will tell you which drive. A similar error message will be printed in case of hard disk errors.

Because DISKATE will automatically compact the disk if an existing file is too short, if you have a DOS or other piece of software that can directly manipulate the directory, you MUST NEVER CREATE OVERLAPPING FILES as the DISKATE compacting routine will not work if files overlap. Of course, if all of the files on a diskette were created using DISKATE, this will be no problem.

NO OVERLAPPING FILES ALLOWED!

Any interval can be given as the second argument of

an S command. The general form of the S command is:

S filename interval

Here the interval MUST be given -- you can´t use the default argument for the disk commands, to prevent errors. Using an S command for an interval smaller than the whole file gives a simple way to BREAK UP A FILE INTO SMALLER PARTS. Editing a file in parts has some advantages. There is less chance that you will run out of room in the memory just at a time when you don´t want such considerations to disrupt your train of thought in editing the text, and DISKATE will run faster with smaller files. Suppose you have a file called SPECS, for instance, and you want to break it into 2 parts, the first part being the first 147 lines and the second part being the rest. Let´s call the two parts SPECS and SPECS´. (Note that ´ is a valid symbol to be part of a file name.) We´ll also assume that the whole thing is in the memory as the current file. To break it up into two pieces you can use the commands:

S SPECS ..147!
S SPECS´ 148!..

(In this case we´re assuming that the REF command is not being used.) Note that after these commands, " used as a file name will denote SPECS´. You can also save intervals from outside the source area. We´ll see some examples of this below.

To examine what is on a disk you can give the I command -- I for Identify. The I command with no argument will list the files on the current drive. If you want the files on a specific drive listed, the drive designator can be given as an argument. For each file the listing will tell you the length of the file, whether it is a source file or not, and if not will give the memory address. The listing will also print the number of blocks of free space on the disk, as well as the number of remaining directory entries. The address of the file on the disk is not included in the listing from an I command, since you don´t have to worry about specific disk addresses. However, if you need all the information that exists in the directory, DISKATE keeps a copy of the entire directory in the memory, and we´ll see in Part II how you can access this if you need to. If you are using many S commands without using the file name ", giving the I command periodically is a good idea to make sure there are no extraneous files on the disk resulting from an erroneous S command.

You may want to know only how much space is left on a disk, without needing a complete listing of all of the

I COMMAND LISTS FILES ON DISK DRIVE GIVEN AS ARGUMENT WITH NO ARGUMENT LISTS CURRENT DRIVE FS COMMAND PRINTS AMOUNT OF FREE SPACE ON DRIVE SAME ARG CONVEN-

files. This can be obtained with the FS command, FS for Free Space. The argument for this command works the same as for the I command: either a disk drive designator, or no argument indicating the current drive.

Of course there has to be a way to delete a file, and this is accomplished with the U command, U for Unsave. The U command takes a file name as its argument, and deletes the file from the disk. Of course, the file name can include a disk drive designator.

At this point we come to another important command, the command for loading a file. Although it might seem straightforward, there are several ways that trouble can occur if not used carefully. The L command, L for Load, will load the file given as the first argument into the memory. The memory address where it is loaded can either be given as the second argument, or omitted, in which case DISKATE will determine the load address in the following way. If the file is not a source file then it is loaded at the memory address given by the memory address portion of the directory entry for the file. If the file was saved by DISKATE then this address will be the address in memory where the file began at the time it was saved. If the file was created by another piece of software, you should always give the load address as a second argument to the L command unless the file is of type Ø. Otherwise there may be no telling what is writtten in the space in the directory where DISKATE expects the memory address. You will get quite a surprise if you give an L command without a load address and the file clobbers part of the resident software.

If the file to be loaded is a source file and the load address is not given as a second argument, then it is INSERTED into the current file between the target character and the preceding character -- i. e. excatly where text is inserted by an E or M command. The target character will then be set to the first character after the text loaded from the disk file. There are two important considerations here. First, if the target character happens to be outside the source area, then the file will be loaded at the location given by the entry pointer OVERWRITING WHATEVER IS THERE. The source area remains unchanged. Because an L command can affect a very large number of bytes, an error of this kind can be catastrophic. If you have been using DISKATE as a monitor to work with memory outside the source area, then give an L command with no second argument for a source file but have forgotten to restore the entry pointer, this could overlay a substantial portion of the memory and possibly clobber part of the resident software resulting in a crash. Second, even if the entry pointer is in the proper place, if the file is a long

TION AS I COMMAND

U COMMAND

DELETES FILE FROM DISK

L COMMAND

LOADS FILE GIVEN AS 1ST ARGUMENT TO MEMORY LOCATION GIVEN AS 2ND ARG

IF NO 2ND ARG GIVEN, USES MEMORY ADDRESS FROM DIRECTORY IF NOT SOURCE,

INSERTS AT TARGET CHAR IF SOURCE AND ^ WITHIN SOURCE AREA,

ELSE OVERWRITES

one it will expand the source area considerably. You should always check to make sure that the expanded source area will still fit in the area of your memory that is safe for the source area to occupy. If you want to work with only a single source file all by itself, it´s always good practise to give an O command with no argument first. That way you have an empty source area to work with.

AT ^

GIVE O COMMAND BEFORE L TO WORK WITH ONLY ONE FILE

Obviously you can use the L command to load into memory a file which was previously saved. But there are also some powerful uses for the L command in addition to this. We recall that a K command followed by an E command will replace the deleted text by the text entered with the E command. A K command followed by an L command which gives a source file as the first argument and has no second argument will REPLACE THE DELETED TEXT BY THE LOADED FILE. In this way it is possible, for instance, to maintain a library on the disk of standard paragraphs, use an abbreviation to denote the paragraphs in editing and then write a macro which will substitute directly from disk the paragraph for the abbreviation. This can be used to write form letters, just to cite one application.

Another important way the L command can be used is to concatenate or combine files. Suppose we go back to the example above where a file SPECS was split into two files, SPECS and SPECS´. Suppose that after further editing the verbosity is reduced and the two file are now small enough that it would be more convenient to have them combined into a single file. Because the two files were saved from the source area they are both source files. The job we want to do can be done by the following commands:

O,
L filel,
L file2

```
O,I
L SPECS,L SPECS´,"..
U SPECS´,S SPECS ..
```

WITH filel, file2 SOURCE FILES

The O command will collapse the source area so that the combined file will not pick up any extraneous information. The I command is given so that we can verify that there will be enough room in the memory to hold the combined file. Note that the O command will put the entry pointer at the end of the empty current file, which will be the only file in the source area. After L SPECS, the entry pointer will be at the end of the material loaded, so we can go ahead and give the second L command. ".. is given to make sure the file looks correct before we go ahead and make any changes on the disk. Now that we know that the file is correct in the memory, we FIRST delete SPECS´. That way there will be no doubt that there will be room on the disk for the combined file.

WILL COMBINE THE FILES INTO ONE FILE AS THE CURRENT FILE IN MEMORY

The S command will save the file and juggle the position of the files on the disk if necessary to make room for it.

Suppose you are working with source files in this way and give an L command, but by mistake specify the wrong file, so that you want to delete that part of the current file in the source area that you just loaded. Of course, you could backtrack, give an O command and set up the current file all over again up to the point you gave the load command, but this is a lot of work, and anyway you may not be able to reconstruct how it was done. DISKATE has a special symbol which will help in this case. <R> is a symbol analogous to <S> and denotes the INTERVAL IN MEMORY OCCUPIED BY THE LAST RECORD (I.E. FILE) READ IN FROM DISK. If you are combining source files by the technique above and realize that the last L command given was in error, that part of the current file that was put there by the last L command can be deleted by the command:

<R> GIVES INTERVAL OCCUPIED BY LAST RECORD LOADED FROM DISK

K<R> DELETES LAST TEXT LOADED

Because the K command will position the entry pointer to the end of the deleted material, you can immediately follow it with the correct L command, and the mistake will be corrected.

There are several disk commands which can be used to copy information or change the status of files. If you want to CHANGE THE CURRENT DRIVE, the command CD, which takes as its argument a disk drive designator, will establish the drive indicated by the argument as the current drive. In Part II we will see how to customize your system so that DISKATE knows which drive designators are valid. It is important that this be done correctly, or else a CD command to a non-existent drive will cause very bad mischief. If you have followed the customization procedure outlined in Part II, you will not have problems, but here is what will happen if you give a CD command by mistake for a non-existent drive. DISKATE will go ahead and register the drive as the current drive without giving an error message, but after that, EVERY SINGLE DISK COMMAND FOR THE CURRENT DRIVE WILL GIVE A DISK ERROR. You will know what has happened, since the disk error messages print the drive designator for the drive that caused the error, and this will show a drive that you don't have.

CD COMMAND TAKES DISK DRIVE DESIG-NATOR AS ARG & MAKES IT CURRENT

You can RENAME a file with the RN command, in the form:

RN COMMAND

RN oldname newname

RENAMES FIRST FILE

be sure to remember to separate the two file names by a

blank. Of course in renaming a file, the file does not change its location on the disk, so that a disk drive designator in newname is superfluous. If one is present it is ignored, even if it incorrectly specifies a drive other than the one the file oldname resides on.

AS 2ND FILE

It frequently happens that you want to copy the contents of a file from one file to another. This is done with the DISKATE T command, T for Transfer. The T command has the form:

T COMMAND

COPIES FIRST FILE TO 2ND FILE

```
T oldfile newfile
```

In this case drive designators are significant, since the file to which you want the information copied could be on another drive from the file which is to be copied. The DIRECTORY ATTRIBUTES -- i.e. size, file type, and memory address are also copied. The T command works analogously to the S command in that if newfile does not exist it is created.

An entire disk worth of information can be copied at once with the TD command, which stands for Transfer Disk. The TD command has the form:

TD COMMAND

COPIES ALL FILES ON 1ST DRIVE TO 2ND DRIVE LEAVING ALONE FILES UNIQUE TO 2ND DRIVE

```
TD olddrive newdrive
```

where both olddrive and newdrive are drive deisgnators. The TD command is more flexible than the usual copy disk command found in floppy disk operating systems. Usually such a command will copy the old disk onto the new disk sector for sector, completely overwriting whatever was on the new disk. The DISKATE TD command WILL COPY TO THE NEW DRIVE ALL FILES FROM THE OLDRIVE LEAVING ALONE ALL FILES ON THE NEW DRIVE THAT DON'T EXIST ON THE OLD DRIVE. To be specific,

```
TD olddrive new drive
```

is equivalent to giving:

```
T file:olddrive file:newdrive
```

for every file on olddrive. Thus if a file from olddrive does not exist on newdrive it is created. If it does exist the contents from the file on oldrive are copied into it.

Of course, it is possible that if there is already considerable information on newdrive that DISKATE will run out of room before the TD command is finished. If this happens, a message informing you of this fact will be printed. In this case not all of the files from olddrive will have been transferred. However, there will be no files which were only partially transferred.

DISKATE will check BEFORE copying the file to see if there is room, and if not the TD command will terminate. When this happens you may want to give the command I newdrive to see what files exist on this drive that weren't on olddrive, and delete some of them with a U command to make more room.

W COMMAND

WRITES VALUE OF 2ND ARG IN MEMORY ADDRESS PART OF DIRECTORY FOR FILE GIVEN AS 1ST ARG

The memory address portion of the directory for a file can be set to a given address by using the W command, which stands for Write address. The W command has the form:

W filename argument

where the argument is the same kind of DISKATE argument that would be recognized by the editing and monitor commands. The only thing that is changed by the W command is the memory address. This is especially useful in connection with the assembler, where during assembly the object code is placed one place in the memory but when executed the code is to be loaded at another place. We'll look at this in more detail in the next section.

GO COMMAND

LOADS FILE AND EXECUTES IT

We conclude this section with a command which is extremely convenient but which can also cause serious grief if used incautiously. The GO command, which takes only a file name as its argument, will load a file from disk and then automatically execute it. If the file is a source file, then:

GO file

is equivalent to:

EQUIVALENT TO L & D COMMANDS FOR SOURCE FILES, L & X FOR NON-SOURCE

L file,D<R>

whereas if the file is NOT a source file, then GO file is equivalent to:

L file,X<R>

The same cautions that applied to the L command apply here, but in addition you will have lots of trouble if the file type has somehow become incorrect. If you are processing a source file with other software which requires it to be of a non-Ø type, and then forget to retype it as Ø, a GO command to that file will cause DISKATE to call it as a machine language subroutine, which will almost certainly result in a crash. Note that just as with the L command, you must be careful to make sure that ^ is in the right place, or you can clobber part of the memory with a GO command to a source file. If the file is an edit macro to be used to work on the current file, SAFE PROCDURE IS TO GIVE AN N

COMMAND BEFORE THE GO COMMAND. Note that in this case the macro will have to restablish the file that was current before the N command as the current file.

## I.4 Using DISKATE as an Assembler

Space will not permit this to be a general textbook on 8080 assembler programming. It will be assumed that you are at least roughly familiar with what an assembler is all about, and that you've had some exposure to the 8080 instruction set. The DISKATE assembler operates in TWO PASSES. In Pass 1, the assembler does not create any object code. Rather, it goes through the source code keeping track of where each symbol is defined. In the process it compiles a SYMBOL TABLE. The symbol table is a string of items containing the symbol and then a 16 bit number giving the value of that symbol. At the end of the table is a 0. The symbol itself is recorded in the following format. There is no field for the length of a symbol. Instead the characters of the symbol occur until the last character, which has the high-order bit set. Thus the symbol AB would be given the hex value 41 C2. In hex normally the character A has the value 41, and B has the value 42. Setting the high-order bit for the B gives the value C2. The symbol table will also contain an entry for every DISKATE variable which has been given a value. There is no distinction made between a variable given a value with an = command and a variable given a value by Pass 1 of the assembler.

ASSEMBLER PASS 1 COMPILES SYMBOL TABLE

SYMBOL TABLE FORMAT:

CHARS OF SYMBOL, LAST CHAR HAS HIGH-ORDER BIT SET, THEN 16-BIT VALUE OF SYMBOL

INCLUDES VARIABLES

There are several commands which can be used to deal with the symbol table. The symbol <T>, analogous to <S>, is a DISKATE symbol which can be used in an argument to DENOTE THE SYMBOL TABLE without any matching. For instance, if you want to know where the symbol table is in the memory, you can give the command:

<T>

```
?<T>
```

DENOTES SYMBOL TABLE

Just as the symbol S^ denotes the address of the first byte beyond the source area, T^ is a non-matching symbol which denotes the first byte beyond the symbol table. This symbol can be useful in commands which serve to allocate memory. We'll see an example of how this can be used below.

T^

DENOTES 1ST BYTE BEYOND <T>

It is the user's responsibility in DISKATE to see that the symbol table is not overlaid and is located within the memory so that there will be enough space for it. THE MOST COMMON SOURCE OF MYSTERIOUS ERRORS USING THE ASSEMBLER COMES FROM A FAILURE TO MAINTAIN THE INTEGRITY OF THE SYMBOL TABLE. For instance: you can be making an assembly listing and everything is going normally, when all of a sudden every symbol is flagged as not having been defined, when you can see their definition right in front of your eyes. What will have

UNUSUAL ASSEMBLER ERROS

CHECK

<T>

happened in this case is that <T> is located at the top of RAM and has overflowed off the top. No matter where you have the symbol table in the memory, when assembling a large program for the first time it is a good idea to have a ?<T>, or perhaps several, in the middle of the assembly so you can keep track of how large <T> is getting, to make sure it won't overflow the area it is safe for it to occupy.

Z
COMMAND

Every symbol will be retained in the symbol table until a command is given to remove it. The Z command, given with no argument, will Zero -- i.e. remove from the symbol table -- all symbols except those symbols that are always initially present in the symbol table on power-up. As mentioned in I.1, these are:

NO ARG
ZEROS
SYMBOL
TABLE
EXCEPT
FOR
INITIAL
SYMBOLS

A, B, C, D, E, H, L, M, SP, PSW

When the Z command is given with no argument, DISKATE rewrites a fresh copy of the initial symbol table, so that if the variables had inadvertantly been given the wrong values, the Z command will restore the initial symbol table to the correct values. If you assemble a program, see some mistakes, then assemble it again without having zeroed the symbol table, you will get an enormous number of assembler errors for having defined symbols twice. If you have symbols of your own that you want to keep in the symbol table, such as the addresses of macros, for instance, you can use the Z> command. This command takes a symbol from the symbol table as its argument, and will ZERO ALL SYMBOLS DEFINED SUBSEQUENTLY TO THE ARGUMENT SYMBOL.

Z>symbol
COMMAND

ZEROS
ALL
SYMBOLS
DEFINED
AFTER
symbol

Suppose for instance you have defined all of the symbols that you will be wanting to keep. You can give the command:

KEEP=Ø

to define the variable KEEP as a marker for those variables that will need to be zeroed for each assembly. Then, to zero all of the symbols except those you want to keep you can give the command:

Z>KEEP

Zsymbol
COMMAND

You can also zero a specific individual symbol. This is done by giving the Z command with the symbol as an argument. The symbol will be removed from the symbol table and the symbol table will be compacted if the symbol zeroed was not the last symbol in the table. Note that in giving this command, there must be NO BLANK BETWEEN THE Z AND THE SYMBOL. If there is such a blank, DISKATE will perform a straight Z command and zero the

ZEROS
ONLY
symbol

entire symbol table.

There are several ways of managing the symbol table using commands we have already seen. Above we mentioned that the M command will update all of the relevant DISKATE pointers. This includes the pointers that define the boundaries of <T>. If your memory is limited, you may want to allocate only a small amount of space to <T> while editing, when the number of symbols in use is small but you will need a lot of space for <S>. But, as we'll see, DISKATE can assemble directly from disk, so that when assembling you may not need a large source area but will want a large space for <T>. Thus the ability to change the location of the symbol table in the memory is important. Let's say you have the symbol table at 1EØØH and want to move it to 2CØØH. This can be done with the command:

M<T> COMMAND

WILL RELOCATE SYMBOL TABLE TO START AT ^

^2CØØH,M<T>

The symbol <T> can also be used with the L and S commands to save and restore symbol tables. This is a useful technique for assembling one program to go with another, without having to assemble the two of them together. For instance the command:

S TBLSAVE <T>

will save the symbol table in a file called TBLSAVE. Now suppose you want to restore the symbol table to the way it was just before you saved TBLSAVE. There are two things that have to be done. First we give the command:

L filename <T>

L TBLSAVE <T>

WILL OVER-WRITE <T>

This will OVERWRITE the current symbol table with the contents of the file TBLSAVE. Now there is one problem. DISKATE will still have the old value stored internally for the upper bound of the symbol table. It is unlikely that this will be the same as the upper bound for the table saved in TBLSAVE. To set the upper bound of a symbol table just read in from disk, the easiest method is to SET A VARIABLE NOT IN THE SYMBOL TABLE EQUAL TO Ø. So, if we can be sure the variable JUNK was not in the symbol table saved in TBLSAVE, the command:

TO SET UPPER BOUND SET UNUSED VARIABLE = Ø

JUNK=Ø

will establish the upper bound of the symbol table. The reason this works is that when an = command is given, DISKATE searches through <T> from the beginning for the symbol, and when it finds it assigns it the value. It defines the symbol if it reaches the end of <T> before

finding it. But, in defining a new symbol, the size of the symbol table increases, so a new upper bound for <T> must be recorded. This is why an = command to a non-used variable will set the upper bound. This technique will be the cornerstone of the process of customizing DISKATE discussed in Part II.

In many cases you may find that commands will work properly if you give only an L filename <T> command without setting a dummy variable = Ø. For instance, you could give a command:

```
L TBLSAVE <T>
```

and then a command ?symbol where symbol is one of the symbols in TBLSAVE, and even though DISKATE will not have recorded internally the proper upper bound for the symbol table TBLSAVE, the ? command will work properly anyway. This is because the ? command will scan through <T> taking the Ø at the end as its indication of the end of the table, rather than consulting the pointer giving the upper bound. However, if you have not set the value of a dummy variable and then give the command ?<T>, an incorrect value will be printed for the upper bound of <T>. Under these circumstances if you move <T> with the M command, you may very well end up moving only a small part of it and losing most of your symbols. In all of our examples in this manual we have gone to the extra trouble to set the dummy variable just to be safe.

Now that we've thoroughly discussed the symbol table, we can get on to describing how the assembler works. A complete understanding of <T> is crucial, because as mentioned above, not properly managing <T> is the greatest source of peculiar assembler errors.

Assembler language statements are divided into two types: PSEUDO operations and MACHINE INSTRUCTIONS. The machine instructions are each coded into a single machine language instruction of object code, whereas the pseudo ops are directives to the assembler itself. There are four fields to every assembler statement, some of which may be empty:

1. The label field
2. The opcode field
3. The operand field
4. The comment field

The DISKATE assembler allows you to use TWO DIFFERENT formats for specifying these fields. In the PROCESSOR TECHNOLOGY FORMAT:

PROCESSOR TECHNOLOGY FORMAT

a. The label field begins in the first character of each line. If an assembler statement is not to be

given a label, the first character of the line must be blank. The label is given without any symbol to terminate it. A label must contain only upper-case letters or digits and must begin with a letter.

b. The opcode field is separated from the label field by one or more blanks.

c. If the statement takes an operand, the operand field is separated from the opcode field by one or more blanks

d. The comment field is separated from the previous field by one or more blanks.

e. The entire line can be designated as a comment by giving * as the first character on the line.

INTEL FORMAT

The INTEL format differs from this in the following respects:

a. A label is terminated with a colon. If a statement is not to be given a label, the opcode may begin the line since the absence of the colon following opcode designates it as the opcode field.

b. The comment field begins with a semicolon (";").

INTE PROS COMMANDS OR PSEUDOS

In both formats the mnemonics for the machine instruction opcodes are the same, which are referred to as INTEL MNEMONICS.

SET FORMAT AS INTEL, PRO. TECH. RESPECTIVELY

On power-up, DISKATE will recognize the Processor Technology format unless you have customized your system so that it comes up recognizing the Intel format. (The method for doing this is described in Part II.) You can change the format for the assembler using INTE for Intel format, and PROS for Processor Technology. These can either be given from the terminal as commands with no argument, or as pseudo-ops in a source program. Thus a program can be a mosaic of segments in both formats.

When the assembler begins Pass 1, it will be going through your source statements, counting where the bytes of object code will go but without actually putting anything anywhere. The assembler needs to know where the first machine instruction is to be located. In DISKATE this is handled by two internal 16 bit locations: the assembler PROGRAM COUNTER, which is denoted by &, and the assembler STORAGE COUNTER which is denoted by $. The program counter keeps track of the location of each machine instruction in terms of WHERE THE INSTRUCTION WILL BE IN THE MEMORY WHEN EXECUTED. The storage counter keeps track of WHERE EACH INSTRUCTION IS STORED IN

& GIVES PROGRAM COUNTER LOCATION OF INSTRUCTIONS WHEN EXECUTED

THE MEMORY AS THE OUTPUT OF PASS 2 OF THE ASSEMBLER.

The storage counter is not changed during Pass 1, since no object code is generated during this pass. DISKATE needs an initial value for the program counter. You may want the default value for & to be Ø, but then again you may not. The value of & at power-up can be set by the customization procedure described in Part II. You can give it any value within the range of 16 bits. You can also set its value using the & command. This command is analogous to the ^ command: it sets & to the value of the argument. Likewise, the $ command will set the storage counter to the value of its argument. When you customize DISKATE you must be sure that the power-up value for $ is the location of a place in memory where object code can safely go, in case you forget to set $ from within an assembler source program.

$

GIVES STORAGE COUNTER LOCATION WHERE INSTRUC-TIONS ARE STORED BY PASS 2

Here is one way the & and $ can be used. Suppose you have a machine language program which you wish to use in connection with editing, but you don't want to commit yourself to where in the memory it will go. Now suppose you have done some work on a source file, and have decided you want this program to be brought in from the disk and to reside 1Ø24 bytes past the end of the source area. First give the command:

&

COMMAND SETS & TO ARGUMENT

```
&S^+1Ø24,$
```

then assemble the program directly from disk. We'll see how to do this below. The method of not specifying initial values for & and $ within a source program but relying on $ and & commands from the terminal can be used to great advantage for flexibility as to where a program will go, but it does have some risk. Assembling the program and forgetting to give the values for & and $ will deposit the code in the wrong place in memory, which could lead to trouble.

$

COMMAND SETS $ TO ARGUMENT

In many cases, possibly in most cases, you will want to set the initial values for & and $ from within the assembler source program. There are three pseudo-ops for this purpose. The pseudo-op AORG (for Assembler Origin) will set & to the value of the operand, while SORG (for Storage Origin) will set $ to the value of the operand. It is important to remember that a SORG statement DOES NOT TAKE EFFECT UNTIL PASS 2, unless $ is used as part of an operand of an assembler statement. A SORG statement will set $ during Pass 1, but after that the storage counter is left completely untouched until Pass 2.

AORG PSEUDO SETS & TO OPERAND, SORG SETS $ TO OPERAND

Another pseudo-op that affects both & and $ is ORG. The value of separating the function of & and $ is that it allows you to assemble programs which will execute from a place in the memory currently occupied by some-

thing else in the DISKATE environment. In such a case you want the object code to be stored in a different place from the place where it will go when the program is run. Here you will need separate AORG and SORG statements. However, in many cases you will be able to assemble a program directly into place. This will have the advantage that you can begin debugging immediately after assembly, while the symbol table is still in the memory. In this type of situation you will probably want to use the ORG statement. The ORG statement resets both & and $ by the following rule:

ORG

SETS & TO OPERAND AND CHANGES $ BY THE SAME AMOUNT AS & WAS CHANGED

1. & is reset from the value it had before the ORG, which we'll call old&, to the value of the operand of the ORG statement, which we'll call new&.

2. $ is reset to $ + new& - old&.

In other words, first & is reset to the operand of the ORG statement, and then $ is reset BY THE SAME RELATIVE AMOUNT AS & WAS. This means that a source program only needs to have one instance of a separate AORG and SORG at the beginning -- after that ORG will maintain the proper offset between & and $ that was established with the AORG and SORG. However, if you go on to assemble a new program with a different offset between the initial value of & and the initial value of $, you may have to first reset & and $ from the terminal.

Operands of assembler statements can be labels, and can include arithmetic operations +, -, *, and / provided the expression can be given a value at the time the assembler encounters it. You can also use DISKATE symbols such as <S>, <T>, $, &, etc., but you can't use complex arguments involving matching of the kind we saw in section 1. The pseudo-op EQU (for EQUate) can be used to assign the value of an expression to a label. If an EQU statement has a label, though, that label had better not appear in the operand. Thus, if you had a table and wanted to set a label equal to the length of the table, you could have:

PSEUDO EQU

ASSIGNS LABEL THE VALUE OF THE OPERAND

```
TABLE . . .
   .
   .
   .
TBLEND  EQU  &
   .
   .
   .
TBLLEN  EQU  TBLEND-TABLE
```

but you can't have:

```
TABLE  EQU  TBLEND-TABLE
```

Also, you cannot give a label a value with an EQU more than once. In any assembly, labels are defined once and then retain that value throughout the duration of the assembly process, unlike variables that may be changed at any time. However, you CAN attatch several labels to the same location in a program. For instance, if you had a label called DATA to indicate the beginning of a data section, and TABLE was to be the first label in this section, both DATA and TABLE should label the same place. This can be accomplished by giving an assembler statement with the label DATA as THE ONLY FIELD PRESENT. In Intel format this might look like:

FOR MULTIPLE LABELS AT ONE LOCATION GIVE STATE-MENTS WITH NAME FIELD ONLY

```
DATA:
TABLE: . . .
```

Most of the other pseudo-ops in the DISKATE assembler are standard for most assemblers. To define constants you can use the pseudo-ops DB, for Define Byte, and DW for Define Word. DB creates a one-byte constant with the value of the operand, and DW creates a 16-bit constant with the value of the operand. In addition to numerical expressions, you can also enclose an ASCII character in single quote marks. Thus to define a one-byte constant with the label STAR which is an ASCII asterisk, in Intel format you could give the statement:

DB PSEUDO

DEFINES BYTE

DW

DEFINES WORD (16 BITS)

```
STAR: DB '*'
```

WITH VALUE OF OPERAND

Operands such as '*' can also be used with the EQU pseudo and machine instructions. DISKATE will also allow you to use multiple operands for a DB or DW, separated only by commas. Thus to define three consecutive bytes of 0's you could use:

MULTIPLE OPERANDS ALLOWED

```
DB 0,0,0
```

To define a block of storage where you don't need to give the bytes in the block an initial value you can use the pseudo DS, for Define Storage. It will advance & and $ by the value of the operand, creating the area as a block of free storage. For instance, to define a block called BUFFER 80 bytes long, in Processor Technology format you could use the statement:

DS

DEFINES STORAGE BLOCK WITH LENGTH GIVEN BY OPERAND

```
BUFFER DS 80
```

In addition DISKATE has a statement for defining a field of ASCII constants which is not standard. The pseudo ASC will define the string of bytes given by the operand. The string is NOT delimited by quotes, but by blanks. The string may also be terminated by the carriage return at the end of the statement line. For instance, to define a constant which is the ASCII string ´ERROR´ you could use the statement:

PSEUDO ASC

DEFINES STRING GIVEN BY OPERAND

```
ASC ERROR
```

OPERAND DELIMITED BY BLANKS

There are two special rules to keep in mind with ASC. First, if mnemonic ASC is suffixed immediately (no intervening blank) with a non-alphanumeric character, then ANY PLACE THAT CHARACTER APPEARS IN THE OPERAND IT WILL BE REPLACED BY A BLANK IN THE STRING CONSTANT GENERATED. Thus the statement:

ASCx WITH x = ANY NON-ALPHANUMERIC REPLACES x WITH BLANK IN STRING GENERATED

```
ASC# A#B#C
```

defines a 5 byte constant whose value is the string:

```
´A B C´
```

Second, the character "^" (hex 5E) WILL NOT BE GENERATED into the string constant but WILL CAUSE THE HIGH ORDER BIT OF THE PREVIOUS BYTE TO BE SET. This can be used to define a symbol table in the same format DISKATE uses for the symbol table it generates itself. For instance, to define a short symbol table called SHORTBL with entries for program labels ALHPA, BETA, and GAMMA in Intel format you could use the statements:

^ IN OPERAND NOT GENERATED, SETS HIGH-ORDER BIT IN PREVIOUS CHAR

```
SHORTBL: ASC ALPHA^
DW ALPHA
ASC BETA^
DW BETA
ASC GAMMA^
DW GAMMA
DB Ø
```

There is a special pseudo, IF, which allows for conditional assembly. The form of the IF statement, not counting the label field, is:

IF PSEUDO, EXPRESSION COMMA LABEL

```
IF expression,label
```

The operand thus consists of expression and label with a comma between BUT NO BLANKS. (A blank would prematurely terminate the operand field.) DISKATE evaluates the expression, and if it is not Ø, the assembly continues as it normally would. However, IF THE EXPRESSION HAS

SKIPS TO LABEL IF EXPRES-

THE VALUE Ø, THEN THE ASSEMBLY SKIPS AHEAD TO THE STATEMENT WITH LABEL label. You can use this to have several variants of an assembler program that share most of their code.

SION HAS VALUE Ø

Last, appropriately, is the pseudo-op END, which is used to terminate an assembly. The END pseudo is not really necessary, since the Ø at the end of the last file in an assembly will also work to terminate the assembly. One use for the END pseudo is if you should want to incorporate an assembly source program and one or more edit macros into the same file.

END PSEUDO

ENDS ASSEMBLY

Now let's focus on how to invoke and use the assembler. A strong advantage of DISKATE over many other assemblers is that the two passes of the assembler may be invoked separately. There are three pertinent assembler commands in DISKATE. The A command, for Assemble, will invoke both passes of the assembler, while the A1 command will invoke only Pass 1 and the A2 will invoke only Pass 2. There are two ways for giving arguments for the various A commands. IF NO ARGUMENT IS GIVEN, THEN THE ARGUMENT IS THE CURRENT FILE. The command A all by itself will simply assemble the current file. On the other hand, you can supply as an argument A LIST OF FILE NAMES SEPARATED BY BLANKS. When given such a list, the assembler will assemble the program logically equivalent to the concatenation of the files given in the list, though the files are not actually concatenated. The files in the list can be pieces of one large program, or separate subroutines. When a list of files is given as the argument, the source area is left entirely free for other use. The files are processed through a special disk buffer within the internal area occupied by DISKATE. (In Part II we'll discuss where this buffer is.) The file list can be given to any of the assembler commands, A, A1, or A2. You can also include in a file list the symbol @ in place of a file name. When the assembler encounters this it will execute a PAUSE command, and you can use this pause to change diskettes in case a source program is too long to fit on one diskette.

A COMMAND

INVOKES BOTH ASSEMBLER PASSES,

A1 PASS1 ONLY, A2 PASS2 ONLY

NO ARG USES <F>

ARG IS LIST OF FILES SEP. BY BLANKS

@ AS FILE NAME CAUSES PAUSE

Until you are used to the assembler, BEFORE giving a command to invoke the assembler you should make sure you have taken care of the following points:

1. Check to make sure & and $ will be set properly, either from the terminal, or preferably from within the source code by ORG or AORG and SORG.

2. If this is not the first assembly during the run, you will have to give a Z or Z> command to prevent labels from being doubly defined.

3. Make sure there will be enough room in the memory for the symbol table.

If you are assembling from disk there is a simple method for allocating memory. This method is used in source programs supplied with DISKATE, and at first sight may seem confusing, so we'll look at it carefully to see why it works. When assembling by giving a list of disk files, there is no need for a source area at all. However, you will need: 1. room for the symbol table, and 2. room for the object code. How do you allocate this division? One method is to let the assembler do it for you by giving the following pseudo-op in your source program:

```
SORG T^+10
```

This statement says to put the object code 10 bytes past the end of the symbol table. (The 10 byte cushion is just to play it safe.) But, as your program is assembled, during Pass 1 the size of the symbol table is expanded considerably. How does this statement not result in having the symbol table clobber the beginning of the object code as more and more symbols are defined? The answer comes from an understanding of the separation of function of the two passes of the assembler. All symbols are defined during Pass 1, but no code is generated and the storage counter is not changed. The SORG statement DOES NOT TAKE EFFECT UNTIL PASS 2. Thus by this time the symbol table has been completely compiled, so the object code can safely be placed just beyond it. If you use this method, though, you must save the object code on disk before going on to assemble something else. A subsequent assembly definitely will cause the symbol table to begin overwriting the area of object code for the first assembly.

There are a number of commands which are related to the process of assembling. Normally the assembler will produce a listing on your terminal of the source and object codes during Pass 2. In many cases, however, you will want to assemble a program without producing the listing. This is accomplished using the Q command, Q for Quiet. The Q command, which takes no argument, will cause the assembler to henceforth refrain from producing a listing during Pass 2. However, during both passes, if the assembler was given a file list for its argument, it will print on the terminal the name of the file currently being processed. Assembler error messages are always printed out regardless of any Q command. The assembler will also print the value of & in the current base on encountering a pseudo INTE or PROS. To cause the assembler to resume printing listings, give the J

SORG T^+10 PUTS OBJECT CODE JUST BEYOND <T> Q COMMAND CAUSES ASSEM- BLER TO QUIT MAKING LISTINGS DURING PASS 1; J COMMAND CAUSES LISTINGS TO RESUME TAB

command, J for Jabber, also taking no argument.

When listings are printed during Pass 2, the source code is printed with a "tab" setting for:

1. The label field
2. The opcode field
3. The operand field
4. The comment field

Line numbers will be printed at the beginning of the field for each source statement. The normal tab settings for these four fields are, repectively, 8 15 20 29. In addition, the entire source code field is offset with respect to the left margin, to allow room for the object code. This offset is normally 21. These values can be changed by giving the TAB command. If the TAB command is given with no argument, the current tab settings will be printed on the terminal, first the four settings for the four fields above, and last the source code offset. To change the settings, give the TAB command with an argument consisting of a list of numbers separated by blanks, the order of the numbers again being the four fields above and lastly the offset. Note that the tab stops for the four fields are each WITH RESPECT TO THE MARGIN + THE SOURCE CODE OFFSET and not with respect to the previous tab stop. In the list of numbers given as argument to a TAB command, the five numbers need not all be given. Any that are omitted will keep their old value.

In editing assembler source programs, you may want to keep the source code in as compressed a form as possible, to save storage. (Note that the Intel format is usually more frugal with storage than the Processor Technology format, since you don't need to begin a line which has no label with a blank.) DISKATE provides an editing command which will print an interval in the same format as that used by the assembler, except there is no object code present, so that the tab stops are relative to the left hand margin rather than the source code offset. The line numbers will also be printed. This is the P command, which takes as argument any of the kinds of intervals discussed above. The P command will format the printout in whichever of the Intel or Processor Technology formats is current. If the source code contains INTE or PROS pseudos, these will NOT cause the P command to change formats. Thus the argument of the P command should not specify an interval that includes code with both formats. The P command WILL respond to INTE or PROS commands given from the terminal, so if the format is wrong one of these commands can be used to change it.

The P command can also be used with ordinary text to

COMMAND

WITH NO ARGUMENT WILL PRINT CURRENT ASSEM-BLER TAB SETTINGS

ARGUMENT IS LIST OF NUMBERS

WHICH REPLACE OLD SETTINGS

OMMITTED SETTING KEEPS OLD VALUE

P COMMAND

PRINTS INTERVAL GIVEN AS ARGUMENT IN ASSEM-BLER FORMAT

obtain line numbers, for referencing with arguments like n!, but the lines will come out looking rather peculiar, since they will be formatted as if they were assembler source statements.

ASSEMBLY ERRORS TERMINATE A COMMAND STRING

There are several ERROR CODES that the assembler will generate if it finds something wrong in your source code. If an A, A1, or A2 command results in assembler errors, the message:

ASSEMBLY ERRORS?

will be printed on the terminal when execution of the command is finished, and if the command was part of a larger command string, such as a macro or multiple-command command line, the command string will abort in the same way as for any other DISKATE error. The meaning of the error codes is as follows:

ERROR CODES

A Argument Error. The operand field is invalid for the given opcode. This will also occur at a statement lbl after an IF pseudo of the form IF expression,lbl if the expression refers to an undefined variable. The A code usually occurs only in Pass 2, though an invalid EQU operand, for instance, will cause it to occur in Pass 1.

M Missing Label. An EQU statement occurs without a label. Printed during both passes.

D Doubly Defined Label. A label is given where the label is identical to a symbol already defined. Failing to give a Z or Z> command after a previous assembly will usually generate quite a few of these.

L Label Error. The first character in a label field is not an upper-case letter of the alphabet. When this occurs, 3 NOPs will be generated in place of the statement. This error can occur especially in Processor Technology format if there are extraneous characters in the source code.

O Opcode Error. The opcode field is not a proper op-code or pseudo-op. 3 NOPs will be generated in place of the statement.

To conclude this section, let's look at a simple example of an assembler program which performs a useful function. We've already seen that DISKATE symbol tables are kept in a format consisting of: (1) the characters of the symbol, with the last character having the high-order bit set; followed by (2) a 16-bit number giving

the value of the symbol; (3) with a Ø at the end of the table. It would be useful to have a simple program which will print out a symbol table. The example below will print on the terminal the entire current symbol table <T>. Because the program uses DISKATE's own internal print routines, all of the usual conventions regarding the panic state, current base etc. apply precisely as if this were a normal DISKATE command.

Some comments about the program. First, it is quite elementary, doing only the minimal in the way of formatting the output, and the symbol table is not sorted. The symbol is printed, then two spaces, then the value of the symbol in the current base. You could make it a bit more elegant by having the symbol printed out in a given column. There are three labels not defined within the program. These are OUT, which prints one character on the terminal, PHLSB, which prints on the terminal in the current base the value in the HL register pair, and SYMTB which is the address of the pointer to the beginning of the symbol table. Where did the addresses come from for these three labels? This question is discussed thoroughly in Part II, but to jump the gun a little bit, your version of DISKATE is provided with a file called ATETBL. This file is an abbreviated version of the symbol table for an assembly of DISKATE, and a complete list of all the symbols in it and their meaning can be found in Part III. This table provides the addresses for a number of internal DISKATE routines and registers. As mentioned above, the table can be made into the current symbol table by the commands:

```
L ATETBL <T>
KEEP=Ø
```

Then to zero all of the new symbols defined during the assembly of this program, the command Z>KEEP can be given. The addresses for OUT and PHLSB have come from this table. It should be emphatically emphasized that in the listing below, these addresses refer to the corresponding locations ONLY IN THE VERSION OF DISKATE CURRENTLY BEING USED BY THE AUTHOR OF THIS MANUAL. IN YOUR VERSION THEY WILL ALMOST CERTAINLY NOT BE THOSE SHOWN IN THE LISTING!! In no case should you take addresses for OUT and PHLSB by reading them off from the object code of this listing. All of this is more fully discussed in Part II.

The program is set to be executed from location 7FØØH. Note that the program sets this initial value for both & and $ at once using the ORG pseudo. You must always be careful when you do this, since the ORG statement preserves the relative offset of & to $, so that in

this case we're assuming no previous assembly gave $ a different value than &. One way of avoiding this situation is to always give separate AORG and SORG statements.

By storing the object code in a file called PRNTSYM, the symbol table can be printed by giving the DISKATE command:

```
GO PRNTSYM
```

which will load the object code, and then execute the equivalent to the command X<R>. The RET at the end of the program will return control to DISKATE. After the assembly, the object code can be stored by the command:

```
S PRNTSYM PRNTSYM..SYMEND
```

Notice that in this command, the first time PRNTSYM is used it is a file name, and the second time it is used it is a symbol in the symbol table. There is no ambiguity here, but if you find it confusing you can use a different name for the file containing the program and an assembler label for the program itself.

Here is the listing of the program, exactly as generated by the assembler. The source code was assembled directly from the disk file PRNTSYMS, and the tab settings were adjusted by changing the last setting, i.e. the source code offset, to 15.

```
>A PRNTSYMS
PRNTSYMS
PRNTSYMS
                    1              INTE
                    2                                  ;
                    3                                  ;
                    4                                  ;  PROGRAM TO PRINT
                    5                                  ;  SYMBOL TABLE
                    6                                  ;
7F00                7              ORG   7F00H
                    8                                  ;
7F00    2A 5B 45    9    PRNTSYM: LHLD  SYMTB          ;  SET POINTER
                    10                                 ;
7F03    7E          11   SYMBEGIN: MOV   A,M           ;  GET SYMBOL
7F04    B7          12             ORA   A             ;  SET FLAGS
7F05    CA 38 7F    13             JZ    SYMEND        ;  QUIT IF 0
                    14                                 ;
7F08    7E          15   SYMLOOP: MOV   A,M           ;  GET SYMBOL LETTER
7F09    47          16             MOV   B,A           ;  SAVE COPY
7F0A    E6 80       17             ANI   80H           ;  CHECK HIGH
                    18                                 ;  ORDER BIT
7F0C    C2 17 7F    19             JNZ   ENDSYM        ;  JUMP IF ON
```

```
7F0F    78          20          MOV  A,B       ;  RESTORE LETTER
7F10    CD 38 35    21          CALL OUT       ;  PRINT LETTER
7F13    23          22          INX  H         ;  INCREMENT POINTER
7F14    C3 08 7F    23          JMP  SYMLOOP   ;  GET NEXT LETTER
                    24                         ;
                    25                         ;
7F17    78          26  ENDSYM: MOV  A,B       ;  RESTORE LETTER
7F18    E6 7F       27          ANI  7FH       ;  STRIP OFF BIT
7F1A    CD 38 35    28          CALL OUT       ;  PRINT OUT LETTER
7F1D    3E 20       29          MVI  A,' '     ;  PRINT OUT A
7F1F    CD 38 35    30          CALL OUT       ;  SPACE
7F22    CD 38 35    31          CALL OUT       ;  AND ANOTHER
7F25    23          32          INX  H         ;  INCREMENT POINTER
7F26    5E          33          MOV  E,M       ;  LOW ORDER
                    34                         ;  SYMBOL VALUE
7F27    23          35          INX  H         ;  INCREMENT POINTER
7F28    56          36          MOV  D,M       ;  HIGH ORDER
7F29    23          37          INX  H         ;  INCREMENT POINTER
7F2A    E5          38          PUSH H         ;  SAVE POINTER
7F2B    EB          39          XCHG           ;  HL CONTAINS
                    40                         ;  SYMBOL VALUE
7F2C    CD E2 34    41          CALL PHLSB     ;  PRINT VALUE
7F2F    3E 0D       42          MVI  A,13      ;  CARRIAGE RETURN
7F31    CD 38 35    43          CALL OUT       ;  PRINT CR,LF
7F34    E1          44          POP  H         ;  RESTORE POINTER
7F35    C3 03 7F    45          JMP  SYMBEGIN  ;  GET NEXT SYMBOL
                    46                         ;
                    47                         ;
7F38    C9          48  SYMEND: RET            ;  RETURN TO DISKATE
                    49                         ;
>
```

Notice that DISKATE will generate line numbers. The reason for having line numbers is to correlate with the error messages. A line number MUST NOT begin a line in the original assembler source code. If you want to use line numbers for your own reference in the source code, and the P command will not suffice, the line numbers must be part of the comment field. The format in this example, of course, is Intel. By way of contrast, here is how the original source code looks:

```
 INTE
;
;
;  PROGRAM TO PRINT
;  SYMBOL TABLE
;
ORG 7F00H
;
PRNTSYM:  LHLD SYMTB  ;  SET POINTER
;
```

```
SYMBEGIN:  MOV  A,M  ;  GET SYMBOL
ORA  A  ;  SET FLAGS
JZ  SYMEND  ;  QUIT IF Ø
;
SYMLOOP:  MOV  A,M  ;  GET SYMBOL LETTER
MOV  B,A  ;  SAVE COPY
ANI  8ØH  ;  CHECK HIGH
;  ORDER BIT
JNZ  ENDSYM  ;  JUMP IF ON
MOV  A,B  ;  RESTORE LETTER
CALL OUT  ;  PRINT LETTER
INX  H  ;  INCREMENT POINTER
JMP SYMLOOP  ;  GET NEXT LETTER
;
;
ENDSYM:  MOV  A,B  ;  RESTORE LETTER
ANI  7FH  ;  STRIP OFF BIT
CALL OUT  ;  PRINT OUT LETTER
MVI  A,' '  ;  PRINT OUT A
CALL OUT  ;  SPACE
CALL OUT  ;  AND ANOTHER
INX  H  ;  INCREMENT POINTER
MOV  E,M  ;  LOW ORDER
;  SYMBOL VALUE
INX  H  ;  INCREMENT POINTER
MOV  D,M  ;  HIGH ORDER
INX  H  ;  INCREMENT POINTER
PUSH H  ;  SAVE POINTER
XCHG  ;  HL CONTAINS
;  SYMBOL VALUE
CALL PHLSB  ;  PRINT VALUE
MVI  A,13  ;  CARRIAGE RETURN
CALL OUT  ;  PRINT CR,LF
POP  H  ;  RESTORE POINTER
JMP  SYMBEGIN  ;  GET NEXT SYMBOL
;
;
SYMEND:  RET  ;  RETURN TO DISKATE
;
```

Obviously, the assembler listing is far more intelligible, whereas the compressed format of the original source code saves on storage. The listing generated by the P command is identical to that given by the assembler, except that it includes no object code. (Of course it can happen that a line which is too long in the assembler listing will not be in the P command listing.) One note. The program assumes that the Processor Technology format might be in force before it is assembled, so the INTE pseudo begins in column 2 to indicate an empty name field. If the Intel format is in force then the lack of a colon automatically indicates an

empty name field.

## I.5 Invoking DISKATE

TO POWER UP

In this section we will discuss the various ways of invoking DISKATE, including the power-up procedure. It might seem like a dirty trick to have put off the power-up procedure until so late into the manual, but there is a good reason for it. When DISKATE initializes it issues internally commands that are exactly like some you might give yourself from the terminal, so it is important to have an understanding of how they work.

LOAD & EXECUTE IO

VERSION PROVIDED BOOT-STRAPS IO

First a word about how DISKATE is organized. DISKATE actually consists of two separate but linked programs: ATE and IO. ATE consists of everything but the I/O routines, and is completely independent of any particular I/O or disk configuration. IO contains the I/O routines, including the disk drivers, and also contains a loader which loads ATE. To bring up DISKATE, all you do is instruct your system to load and execute IO. If your disk will allow any program to be bootstrapped, as does the Disk Jockey, you will probably want it to bootstrap IO. This is the way DISKATE is provided. If you are using the North Star disk then DISKATE can be brought up with the DOS command GO IO. There is one very important point here if you go to create another copy of DISKATE on another disk. IO ASSUMES THAT ATE IS THE SECOND ENTRY IN THE DIRECTORY. If you use DISKATE to create another copy of itself, you must make sure that this holds.

ATE MUST BE 2ND ENTRY IN DIREC-TORY

ON POWER-UP COMMAND GO STARTUP IS EXECUTED

When IO has loaded ATE, it will transfer control to it. Initialization then takes place. This includes executing the command string:

```
O,Z,Y
```

The last thing DISKATE does at power-up is to execute the command:

```
GO STARTUP
```

STARTUP PERFORMS USER´S INITIAL-IZATION

STARTUP as provided is an ordinary source file. To see what it contains in your version, type O,L STARTUP,".. The STARTUP file can be used to initialize your own variables which you want to have a value every time you use DISKATE. However, since the GO command will either L and D a source file or L and X a non-source file, the STARTUP file can also be a machine language program.

ENTRY POINT TO INITIAL-

If you create a new copy of DISKATE, you might happen to forget the STARTUP file. This is not serious. In this case DISKATE will issue an error message that it can´t find STARTUP, and then take commands from the terminal as usual. You can edit STARTUP just as you

would any other file.

IZE IS

Now suppose you want to leave DISKATE for another program, such as a DOS, which will be co-resident with DISKATE, and then return to DISKATE. What entry point should you use? There are two. If you give an I command for the disk containing DISKATE, the address for the file ATE will give you the entry point which causes initialization. You can jump to this address from a front panel, DOS, or a piece of software. To re-enter DISKATE without full initialization, the entry point RENT as given in ATETBL can be used. To find out its value, mount the disk provided and type the command:

LOAD ADDRESS OF ATE

REENTRY POINT GIVEN BY SYMBOL RENT IN ATETBL

```
L ATETBL <T>,KEEP=0,?RENT
```

IO PUTS JUMP TO LOADER AT MEMORY LOCATION 3

In DISKATE release 1.0 this entry point is 12 bytes beyond the beginning of ATE, though this might change in the future. Jumping to this entry point WILL initialize the stack, but will not change the pointers to <S>, <F> etc. In Part II we'll discuss how a machine language program can call DISKATE as a subroutine and have DISKATE return to it. If you jump to RENT in such a situation, the stack in the program from which you jump will be lost.

TO RELOAD DISKATE JUMP TO 3 IF IO INTACT

You might want to reload DISKATE from the disk, in case something serious has gone wrong. IO will write at memory location 3 a jump to IO. Since IO begins with the loader for ATE, you can reload DISKATE by jumping to memory location 3, so long as the area of memory occupied by IO is intact. To do this from within DISKATE, you can give the command:

```
X3
```

In addition to these entry points, DISKATE provides some recursive features whereby it can be invoked from within DISKATE. These commands allow a macro to issue a prompt, accept a DISKATE command, and then resume. The command COM, for command, takes as argument a string enclosed in brackets. When this is executed, the string is printed on the terminal as a prompt, replacing the normal DISKATE prompt. You can then type in one "command line", which may include a multi-line E command. When the command has been executed, execution of your macro will resume. Here's an example of how this can be used. Suppose in loading files you want to have a way to prevent yourself from giving an L command without first having given an O command. That way you will avoid clobbering the wrong part of the memory. You can do this by having the following macro tucked away in an out of the way place in the memory:

COM COMMAND

PRINTS STRING GIVEN AS ARGUMENT FOR A PROMPT, ACCEPTS ONE COMMAND

```
O,I,COM[TYPE LOAD COMMAND],"..
```

The best way to invoke the macro is to set a variable equal to the address of the macro, then give the command D variable. Note that the macro can be loaded and the variable set to its location by the STARTUP file. In using this method you must be very careful that the macro DOES NOT RESIDE IN THE SOURCE AREA, since the O command in the macro collapses the source area.

Using the COM command can be slightly tricky, since you can easily get confused about whether you are talking to the terminal at the "ground level" or via a COM command. If you give an ESC to the command line being typed in response to a COM command, DISKATE expects the line to be retyped. On receipt of the line, whatever process contained the COM command continues. Thus typing ESC to a command expected by the COM command DOES NOT return you to the ground level. If you want to return to the ground level when DISKATE expects a command line for a COM command, you can give the command RENT, for Reenter. This command performs a jump to the DISKATE reentry location. The stack will be reinitialized, but all of your variables will keep their old values.

Note that the COM command will accept only a single command, and then returns to the process that invoked it. There is a similar command, ATE, which when executed CALLS DISKATE RECURSIVELY. The ATE command allows you to give DISKATE any number of commands, precisely as if you were talking to the "ground level" DISKATE. An error will cause the error message to be printed, but you will NOT be returned to the ground level. Instead you will still be within the invokation of DISKATE which is a subprocess of whatever process contained the ATE command. Then when you give the BYE command, DISKATE returns to the process that invoked the ATE command. As with COM, ATE takes as argument a string enclosed in brackets which it prints on the terminal as a prompt. The ATE command can be used to create macros which handle various functions automatically, but where you want to leave some "holes" for doing work with an indefinite number of DISKATE commands yourself. Here's an example of how you might use this. Consider the following macro:

```
R-1,^<F>|>+1,ATE[POINTER IS NOW AT END OF FILE],S " ..
```

Suppose you begin your editing work by loading a file and then invoking this macro. The macro will make certain that you start to work with the entry pointer at the end of the file. DISKATE will issue the prompt, and then you can give any number of editing commands. Then,

RENT COMMAND

JUMPS TO REENTRY POINT (GROUND LEVEL) INITIALIZES STACK ONLY

ATE COMMAND

PRINTS AS PROMPT THE STRING GIVEN AS ARGUMENT,

ACCEPTS ANY NUMBER OF COMMANDS

RETURNS WHEN GIVEN:

BYE COMMAND

you give the BYE command to resume with the macro. This will automatically store the file on the disk, restore the entry pointer to the end of the file, and then issue the prompt and wait for more commands. As long as none of the editing commands you type in response to the ATE command changes the latest file reference, this macro will allow you to do any editing work you want, and will automatically handle storing the file on the disk and restoring the entry pointer. If properly written, such macros can decrease the likelihood of catastrophic errors significantly.

BYE COMMAND AT GROUND LEVEL RETURNS TO CALLING PROGRAM

One final point. Although this is discussed in detail in Part II, we must note that DISKATE can be called by other machine language programs. In addition to the jump to IO written at memory location 3, a jump is written at location 0 to a special DISKATE entry point called ATECOMS. This entry point will cause DISKATE to be called as a subroutine, with the HL register pair pointing to a string of DISKATE commands. This string will be executed, and then control will be returned to the calling program. In this case if this string does not cause DISKATE to leave the "ground level", then a BYE command in the command string will cause control to be returned to the calling program. (Control will also be returned to the calling program if a 0 indicating the end of the string is encountered.) Note however that invoking the entry point RENT, either by a RENT command or by jumping to the entry point RENT, will cause reentry at the ground level and the return address to DISATE's caller will be lost. If you are at the ground level and there was no calling program, i.e. you invoked DISKATE by the usual power-up procedure, then the BYE command will cause DISKATE TO BE RELOADED FROM THE DISK.

IF THERE IS NONE CAUSES DISKATE TO BE RELOADED FROM DISK

RENT LOSES RETURN ADDRESS

# Part II Installation and Maintenance

## II.1 Bringing up DISKATE

BAUD RATE FOR SERIAL TERMINAL ON D.J. IS 1200

The version of DISKATE supplied with the Disk Jockey controller comes ready to run with a serial terminal attached to the Disk Jockey serial I/O port. If you have a serial terminal, attach it to this serial port and set the terminal to run at at 1200 baud. Then all you have to do is bootstrap the DISKATE diskette and you should be up and running. If you have a serial terminal and want to run it from another I/O board, the best thing to do is bring up DISKATE from the serial terminal on the Disk Jocky, then use DISKATE itself to customize your I/O routines. If your terminal hardware is not simply a serial terminal, or if it is a serial terminal but won't run at 1200 baud, then you will have to make some minor patches. Before describing the patching procedure in detail, here is the general structure of how DISKATE is brought up. As mentioned in Part I, DISKATE consists of two independent but linked modules, IO and ATE. IO contains all of the I/O drivers including those for the disk. It also contains a loader which looks in the directory for ATE, which IT ASSUMES IS THE SECOND ENTRY IN THE DIRECTORY, and loads it. IO writes a jump to the beginning of IO at memory location 3. This jump is the only link ATE has to IO. Thus ATE and IO can be positioned in the memory completely independently.

DISKATE READY TO RUN WITH SERIAL TERMINAL CONNECTED TO D.J.

ATE MUST BE 2ND FILE IN DIRECTORY

The beginning of IO is a JUMP TABLE to all of the I/O drivers DISKATE will need. The initial power-up procedure if your terminal system is not connected to the Disk Jocky serial port is to bootstrap as if you were, and then to patch into this jump table. We'll discuss the first part of the jump table first, because the first part deals with the terminal I/O and the second part deals with the disk drivers. Here is the format for the beginning of IO:

IO BEGINS WITH JUMP TABLE

1ST JUMP TO LOADER, THEN TERMINAL I/O DRIVERS

| Location | | Contents |
|---|---|---|
| IO | JMP | Loader |
| IO+3 | JMP | Character Output Routine |
| IO+6 | JMP | Character Input Routine |
| IO+9 | JMP | Terminal Initialization Routine |
| IO+12 | JMP | Panic Detect Routine |

In the standard version currently provided, IO begins at 0100H.

A jump to the loader begins the jump table. Thus DISKATE can be reloaded at any time by jumping to IO.

Since a jump to IO is written at memory location 3, the easiest way to jump to IO to reload DISKATE is simply to jump to memory location 3.

The general procedure for getting started with a terminal either not connected to the Disk Jockey or a serial terminal such as a TTY that won't run at 1200 baud is as follows:

1. Bootstrap in the diskette anyway. The system will not respond, but that doesn't matter.

2. Stop the machine, and using a front panel or monitor, install your own I/O routines in the computer's memory. The guidelines that these routines must satisfy are given below. A safe place to put them when you first bring up DISKATE is the 256 bytes from 2900H to 29FFH. Once you have DISKATE running you can use it to incorporate your I/O routines within IO. If your I/O routines won't fit in 256 bytes, use the top of the memory available.

DURING FIRST STARTUP I/O ROUTINES CAN GO AT 2900H.. 29FFH

3. Again using a front panel or monitor, change the jumps in the jump table to jump to your I/O routines. Be sure to remember that the addresses of the jumps must be stored with the low order byte first.

4. Cause your system to begin executing instructions at the beginning of IO. This should be 0100H (001:000Q), unless your version of DISKATE comes with a corrections sheet indicating a different starting address. The module ATE will be reloaded from the disk, and at this point a start-up message should appear on the screen and then the prompt symbol, ">".

AFTER PATCHING JUMP TO 0100H UNLESS OTHERWISE INDICATED ON CORRECTION SHEET

5. Once DISKATE is running, the source code for IO is contained in the disk file SYSIO. This should be reassembled to contain your own I/O routines, so that they will automatically be loaded when DISKATE is bootstrapped. If your I/O routines are in PROM or otherwise manage to find their way into the memory without needing to be loaded within IO, IO should be reassembled so that the jump table contains the proper jumps.

TO RUN SERIAL TERMINAL FROM D.J. PORT AT

The procedure for connecting a serial device that won't run at 1200 baud, such as a TTY, to the Disk Jockey serial port, is a special case that is somewhat simpler. In this case the only thing that will have to be patched is the instruction which loads the baud rate constant for the serial I/O port. Here is what you do:

OTHER THAN 1200

1. Obtain the address of the terminal initialization routine from the jump table. The low order byte of this address is at IO+10 and the high order byte at IO+11.

PATCH SPEED CONSTANT LOAD IN TERMINAL INITIAL-IZATION ROUTINE

2. Unless otherwise indicated in a corrections sheet, the first byte of the terminal initialization routine will be a NOP, or 0. The next instruction will be:

| mnemonics | octal | hex |
|---|---|---|
| LXI H,041Q | 041 041 000 | 21 21 00 |

ADDRESS OF THE ROUTINE AT IO+10

This is the instruction which loads the initial speed constant for the Disk Jockey serial port. Consult the Disk Jockey manual to find the speed constant for the baud rate you want, then patch it in in place of the 041 000 octal or the 21 00 hex in this instruction. Be sure to patch only the address portion of the instruction and not the opcode, and be sure to enter the speed constant as a 16 bit address, low order byte first.

3. Jump to the beginning of IO. There is no need to patch anything in the jump table.

Once you have DISKATE up for the first time, the next thing you will want to do is to customize your I/O routines, unless you should happen to want to run from the on-board serial port at 1200 baud indefinitely. We've seen that the module IO begins with a jump table, and that the four jumps after the first jump to the loader are for the terminal I/O routines. Here are the guidelines you will need for your own terminal routines.

1. All of the four terminal I/O routines will be passed a number in the A register which will serve as a DEVICE NUMBER. Initially this device number is 0, and is changed by an IO command. If you don't have multiple I/O devices this feature can be ignored. If you have both a video terminal of some kind and a printer, for example, then you could have the output routine check the A register for 0, branching to the printer driver if the A register is not 0. Note that the IO command will cause its argument to be supplied to all the terminal I/O routines in the A register. Thus your character input routine should always go to the keyboard, unless you have more than one keyboard. You can have up to 256 different devices -- the maximum number of codes that can be contained in the A register.

DEVICE NUMBER IN A REGISTER

CHAR OUT PRINTS

CHAR IN B REGISTER

2. The character output routine should print on the terminal the character supplied in the B register. Be very careful not to make the mistake of printing the character in the A register -- remember this is the device number.

CHAR IN PUTS

CHAR IN A REGISTER

3. The character input routine should return the input character in the A register, replacing the device number.

4. The terminal initialization routine is executed only in response to the Y command and at power up. If you have multiple I/O devices you may want to initialize all of them in response to any call to the initialization routine, or you may want to initialize only the device whose number is in the accumulator. If you have this routine initialize only the device whose number is in the accumulator, then you will have to remember to issue a Y command before using any devices other than Ø. The IO command will issue a carriage return and line feed to the device given as argument. Thus if you initialize only the device whose number is in the A register, the first time you give an IO command for a non-zero device this carriage return and line feed will be passed to the character out driver for a device which has not been initialized. This will not cause a problem if the driver for this device simply loses the carriage return and line feed, but you should be careful that it doesn't cause an error. You may find it more convenient to initialize all devices at one time.

PANIC DETECT RETURNS WITH Z FLAG ON IF PANIC, OTHERWISE OFF

SHOULD NOT WAIT FOR INPUT

5. The panic detect routine should inspect your keyboard for whatever condition you want to use to indicate that a process should be interrupted. This can be a special character having been pressed, such as control-C, or any key having been pressed. The panic routine should return with the Z flag on if it determines a panic stop, and with the Z flag off otherwise. The device number is in the A register. If you have a printer and a video terminal, and the printer is a TTY or Decwriter or the like that has a keyboard of its own in addition to the keyboard on your video terminal, you will have to decide whether to have the panic routine consult the keyboard for the device in the A register, or always consult your terminal keyboard. Note that the panic detect routine could also consult a front panel sense switch if you have one, but this is usually less convenient than inspecting the keyboard.

The panic detect routine MUST NOT WAIT for a keyboard input, but should simply inspect to see if a panic is being requested. For instance, if you have a serial

terminal being run from an I/O board with a UART, and one port serves as a status port, issue an input to the status port to see if input is ready. If not return to the calling program with the Z flag off. If the proper status bit indicates there is an input present, take the input if you want to use it to determine whether or not to interrupt the process, but your routine should not loop in case there is no input present.

The panic detect provided in SYSIO will respond to any key being pressed, if it can detect it. Because there are timing considerations involved in sampling the input line with the "software UART" on the Disk Jockey, it's best to request a panic stop by hitting the break key on your terminal if it has one, since this transmits Ø's continuously.

6. In all of the terminal I/O routines the only register that need be preserved is the stack pointer.

I/O PROTOCOL COMPATIBLE WITH NORTH STAR DOS

If you are using DISKATE with a North Star Disk, the protocol above is compatible with that for the North Star DOS I/O routines, though less restrictive. (You don't have all of their requirements to preserve the contents of registers.) Thus for North Star users the jumps in the IO jump table can go directly to the North Star I/O vectors.

WHEN IO REASSEMBLED IF TOO BIG FOR SAME SPOT, MUST BE RELOCATED BY CHANGING THE AORG

When you go to reassemble IO if this is necessary, you can give an I command to the DISKATE diskette to find out where in the memory ATE and IO will go. The memory address listed in the directory printout gives the address, and the size in blocks is also given. If ATE follows IO immediately in memory, then you must be sure that your reassembly of IO will not result in a module to big to fit. After an assembly of IO, the command:

```
?IOEND-SYSIO
```

will tell you how big IO is. If the number of blocks needed to contain it is more than the number listed in your directory printout, you may have to change the AORG statement in SYSIO and reassemble so that IO goes somewhere else in memory. This will not affect ATE, since ATE picks up the location of IO by the jump at memory location 3.

The I/O system in DISKATE also allows you to customize the drivers for the disk. In most cases you would not need to do this, unless DISKATE were being used with some other kind of hardware, but you might want to consider this for certain special applications. For instance, if you wanted to include on diskettes files which you would not want DISKATE to touch or even

know about, this could be achieved by rewriting the DISKATE disk drivers to only begin reading the disk at a particular block. Or, to interface DISKATE to someone else's operating system, a single very large file could be created for each diskette within this operating system, and then the DISKATE disk drivers could call direct access reads and writes to this file through the operating system. So, the protocol for the disk drivers will be given, even though in many cases you won't need to use it.

There is one important way in which YOU MUST CUSTOMIZE THE DISK DRIVERS. The version supplied assumes that there actually exists a disk drive for all of the possible disk drive designators. You should alter the jump table to indicate that a drive is not present in your system, so that if you try to access a file on a non-existent drive an error will be returned.

The format for that part of the jump table for the disk drivers is:

| Location | Contents | | |
|---|---|---|---|
| IO+15 | MVI | C,<code for drive A> | MUST SUBSTI- |
| | JMP | <driver for drive A> | TUTE: |
| IO+20 | MVI | C,<code for drive B> | |
| | JMP | <driver for drive B> | STC |
| | | | RET |
| IO+25 | MVI | C,<code for drive C> | |
| | JMP | <driver for drive C> | FOR THE |
| | | | MVI |
| IO+30 | MVI | C,<code for drive D> | JMP |
| | JMP | <driver for drive D> | |
| | | | FOR EACH |
| IO+35 | MVI | C,<code for drive E> | DRIVE |
| | JMP | <driver for drive E> | THAT |
| | | | DOES NOT |
| IO+40 | MVI | C,<code for drive F> | EXIST |
| | JMP | <driver for drive F> | |
| IO+45 | MVI | C,<code for drive G> | |
| | JMP | <driver for drive G> | |
| IO+50 | STC | | |
| | RET | | |

The pair of instructions STC, RET at IO+50 is in lieu of a driver for drive H. All of the disk drivers return with the carry flag set in case of error. When you reassemble SYSIO, then, you should substitute a STC, RET for the MVI, JMP for each drive that does not exist in your system. This way any access to drives that

don't exist will properly signal an error. Be sure that when you do this, the name fields are preserved. If you are running DISKATE with only North Star disks, then the loader part of SYSIO must also be changed so that all of the calls to DRIVEA go to DRIVEE.

The protocol for the disk drivers is as follows.

When the disk drivers are called, the B reister contains a code giving the disk command. The codes are:

B=3: The disk drive should be initialized. This is given when a disk is first accessed after power-up, and also after a disk error. If initialization is not required, this should simply return. Note that this command is given to initialize the disk DRIVE and/or drivers, rather than to initialize a diskette. (For the Disk Jockey, diskettes are initialized by invoking the program DISKINT.) Nothing should actually be written on the diskette when this command is executed. Usually the routine executing command will want to be sure it knows where the head of the disk is. This can be achieved by moving the head to track 0.

B=2: The size of the disk in blocks is to be returned in BC. For the Disk Jockey this is 1000. (The drivers supplied for the Disk Jockey leave the bootstrap record for controller routines untouched. This record is not included in the 1000 blocks, and as far as DISKATE is concerned is not even there.) For the North Star disk this number is 350. The maximum number of blocks DISKATE can handle is given by the constant MAXBS which is a symbol in ATETBL. (The use of ATETBL is discussed in the next section.) If this command returns with a value larger than MAXBS, a disk error will result. Each block is 256 bytes long, regardless of how the controller physically organizes disk records.

B=1: A read operation. A contains the number of blocks, DE contains the beginning memory address to which the information is to be transferred, and HL contains the beginning disk address in blocks. The directory occupies blocks 0-3. The drivers supplied for the North Star Disk invoke only the North Star DOS routine DCOM and not DLOOK.

B=0: A write operation. The registers A, DE and HL follow the same convention as for the read operation.

For all disk commands, a return with the carry flag set is the indication of a disk error.

If you should want to rewrite the disk drivers for your own applications, all DISKATE disk addressing is relative to your drivers. Thus if your drivers examine only a certain portion of the disk, leaving the rest beyond the reach of DISKATE, DISKATE's block n refers to the n'th 256 byte block within the area of disk you've set aside. The same would apply for drivers that work through an operating system's direct access file routines. In most cases, however, you will probably not need to touch the disk drivers themselves, and the only customizing needed as far as the disk is concerned is to maintain in the jump table the proper indication of non-existent drives.

Let's go review now the step-by-step procedure for getting DISKATE running for the first time.

1. Bootstrap in the DISKATE diskette, with a serial terminal at 1200 baud connected to the Disk Jockey serial port if this is possible. If not, boot anyway and wait for the disk to become quiet. Even if you intend to run at 1200 baud from the serial port, you should make a backup copy of DISKATE, so you will need to do some of the steps below.

2. Put the patches in the jump table for jumps to your I/O routines if needed. If you want to run from the serial port at a slow speed, patch in the initial baud rate constant. The STC, RET for non-existent drives can also be patched in if needed.

3. If you have needed to make patches, force the computer to jump back to the beginning of IO. DISKATE should now respond.

4.

| single drive | dual drive |
|---|---|
| type:<br>GO DSKINT | Mount a fresh diskette on drive B and type: |
| Mount a fresh diskette. | GO DSKINT |

DSKINT will ask which disk is to be initialized.

5.

| | |
|---|---|
| Remount DISKATE diskette, type: | type: |
| L IOTBL <T>,KEEP=0 | L IOTBL <T>,KEEP=0 |

6.

Remount new diskette, type:

```
S IO SYSIO..IOEND
```

type:

```
S IO:B SYSIO..IOEND
```

7.

Remount DISKATE diskette, type:

```
L ATETBL <T>,KEEP=0
```

8.

Remount new diskette, type:

```
S ATE BEGIN..END
ZKEEP
S ATETBL <T>
```

type:

```
T ATE ATE:B
T ATETBL ATETBL:B
```

9.

Remount DISKATE diskette, type:

```
L IOTBL <T>,KEEP=0
```

10.

Remount new diskette, type:

```
ZKEEP
S " <T>
```

type:

```
T IOTBL IOTBL:B
```

11.

Remount DISKATE diskette, type:

```
O,L STARTUP
```

12.

Remount new diskette, type:

```
S " ..
```

type:

```
T STARTUP STARTUP:B
```

At this point your new diskette will have a copy of the DISKATE system in which the patches to the IO jump table and/or new Disk Jockey serial port baud rate will have been installed. If you need to reassemble IO to include your own I/O routines, continue on. Be sure to change those entries in the jump table for disk drivers so that there is a STC, RET for each non-existent drive.

13.

| | |
|---|---|
| Mount DISKATE diskette, type: | type: |
| O,L SYSIO | O,L SYSIO |

14. Edit SYSIO using the DISKATE editor commands to install your I/O routines in place of those there. Change the AORG statement to relocate IO in case your I/O routines make IO large enough to conflict with the space occupied by ATE.

15. Type: Z,A

If the assembly has errors go back and correct with the editor.

16.

| | |
|---|---|
| Remount new diskette, type: | type: |
| S IO T^+10..<+IOEND-SYSIO | S IO:B T^+10..<+IOEND-SYSIO |
| S IOTBL <T> | S IOTBL:B <T> |
| S SYSIO <F> | S SYSIO:B <F> |

Note that this procedure does not necessarily call for you to transfer every file to the new disk. The rest are discretionary. It's very important, though, that whenever you reassemble IO you also save the symbol table as the file IOTBL.

## II.2 Personalizing DISKATE Settings

When DISKATE is brought up, many DISKATE variables will already have been given values. Because these values serve to allocate the memory, you may wish to change their power-up settings. In working with the assembler, for instance, memory must be allocated for the source area, the symbol table, and the area of memory into which the object code is to be placed. These may require different amounts of area depending on the type of job. For complex programs the symbol table area will have to be quite large, perhaps upwards of several K. On the other hand, when editing source files where there is no question of assembly, the only requirement for a symbol table is to hold variables to be used in commands and macros, so that a symbol table of only 512 bytes or less may suffice. DISKATE provides two ways that your own choice of values at power-up time can be established, and both of them are both simple and systematic.

CAN PERSONALIZE BY SETTING VALUES IN STARTUP FILE

The easiest way to establish your own settings it to include commands for them in the STARTUP file. Suppose you want the memory at power-up to be allocated as follows:

```
3200H..41FFH   Symbol Table
4200H..6FFFH   Source Area
7000H..7FFFH   Object Code Area
```

These settings can be achieved by the following commands:

O COMMAND FOR <S>, &, $ FOR OBJECT CODE, ^adr, M<T> FOR <T>, ETC.

```
^3200H,M<T>
O4200H
&7000H,$&
```

In order for these commands to be executed automatically at power-up, all you have to do is make sure they are included in the STARTUP file. This file can also include such commands as INTE or PROS, Q or J, TAB, WID, and so forth. Since STARTUP is a source file it is easy to change: just type O,L STARTUP and then edit it like any other source file, then type S " <F> when you are finished. When STARTUP is executed at power-up time, it may be anywhere on the disk. The advantage of concentrating your personalization in the startup file is that it's easy to change, and you can initialize your system easily by typing O,GO STARTUP. Because STARTUP will most likely be a very small file, you may find it convenient to include a copy on each diskette.

CAN

In some instances this may be inconvenient, or

won't even work at all. For instance, if you want a drive other than A to be the default drive at power up, this cannot be done with a CD command in the STARTUP file, since at power-up DISKATE will get STARTUP from the default drive, which unless you change it will be A. The other method for personalizing values is to set them however you would like, and then save ATE on the disk with these values having been established. Whenever you do this YOU MUST MAKE CERTAIN ATE WILL BE THE SECOND ENTRY IN THE DIRECTORY. Otherwise the loader in IO will load the wrong file. If you save a new copy of ATE, any commands you have given which change internal settings, such as CD, INTE, and the like, will result in DISKATE being brought up at power-up in just the state in which it was saved, unless settings are changed by the STARTUP file or the power-up initial command string O,Z,Y. Thus for instance, if you move the source area and the symbol table, then save a new copy of ATE, afterwards at power-up the source area and symbol table will begin at the same places they did at the time ATE was saved, but the symbol table will contain only the initial assembler variables and the source area will be empty.

PERSONALIZE BY MAKING SETTINGS THEN SAVING ATE

THEN AT POWER-UP ALL SETTINGS WILL BE AS THEY WERE AT THE TIME ATE WAS SAVED EXCEPT AS CHANGED BY O,Z,Y OR STARTUP

To aid in this process you can use the symbols contained in ATETBL. ATETBL is a file on the DISKATE diskette which gives an abbreviated version of the actual symbol table from an assembly of your version of ATE. For the module IO, of course, the entire source code is given. ATETBL includes a large number of addresses which are important for use in machine language programs, which will be discussed in the next section. A listing of the meaning of the symbols in ATETBL is given at the end of Part III. In this section two symbols concern us particularly: BEGIN and END. The interval BEGIN..END, where BEGIN and END are symbols from ATETBL, is the interval in memory occupied by the code for the module ATE. Thus to save a new copy of ATE, here is the procedure:

PROCEDURE FOR SAVING NEW COPY OF ATE

1. Give the command L ATETBL <T>,KEEP=Ø. This will establish ATETBL as the symbol table, with KEEP at the end.

2. Change whatever settings you want to personalize.

3. Give the command S ATE BEGIN..END

Remember, again, that if the copy of ATE is being created on a new diskette, ATE must be the second directory entry. When a file is saved which does not exist in the directory, DISKATE creates a directory entry for it in the first empty slot in the directory. (The I command lists the files in the order in which they occur in the

ATE MUST BE 2ND DIRECTORY ENTRY

directory.) If you are creating a new copy of your system on an initialized diskette, if ATE is the second file saved it will be the second directory entry. If another file is already the second directory entry, you can reclaim the second directory entry by making a copy of the file under another name with the T command, unsaving the file under the orginal name, and then renaming the copy. Also, you should be sure that the power-up version of ATE you create allocates enough memory in the symbol table to hold ATETBL. You can find the size of ATETBL by giving an I command for a diskette containing it. ATETBL is not particularly large, so this should be no problem. Don't make the mistake, though, of thinking you don't need any space at all for the symbol table even if you want to prepare a version only for editing prose text.

SHOULD ALLOCATE <T> BIG ENOUGH TO HOLD ATETBL AT LEAST

If you want to make a copy of DISKATE on a new diskette with new settings, you can follow the same type of procedure already given to make a backup of DISKATE when you first get up and running. Rather than reviewing all of the details of this procedure again, there are a few important points to note. The minimum steps needed to create a new copy of DISKATE are:

1. The new diskette must be initialized.

2. A copy of IO must be saved as the first entry in the directory. (This is not necessary if your system will be bootstrapped from a North Star Disk.)

3. A copy of ATE must be saved as the second entry in the directory.

4. A copy of STARTUP must be saved.

Whenever you reassemble IO you should also update IOTBL so that IOTBL is the symbol table for an assembly of the same code as IO. In this way, IO can be saved by setting the symbol table to IOTBL and typing:

```
S IO SYSIO..IOEND
```

Likewise, ATE can be saved by setting the symbol table to ATETBL and typing:

```
S BEGIN..END
```

## II.3 Interfacing DISKATE to Machine Language Programs

DISKATE provides no less than three different methods for interfacing machine language programs. They can either be called from DISKATE, or they can call DISKATE and execute it as a subroutine. We'll discuss these in order. The easiest method of invoking a machine language program is with the X command, which we've already discussed. Writing a program to be invoked in this way gives great flexibility since the program need not actually be resident in memory at the time it is invoked -- it can just as well be invoked by a GO command. Of course, when you issue a GO command to a machine language program, you must be sure it will not overlay a part of the memory needed for something else. In this section we'll describe how such a program can have access to DISKATE information.

There are 3 subroutines within DISKATE whose addresses are given in ATETBL that can be used in a linkage procedure whereby the same kind of interval arguments used by DISKATE commands can be passed to a machine language program. The structure of the process is this: the machine language program calls a subroutine within DISKATE to determine if an argument is present. If so, another subroutine can be called to evaluate it. The easiest way to link the source code of your machine language program to the addresses of these subroutines within DISKATE is to set the symbol table to ATETBL before assembling your program by the same procedure we saw in the last section. The routine to be called to see if an argument is present is called VCHK. It returns with the Z flag OFF if an argument is present, otherwise the Z flag is on. This allows writing a program which takes special action in the default case of no argument.

CALL VCHK

TO SEE IF ARG PRESENT

Z FLAG OFF IF THERE IS AN ARG

GET ADDR OF VCHK FROM ATETBL

If an argument is present you can call the subroutine called CVALS to evaluate it. It is important that a blank separate the argument and the address of the subroutine in the case of an X command, or the file name in the case of a GO command. CVALS will return with the lower address of the argument in HL, and the upper address in DE, unless an error occurs. In this case CVALS WILL NOT RETURN AT ALL but will exit by jumping to the same error exit used by DISKATE commands which have an error. This exit routine is called WHAT and its address is also given in ATETBL. You could trap all DISKATE errors by patching at the location of WHAT, though this is not recommended except for experienced programmers.

CALL CVALS TO EVALUATE ARG -- RETURNS LOWER ADDR IN HL, UPPER IN DE

JUMPS TO WHAT IF ERROR

For example, you might have a memory test program located at an address in memory which has been set as

the value of a variable MEMTEST, and which takes as an argument the interval of memory to be tested. To test the block of memory from 3000H to 31FFH, the program could be invoked by the command:

```
X MEMTEST 3000H..31FFH
```

(Be sure to leave a blank between the first argument of the X command, which gives the address of the subroutine to be executed, and the second argument, which gives the subroutine's argument.) The subroutine can then obtain the value of the argument by calling CVALS: CVALS will put the lower address of the argument in the HL register pair and the upper address in DE.

What if you want to evaluate an argument and have control passed back to your machine language program in case of an error? In this case you can call VALUS instead of CVALS. This routine works like CVALS, but will return with the Z flag OFF if there is an error in evaluating the argument. There is one important other difference. You must supply to VALUS the addresses of the initial reference interval yourself, the lower address in HL and the upper address in DE. Note that this can be used to have the machine language program determine the initial reference interval and the argument within it passed from the terminal or a macro. You don't need to supply the initial reference interval if you can be certain that an argument will not contain any matching symbols, but this is risky since if the argument contained matching symbols by mistake you could get unintended results.

VALUS WILL EVALUATE ARG & RETURN, Z FLAG OFF IF ERROR. MUST GIVE INIT REF INTERVAL IN HL DE

ARGS MUST BE PRECEDED BY BLANK

You can have multiple arguments, as long as they are separated by blanks. To evaluate further arguments just call CVALS or VALUS repeatedly.

Suppose you wanted a machine language program to follow the same protocol as DISKATE commands and use the last interval computed in case the argument is missing. How do you obtain this interval? Such information as this can be obtained by reading the contents of the relevant internal DISKATE registers. Again this is done by using symbols from ATETBL. In this particular example, the lower address of the last interval computed is stored at the address denoted by the symbol P1, and the upper address at P2. As it happens this example is not really too useful, since the CVALS subroutine will automatically give you the addresses for the previous interval computed if it detects that there is no argument.

GET ADDRS OF DISKATE INTERNAL REGISTERS FROM ATETBL

E.G.
< AT P1
> AT P2

<S> AT BOSAP, EOSAP

If you take a minute to skim through the listing of the meaning of the symbols in ATETBL at the end of Part III, you will see that you have complete access to all of the information available to DISKATE commands. The

<F> AT

source area is the interval from the contents of BOSAP to the contents of EOSAP, and the current file is the interval from the contents of BOFP to the contents of EOFP, for instance. You will need such information to supply an initial reference interval if you call VALUS. Note that reading the contents of internal DISKATE locations in this way is a second method of passing information from DISKATE to a machine language subroutine. If you use this method be careful to avoid mistaking an address like BOSAP, which is the address WHERE THE LOWER ADDRESS of the source area IS STORED, for the contents of the 16 bit area beginning at this address.

BOFP, EOFP

A second method of invoking a machine language program from DISKATE is by using a special feature which we haven't discussed before: the USER COMMAND TABLE. To execute a machine language program by the method above you have to give a command in the form X addr, or GO filename, optionally followed by an argument. By using the user command table you can invoke a command precisely as if it were a DISKATE command. When DISKATE is powered up, it contains a table which it consults whenever a command is given. The format of this table is identical to the format of the symbol table, which we've already seen. However, the symbols in this table are not labels or variables, but are the names of commands created by the user. The "values" for these symbols are the addresses of the machine language programs which are to perform the given commands. Whenever a command is given, DISKATE first scans this user command table to see if there is a command with that name. If so, it calls the program whose address follows the symbol for the command in the user command table. If not, it scans the internal DISKATE command table. Hence if you have any user defined commands, their names will have priority over the DISKATE built-in commands. For instance, if you had a user-defined command called PRINT, the command PRINT would invoke your command instead of invoking the DISKATE P command and assuming RINT is a variable name.

USER COMMAND TABLE

SAME FORMAT AS SYMBOL TABLE

SCANNED BEFORE TABLE OF BUILT-IN COMMANDS

LISTS COMMAND NAME THEN ADDR OF PROGRAM TO PERFORM IT,

When the version supplied of DISKATE is brought up, a small amount of space within the code for the module ATE is allocated for a user command table, and the user command table will be in this space unless you make a new copy of DISKATE with the user command table somewhere else. Of course, to start with there are no user-defined commands, so the initial contents of this space is a Ø indicating the end of the user command table. This space consists of the interval USRCT..ROMEND, where USRCT and ROMEND are symbols from ATETBL. Let's see an example of how to create a user defined command. In Part I we gave a functional but inelegant program called PRNTSYM to print out the symbol

Ø AT THE END

SPACE WITHIN ATE FOR IT AT USRCT.. ROMEND

table. Suppose we want to make this into a user defined command, which to abbreviate we'll call PSYM. Recall that PRNTSYM was assembled to begin at 7F00H, and that to assemble PRNTSYM properly we had to set the symbol table to ATETBL to get addresses for OUT, PHLSB, and SYMTB. Let's suppose that we've just assembled PRNTSYM, that the source code for it is the current file, and that the symbol table has not been zeroed. Let's also assume that there are no user defined commands at this point. To create the user command table, we can give the following commands:

```
N
E[ORG USRCT
ASC PSYM^
DW PRNTSYM
DB 0
]
A
```

(Remember that PRNTSYM sets the assembler format to Intel.) This simple source program will create a user command table with only one entry, PSYM. Now any time we want to print out the symbol table we can give the command PSYM as if it were a built-in DISKATE command. However, for this to become permanent there are two things that must be done immediately. First, we have to save a new copy of the module ATE, so that when DISKATE is brought up the new user command table will be in place. Since the symbols for ATETBL are already in the symbol table, we can do this by mounting a diskette with ATE as the second directory entry and typing:

CAN SAVE NEW COPY OF ATE WITH USER COMMAND TABLE TO HAVE IN PLACE AT POWER-UP

```
S ATE BEGIN..END
```

Since the user command table is located within the space for the module ATE, when DISKATE is brought up from this version, our user command table will be there. The second thing that must be done is to include the command:

COMMAND TO LOAD CODE FOR EXECUTING USER COMMAND SHOULD GO IN STARTUP FILE

```
L PRNTSYM
```

in the STARTUP file, for otherwise the code for the user command PSYM will not be in the memory at power-up, and giving the command PSYM would probably cause a crash.

In general there are three things to keep in mind when you create a new user defined command:

1. The code for the new part of the user command table must be ADDED ON TO THE END of any code that you already have for a user command table, so that existing user

defined commands are not lost. Thus you should always keep on disk a copy of the file which assembles the user command table.

2. After assembly of a new user command table you have to resave ATE.

3. You must make sure you have provided a way at power-up time for the object code for the routine which will carry out the user defined command to be brought into the memory. The easiest way is to include the proper L command in the STARTUP file.

The third method of interfacing a machine language program to DISKATE has a completely different structure. It is possible for DISKATE to serve as a subroutine to be called by a machine language program. The machine language program can pass a string of commands for DISKATE to execute, and if these commands include an ATE command, further commands can be entered from the terminal just as if DISKATE were being executed from the power-up state. To use this method you must NOT call the entry point ATE, since this will initialize the stack and lose the return address. Rather, you call a special entry point called ATECOMS. This entry point assumes that the HL register pair contains the beginning address of a string of DISKATE commands. It will execute the string, then return back to the calling program. If the command string results in an error, the error message will be printed on the terminal but control will still return to the calling program. The one thing you must be careful of is that the command string cannot include a RENT command, nor should this be given from the keyboard during an invokation of DISKATE called by a machine language program, since it will result in the return address being lost.

MACHINE LANGUAGE PROGRAMS CAN CALL DISKATE VIA ENTRY POINT ATECOMS

EXECUTES COMMAND STRING WHOSE ADDRESS IS IN HL

JUMP TO ATECOMS WRITTEN AT MEMORY LOCATION Ø -- CAN BE INVOKED BY RST Ø

To make it easier to use this feature, when DISKATE is powered up a jump to ATECOMS is written at memory location Ø. Thus DISKATE can be called by a machine language program by the following procedure:

1. Load HL with the address of a string of DISKATE commands ending with a Ø. The string can include several command lines.

2. Execute the instruction RST Ø. DISKATE will be called and will execute the command string. Note that it will be using the stack of the machine language program that called it. 100 bytes of stack should be sufficient for most applications.

RETURNS WITH CARRY SET IF ERROR

3. When control is passed back to the machine language program, the carry flag will have been set if an error

has occurred, so your program can check for this case.

Because of the jump to ATECOMS at memory location 0, the RST 0 will automatically cause a call to be vectored to ATECOMS.

One way for your machine language program to pass information to DISKATE using this method is to set the value of internal DISKATE variables. For instance, suppose you want to generate a source file using a machine language program and then operate on it with DISKATE commands. The contents of BOFP and EOFP can be set to point to the beginning and ending of the file, with the contents of BOSAP and EOSAP being set to point to a 0 at the beginning and a 0 at the end of the file. Then when you call DISKATE the current file will be the file generated by your machine language program.

Information can be passed from DISKATE to a machine language program with this method using a special command which we haven't seen before. The V command, V for eValuate, will put the lower address of the argument in HL and the upper address in DE. This command will not achieve anything if given from within DISKATE itself, since the information in HL and DE will be overwritten by subsequent internal routines. However, if the V command is the last command in a string of commands executed by ATECOMS, on returning to the caller HL and DE will have the addresses of the argument. For instance, suppose you wanted the value of the DISKATE variable X. The machine language source statements:

V COMMAND

PUTS LOWER ADDR OF ARG IN HL, UPPER IN DE

ONLY OF USE IN COMMAND STRING TO BE EXECUTED BY ATECOMS

```
        LXI H,COMMAND
        RST 0
LABEL   . . . .
         .
         .
         .
COMMAND ASC VX
        DB  0
```

as part of a machine language program would cause the value of X to be in both HL and DE after the return to the statement LABEL.

The commands in a command string executed by ATECOMS can certainly include disk commands. You can even use this method to link to the value of symbols in ATETBL during the execution of a machine language program, rather than during assembly, by having a command string include L ATETBL <T>,KEEP=0, or the like, and then a V command for a symbol in ATETBL. With this flexibility, virtually any type of interface needed between DISKATE and a machine language program can be constructed.

COPY OF DIREC-TORY LEFT AT 1ST 1024 BYTES OF DISKBUF AFTER DISK COMMANDS

If a machine language program needs access to the information in the directory regarding a disk file, it can be obtained by consulting the DISKATE disk buffer. This buffer begins at the address DISKBUF, where DISKBUF is again a symbol from ATETBL. After an I command, or any other disk command, a copy of the entire directory of the accessed drive will occupy 1024 bytes beginning with DISKBUF. (DISKATE consults the disk-resident copy of the directory for any disk commands -- this copy of the directory is left in memory only for use by user programs.) A machine language program can consult the directory by statements such as:

```
        LXI   H,DIRCOM
        RST   Ø
         .
         .
         .
DIRCOM  ASC#  L#ATETBL#<T>,
        ASC   KEEP=Ø
*
*       ASC#  I#drive-designator  if file not on
*                                 current drive
*
        ASC   VDISKBUF
        DB    Ø
```

Upon returning to the machine language program after the RST Ø, HL will point to a copy of the directory of the current drive. If the directory is needed for a different drive than the current drive, the command string at DIRCOM above should include an I command for the proper drive. The program can page through the directory at DISKBUF entry by entry, testing the first 8 bytes of each 16 byte entry for the desired file name, then reading from memory the desired information.

Note that you can also consult the directory directly from DISKATE. For instance, to get a core dump of the directory type:

```
L ATETBL <T>
KEEP=Ø
I
#DISKBUF..<+1Ø23
```

# Part III: System Reference Summary

## Special input characters:

ESC causes current line to be ingnored, \ echoed. ESC causes escape from the panic state.

BACKSPACE causes just-typed character to be ignored. Backspace, space, backspace is echoed.

S in the panic state causes one character to be printed and the panic state remains in effect.

## Panic State:

A panic detect routine whose address is given by the jump at IO+12 is called periodically, including prior to the output of each character, to determine if a process is to be interrupted. The routine provided will cause a process to be interrupted if any key is pressed, provided it can be detected. The Break key gives the most easily detected signal. A user-provided panic detect establishes its own conditions for determining when a process is to be interrupted. If the process is to be interrupted, the panic state is entered. The panic state is also entered when a PAUSE command is executed or @ is encountered as the file name in a file name list for assembler commands.

When the panic state is entered, the system will wait for an input. If this input is the character S, the process continues but the panic state is automatically still in effect the next time the panic detect routine is to be consulted. Thus S will allow one character to be output during listings. If the character input is ESC, the process is aborted. If the character is anything other than S or ESC the process resumes with the panic state no longer in effect.

## Command Format:

Blanks allowed anywhere EXCEPT WITHIN AN ARGUMENT. A blank is mandatory to separate multiple arguments of one command. Multiple commands separated by commas may be given on a single line.

Number Symbols:

A simple number is given by a string of digits or hexadecimal digits beginning with a decimal digit, preceded by a minus sign for negative numbers, and suffixed immediately by:

| | |
|---|---|
| no suffix | decimal |
| Q | octal |
| H | hexadecimal |

A number can be given in the SPLIT FORM a:b where a and b are simple numbers. a:b denotes 256*a + b. Numbers denote 16 bit values. Negative numbers are represented in two´s complelement form as signed 15 bit integers. Addresses are 16 bit non-negative integers. When a negative number is treated as an address the sign bit will be treated as a digit bit.

Variables:

A variable name may be any length, must contain only upper case letters and digits and must begin with a digit. Variables denote 16 bit values.

Arithmetic Operations:

The arithmetic operations +, -, *, / are performed left to right with the same priority, and all operations are modulo 65536. The / operation gives only the quotient discarding the remainder.

Arithmetic Expressions:

Variables and numbers may combined into an arithmetic expression using the arithmetic operations. Parentheses are not allowed.

Intervals:

An interval is a pair of addresses, with the first address <= the second address. It specifies the interval in the computer´s memory from the lower address to the upper address inclusive. A single number is identified with an interval whose lower and upper

addresses are equal.

Initial Reference Interval:

The initial reference interval is determined by:
1. The command, and
2. The argument of the last REF command if one has been given.

The initial reference interval for D, G, and F commands is the source area. For all others the initial reference interval is the current file if no REF command was given or the argument of the last REF command was empty; otherwise the initial reference interval is the argument of the last REF command given as applied to the current file.

Matching Interval Symbols:

[text] matches the first occurrence of text within the initial reference interval. text may contain any ASCII characters including non-printing control characters.

#sequence of numbers separated by blanks#

matches the first occurrence in the initial reference interval of the sequence of bytes whose codes in the current base are given by the numbers. The numbers must be positive, <= 255 and should NOT have a Q or H base suffix.

_ matches the first carriage return in the initial reference interval

@ matches the first character in the initial reference interval

. matches anything in the initial reference interval. (Any number of consecutive dots can be used.)

Non-Matching Interval Symbols

exp an arithmetic expression specifies the address given by the value of the expression. Expressions are treated as non-negative 16 bit numbers.

<F> denotes the current file. See below under Memory Files.

<S> denotes the source area. See below under Memory Files.

<T> denotes the symbol table. See below under Symbol Table.

<R> denotes the last record read from disk. See below under Disk File Reference Symbols.

S^ denotes the address next after the end of the source area

T^ denotes the address next after the end of the symbol table

^ denotes the address given by the entry pointer. The character whose address is the value of ^ is called the target character. The value of ^ is given by the 16 bit number stored at CHPTR where CHPTR is a symbol from ATETBL.

< denotes the lower address of the last interval computed. This is assigned a value as soon the lower address is determined, even if the upper address has not yet been determined. The value of < is given by the 16 bit number stored at P1 where P1 is a symbol from ATETBL.

> denotes the upper address of the last interval computed. Assigned a value when the complete argument is evaluated or | is encountered. The value of > is given by a 16 bit number stored at P2 where P2 is a symbol from ATETBL.

? denotes the address of the last error in a macro or command line, or the point at which printout from a " command was aborted via panic detect. The value of ? is the 16 bit number stored at ERSAV where ERSAV is a symbol from ATETBL.

& denotes the assembler program counter. The value of & is the 16 bit number stored at ASPC, a symbol from ATETBL.

$ denotes the assembler storage counter. The value of $ is the 16 bit number stored at STCTR, a symbol from ATETBL.

exp! where exp is a numerical expression, denotes the exp´th line in the current file. exp is interpreted as a positive 16 bit number, and if this is larger than the number of lines in the file exp! denotes the Ø at the end of the file. ! without exp and Ø! are the same as 1!.

Interval Arguments:

An Interval Argument may be specified by a Matching Interval Symbol, a Non-Matching Interval Symbol, or an expression in which interval symbols are combined by:

Interval Operations:

Concatenation: Denoted by one interval symbol immediately followed by another. Neither interval symbols may contain any non-matching interval symbols. The interval denoted is the interval matching the concatenation of the patterns specified by the two interval symbols.

Arithmetic Operations: The symbol <I1><op><I2>, where <I1> and <I2> are interval symbols and <op> is an arithmetic operation has lower address = <lower address of I1><op><lower address of I2>, upper address = <upper address of I1><op><lower address of I2>.

% The symbol <I>% where <I> is an interval symbol expands the upper end of the interval to an entire line

Occurrencing: The symbol <exp><I> where <exp> is an arithmetic expression and <I> is an interval symbol containing no non-matching symbols denotes the exp'th occurrence of the pattern given by <I>. <exp> is interpreted as a signed 15 bit number. If the value of <exp> is negative, <exp><I> denotes the ABS(exp)'th occurrence of the pattern given by <I> counting backwards from the end of the initial reference interval.

. Any number of consecutive dots can be used. <I1>.<I2> denotes the interval from the beginning of I1 to the end of I2. If <I2> contains only matching symbols, they match the first occurrence of the pattern given which occurs AFTER I1. If the end of I2 is lower than I1 an error and a matching failure will occur, even if <I1> and <I2> contain no matching symbols.

| <I1>|<I2> causes the initial reference interval for all matching symbols in <I2> to be the interval denoted by <I1>, and as soon as | is encountered > is given the value of the upper address of I1.

Order of Priority of Interval Operations:

| | | |
|---|---|---|
| 1 | Concatenation | Highest |
| 2 | Arithmetic, %, occurrencing | |
| 3 | . | |

4 | Lowest

Default Interval Argument:

For any command except disk commands which take an Interval Argument where no argument is given, the argument used will be the last interval computed.

Disk File Reference Symbols:

Disk Drive Designator:
a letter A through H. A through D denote Disk Jockey Drives 1 through 4, E through G denote North Star Drives 1 through 3. H is currently unused.

Disk File Names:
Any sequence of up to 8 ASCII characters whose value is greater than hex 20 (a blank) except for:

1. ,
2. :
3. @ as the first character of the file name
4. " as the first character of the file name

optionally followed immediately (no intervening blank) by : then a Disk Drive Designator. If no Disk Drive Designator is present it is assumed the file resides on the Current Drive.

In place of a file name, the symbol " denotes the most recently referenced file. If a file name is prefixed immediately (no intervening blank) by the character @, a disk command referencing this file will first execute a PAUSE command and then execute the disk command.

<R> denotes the interval in the memory occupied by the most recently read record from the disk. The lower address of <R> is given by the 16 bit value stored at RECAD and the upper address by the 16 bit value stored at RECND, where RECAD and RECND are symbols from ATETBL.

Memory Files:
Files in the memory are located within an area called the source area, denoted by <S>. The lower address of <S> is given by the 16 bit value stored at BOSAP and the upper

address by the 16 bit value stored at EOSAP, where BOSAP and EOSAP are symbols from ATETBL. S^ deontes the same value as 1 + the contents of EOSAP. Each file is a sequence of bytes bounded by Ø´s. One such file is the current file, denoted by <F>. The lower address of <F> is given by the 16 bit value stored at BOFP and the upper address by the 16 bit value stored at EOFP, where BOFP and EOFP are symbols from ATETBL. <F> does not include its bounding Ø´s. <S> includes the Ø giving the lower bound of the lowest file and the Ø giving the upper bound of the highest file.

Symbol Table:

The symbol table is denoted by <T>. The lower address of <T> is given by the 16 bit value stored at SYMTB, and the upper address by the 16 bit value stored at TABA, where SYMTB and TABA are symbols from ATETBL. Each entry in the symbol table consists of the characters of the symbol with the last character having its high order bit set, followed by the 16 bit value of the symbol. A Ø occurs at the end of the table. The symbol table includes both variables and assembler labels. When a value is assigned to a symbol, the symbol table is searched until the terminal Ø regardless of the value of TABA, and if the symbol is undefined it is added to the end of the table and TABA updated. T^ denotes the same value as 1 + the contents of TABA. On power-up and after the Z command the symbol table contains values for the following symbols used by the assembler:

A, B, C, D, E, H, L. M, SP, PSW

Command Summary:

" <Interval Argument> ("Quote interval")

Prints on the terminal the characters contained in the interval given as argument, with a line feed issued automatically at each carriage return.

´ (no argument) ("Quote one line")

Prints on the terminal the line containing the target character, with the character ^ immediately preceding the target character.

E <Entry Argument> where ("Enter")

an Entry Argument is a sequence of Matching Symbols of the form [text] or #sequence of numbers# or _ as defined above. In this case text may extend across any number of lines.

If the entry pointer is within the source area, the characters denoted by the argument are inserted into the current file between the target character and the preceding character, expanding both the current file and the source area; otherwise they overwrite those characters in the memory beginning with the target character. In either case the target character after the command is the character after the material entered by the E command.

K <Interval Argument> ("Kill")

Kills the interval given as argument. If the interval is within the source area, the interval is deleted from the file in which it occurs and moved to the area of memory immediately following the source area by an internal M command. The deleted text is not surrounded by Ø's. Immediately after a K command the Default Interval Argument will be the interval occupied by the deleted text outside the source area. The target character will be the first character in the source file after the material that was deleted.

If the interval is not within the source area, the interval is zeroed and the target character becomes the first character in the interval.

M <Interval Argument> ("Move")

Moves the interval given as argument to the address given by the entry pointer. If the entry pointer is within the source area, then the interval is inserted into the file containing the entry pointer between the target character and the preceding character. Otherwise the characters of the interval are copied to overwrite the area of memory beginning with the target character. The target character becomes the character after the new location of the last character moved. Internal pointers to the following are updated by the M command:

<S>, <F>, <R>, <T>, S^, T^, <, >, (and hence the default interval argument), User Command Table, Macro Command Interpretation Pointer, Return Adresses, Repeat Addresses. A pointer is updated if the address it gives is within the interval given by the argument before the move takes

place. There is a pointer for both the lower and upper address for each of <S>, <F>, <R>, and <T>, and if only one such pointer lies within the argument because the argument overlaps, only the one pointer will be updated. This may cause an erroneous value.

^ <Interval Argument> ("set Entry Pointer")

The entry pointer is set to the lower address of the argument.

C <Interval Argument> ("Copy")

The characters of the interval are copied to the area of memory beginning with the target character. No pointers are updated and the action of the command is not affectd by whether the entry pointer is within the source area.

B <Arithemetic Expression> ("Base")

The low order byte of the value of the expression becomes the current base. It may be any number > 4. For bases > 1Ø, ASCII characters in order beginning with A are used for digits. Numbers are printed in split form for all bases other than 16.

# <Interval Argument> ("Quote numbers (core dump)")

The numerical value in the current base for the bytes of the interval are printed on the terminal, with the address of the first byte of a line printed at the left of the line.

? <Interval Argument> ("Where?")

Prints on the terminal in the current base the upper and lower address of the interval given as argument.

? " Prints on the terminal the file name including disk drive designator for the most recently referenced file.

N (no argument) ("New file")

Creates a new empty source file at the end of the source area and makes it current. The target character becomes

the Ø at the end of this file.

REF [<Interval Argument string up to 24 characters>]

("Reference interval")

The string in brackets is saved in a special location. For all subsequent commands which take <F> as the initial reference interval the Interval Argument given by this string is evaluated and becomes the initial reference interval. Commands taking <S> as the initial reference interval are unaffected.

REF []

Discontinue use of the REF feature.

REF (no argument)

Print on the terminal the last string saved as the argument for a REF command.

DEF [<command string up to 24 characters>] ("Default command")

The string in brackets is saved in a special location. Subsequently whenever a command is expected and a carriage return only is typed with an empty command line, the string saved will be executed as a default command.

DEF []

Discontinue use of the DEF feature

DEF (no argument)

Print on the terminal the last string saved as the argument for a DEF command.

F <Interval Argument> ("File")

Establishes as the current file the file containing the lower address of the interval given as argument. If this address contains Ø this Ø becomes the lower bounding Ø of the file, otherwise the lower bounding Ø is the first Ø backward in the memory from this address. The upper bounding Ø is the first Ø in the memory forward from the lower bounding Ø. The entry pointer is set to the upper bounding Ø. If either bounding Ø is within the source area, the source are is expanded if necessary to include

the new file. If the new file is entirely outside the source area the source area consists only of the new file and its bounding Ø´s. The initial reference interval is <S> regardless of any REF commands.

O <Interval Argument> ("Originate source area")

The interval given as argument is established as the source area. If the lower address of the interval and the upper address are the same, an empty source area consisting only of two consecutive zeros is created at the address. Otherwise the Ø forming the lower bound of the source area is written at the lower address of the interval and the upper bound Ø is written at the upper address. The highest (possibly empty) file in the new source area is made current and the target character becomes the Ø at the end of this file and hence also at the end of the source area.

O (no argument)

An empty source area is created at the beginning of the current source area.

D <Interval Argument> ("Do")

The string of commands located at the lower address of the interval given as argument is executed. Execution of this string is terminated by either an end of the file (a Ø) or a QS or QF command. When execution terminates control is passed back to whatever command string invoked the D command. Analogous to subroutine call. The initial reference interval is the source area regardless of any REF commands.

G <Interval Argument> ("Goto")

Execution of commands is transferred to the lower address of the interval given as argument. There is no return to the command string invoking the G command. Analagous to a GOTO command. The initial reference interval is the source area regardless of any REF commands.

R <Interval Argument> ("Repeat")

The rest of the command line or file in which the command occurs is repeated the number of times given by the lower address of the argument, which is usually an arithmetic

expression. The value of the lower address of the argument is treated as a 16 bit non-negative integer. If the argument is Ø the remainder of the command string containing the R command is not executed. Execution of an R loop may be terminated by a QS or QF command. R loops may be nested. If the command string containing the R command is to issue a return on completion, the return is issued after the repetition has been terminated.

* <string> where ("comment")

the string contains no commas, blanks, carriage returns, or end of file characters (Ø's).

All characters from the * up to the next comma, blank, carriage return or end of file are ignored allowing the string to serve as a label.

QF <Interval Argument> ("Quit on Failure")

Execution of a command string invoked by a D command, or repetition invoked by an R command, is terminated and a return if pending is executed provided the argument results in an Argument Failure, where an Argument Failure is caused by either the failure to obtain a match for a matching argument, or by an interval which results in the lower address being greater than the upper address. An argument containing a syntax error or undefined variable will cause an actual error rather than an Argument Failure. If an Argument Failure does not occur execution continues. Only the innermost process is terminated. An Argument Failure does not result in an error condition.

QS <Interval Argument> ("Quit on Success")

Execution of a command string invoked by a D command, or repetition invoked by an R command, is terminated and a return if pending is executed unless the argument results in an Argument Failure, where an Argument Failure is defined as above. If an argument failure occurs execution continues. Only the innermost process is terminated. An Argument Failure does not result in an error condition.

X <Interval Argument> ("Execute")

The machine language subroutine at the lower address of the interval given as argument is called. If it maintains

the integrity of the stack it may end with a RET statement and execution of the process which invoked the X command will continue.

<variable>= <Interval Argument>

The lower address of the argument is assigned to be the value of the variable. If the variable is not present in the symbol table it is entered there as described above under Symbol Table. There must not be a blank between the variable name and the = sign.

COM [<prompt string>] ("Accept Command")

The string within brackets given as argument is printed on the terminal as a prompt. One command will be accepted and executed and then the process invoking the COM command will continue. The command may be a multi-line E command, or a multiple command command line. The command line typed in response to the COM command may be ESC'ed and retyped. An error in the command will cause the process invoking the COM command to terminate and issue an error condition.

ATE [<prompt string>] ("invoke DISKATE recursively")

The string within brackets given as argument is printed on the terminal as a prompt. DISKATE calls itself recursively, and then will accept and execute any number of commands. An error condition terminates only those processes invoked by commands given after the ATE command is executed. Execution of DISKATE continues in this state until the BYE command is given, at which time execution of the process that invoked the ATE command continues.

BYE (no argument) ("bye-bye: return to to DISKATE caller")

DISKATE returns to its caller. If it was called internally by an ATE command, the process invoking the ATE command resumes. If it was called by a machine language program, execution of that program resumes. If DISKATE was invoked only by the power-up procedure, the BYE command will cause DISKATE to be reloaded from the disk.

RENT (no argument) ("Reenter")

DISKATE is reentered at the "ground level" at the same

entry point given by RENT where RENT is a symbol in ATETBL. The stack is reinitialized but all other variables and pointers are left intact. All processes active at the time the RENT command is given are terminated, including processes invoking ATE commands. If a machine language program has called DISKATE, the RENT command will cause the return address to be lost.

Y (no argument) ("Wipe")

Calls the terminal initialization routine. The vector to this routine is located in the vector table which begins the module IO, at IO+9. On power-up this routine is called by an internal Y command.

WID <arithmetic expression> ("terminal Width")

The width for the terminal is set to the low order byte of the value of the argument. If the terminal produces a new line automatically on line overflow, a width equal to the hardware terminal width will produce an extra blank line for lines whose length is exactly the terminal width.

WID (no argument)

Prints the current terminal width on the terminal.

ECHO [<one-line character string>]

The character string given as argument is printed on the terminal. There is no matching.

PAUSE (no argument)

The message PAUSE is printed on the terminal and the panic state is entered.

IO <arithmetic expression> ("set I/O device number")

The low order byte of the value of the arithmetic expression is subsequently supplied in the A register as an I/O device number to all calls to the terminal I/O routines. A carriage return and line feed are printed on the device.

L <filename> <Interval Argument> ("Load disk file")

The file is loaded at the lower address of the interval given as the second argument. The Default Interval Argument may not be used as the Interval Argument.

L <filename> (no Interval Argument)

If the file is not a source file it is loaded at the address given by the memory address portion of the directory. If the file is a source file it is loaded at the address given by the entry pointer. If the entry pointer is within the source area, the file is inserted into the memory file containing the target character between the target character and the preceding character, with any bounding Ø´s removed. In this case the target character becomes the character in the source area after the last character of the disk file.

S <filename> <Interval Argument> ("Save disk file")

The interval given as the second argument is saved as a disk file whose name is the first argument. If a file with this name already exists, it is replaced by the contents of the interval. If no file exists with this name it is created. If enough room for the file exists on the disk but not in one contiguous block the disk will be compacted and the message COMPACTING ON DRIVE (drive designator) will be printed on the terminal. If the interval is within the source area the file type is set at Ø (source file), otherwise the file type is set at 1. The memory address portion of the directory will contain the lower address of the interval given as the second argument. If the disk or directory is full, an error message to that effect will result and the contents of the disk will be unchanged.

GO <filename>

The disk file whose name is the argument is loaded and executed, via an internal D command if it is a source file or via an internal X command otherwise. Equivalent to L <filename>,D<R> for source files or L <filename>,X<R> for non-source files. The same conventions for loading the file that apply to the L command apply to the GO command.

I <optional Disk Drive Designator> ("Identify")

The directory on the disk drive given as argument is listed. If no argument is given the directory on the current drive is listed. A copy of the directory is left at the

1024 bytes beginning at DISKBUF, where DISKBUF is a symbol in ATETBL.

FS <optional Disk Drive Designator> ("Free Space")

The amount of free space on the disk and on the directory for the drive given as argument is listed. If no argument is given the listing is for the current drive.

U <filename> ("Unsave")

The disk file whose name is the argument is deleted from the directory.

T <source filename> <destination filename> ("Transfer")

The contents and directory attributes of the disk file whose name is the first argument are transferred to the file whose name is the second argument. If there is no file whose name is the second argument it is created, with compacting occurring as with the S command.

TD <source Disk Drive Designator> <destination Disk Drive Desig.>

("Transfer Disk")

Each file on the disk designated by the first argument is transferred to the disk designated by the second argument with the same conventions as for the T command. Files on the destination disk drive whose names are not in the directory on the source disk drive are unaffected. If the destination disk drive or directory does not have enough room to complete the transfer for every file on the source disk drive, the transfer will be carried out for each file for which there is room and then an error message will occur indicating that the disk or directory is full.

RN <current filename> <new filename> ("Rename")

The file whose name is given by the first argument is renamed to the name given as the second argument. If the second argument includes a Disk Drive Designator it is ignored.

W <filename> <Interval Argument> ("Write memory address")

The lower address of the interval given as the second argument is written in the memory address portion of the directory for the file name given as the first argument.

CD <Disk Drive Designator> ("Current Drive")

The disk drive designated by the argument is established as the current drive.

A <list of file names separated by blanks> ("Assemble")

Invokes both passes of the assembler to assemble the source program logically equivalent to the concatenation of the files given in the argument list. @ may be included as a file name to cause a PAUSE command to be executed while diskettes are changed. The name of each file in the list is printed on the terminal as it is assembled by each pass. The source area is not affected.

A (no argument)

Invokes both passes of the assembler to assemble the current file. (I.e. <F>.)

A1 <list of file names> ("Assembler Pass 1")
A1 (no argument)

Identical to the A command but invokes only Pass 1 of the assembler. The symbol table is compiled but no object code is generated.

A2 <list of file names>
A2 (no argument)

Identical to the A command but invokes only Pass 2 of the assembler.

Q (no argument) ("Quiet")

Assembler listings are suppressed execpt for assembler error messages, until the next J command is encountered.

J (no argument) ("Jabber")

The assembler will resume printing listings.

& <Interval Argument> ("set assembler program counter")

The assembler program counter is assigned the value of the lower address of the interval given as argument.

$ <Interval Argument> ("set assembler storage counter")

The assembler storage counter is assigned the value of the lower address of the interval given as argument.

Z (no argument) ("Zero the Symbol Table")

All symbols are removed from the symbol table except the initial symbols used by the assembler (see above under Symbol Table.)

Z <symbol>

The symbol, either a variable or assembler label, is removed from the symbol table and the symbol table is compacted.

Z> <symbol>

All symbols defined after the symbol given as the argument are removed from the symbol table. The symbol given as the argument is left intact.

INTE (no argument) ("use Intel Format")

All subsequent P commands and Assembler commands will use the Intel format until a PROS is encountered.

PROS (no argument) ("use Processor Technology Format")

All subsequent P commands and Assembler commands will use the Processor Technology Format until an INTE is encountered.

P <Interval Argument> ("Print in Assembler Format")

The interval given as argument is printed on the terminal with line numbers in assembler format, with label fields, opcode fields, operand fields, and comment fields in columns as determined by the TAB settings. The choice of

Intel or Processor Technology format is unaffected by any INTE or PROS psuedo-ops within the source code printed, but is affected by INTE or PROS commands as well as INTE or PROS pseudo-ops executed by the assembler during Pass 2.

TAB (no argument) ("list TAB settings")

The current TAB settings are printed on the terminal in the following order:

1. relative column for the label field
2. " " " " opcode field
3. " " " " operand field
4. " " " " comment field
5. source code offset

The columns for items 1-4 above are relative to Ø for P command listings, and relative to item 5 for assembler listings.

TAB <list of up to 5 arithmetic expressions separated by blanks>

The TAB setting items listed above are replaced by the value of the corresponding expressions in the argument list. If there are less than 5 expressions in the list, items for which no expression is given are unchanged. The TAB is set to the low order byte of the value of the expression.

V <Interval Argument> ("Evaluate")

The lower address of the interval given as argument is left in the HL register pair and the upper address in the the DE register pair. Nothing is printed on the terminal except for error messages. For use only in machine language programs calling the entry point ATECOMS, where ATECOMS is a symbol in ATETBL. A jump to ATECOMS is written at memory location Ø.

Assembler Formats:

Processor Technology Format:

The label field begins in the first character of each line. If a statement is not to be given a label, the first character of the

line must be blank. The label is given without any symbol to terminate it. A label must contain only upper-case letters or digits and must begin with a letter. A statement consisting of only a label field assigns the value of the program counter to the label. The opcode field is separated from the label field by one or more blanks. If the statement takes an operand, the operand field is separated from the opcode field by one or more blanks. The comment field is separated from the previous field by one or more blanks. The entire line can be designated as a comment by giving * as the first character on the line.

Intel Format:

Same as above, with the following differences: a label is terminated with a colon. If a statement is not to be given a label, the opcode may begin the line. The comment field begins with a semicolon (";").

Assembler Pseudo-ops:

AORG ("Address Origin")

The assembler program counter, &, is set to the value of the operand.

SORG ("Storage Origin")

The assembler storage counter, $, is set to the value of the operand

ORG ("Origin")

The assembler program counter is set to the value of the operand. The assembler storage counter is set to its current value + the new value of the program counter - the previous value of the program counter.

DB ("Define Byte")

The operand is defined as a one byte constant. Multiple operands separated by commas may be given, in which case a one-byte constant is defined for each operand.

DW ("Define Word")

Same as above except that constants defined are 16 bit

words.

DS ("Define Storage")

A block of storage is defined whose length is the value of the operand. The storage is not initialized.

ASC ("Define ASCII string")

Generates a string constant consisting of the ASCII characters comprising the operand field.

ASCx where

x is any printing non-alphabetic or digit character

Same as ASC except that in the string generated each occurrence of x is replaced by a blank.

EQU ("Equate")

The label of the EQU statement is assigned the value of the operand. A label must be present, and the same label cannot be given a value by more than one EQU statement.

END Indicates the end of the source code. In the absence of an END pseudo the end of file serves the same purpose.

INTE Sets the current format as the Intel Format.

PROS Sets the current format as the Processor Technology Format.

IF ("Conditional assembly")

The operand must be in the form:

<expression>,<label>

If the value of the expression is Ø, assembly skips ahead to the label. Otherwise assembly continues with the next statement.

Assembler Error Codes:

A Argument Error. The operand field is invalid for the given opcode. The A code usually occurs only in Pass 2, though an invalid EQU operand will cause it to occur in Pass 1.

M Missing Label. An EQU statement occurs without a label. Printed during both passes.

D Doubly Defined Label. A label is given where the label is identical to a symbol already defined.

L Label Error. The first character in a label field is not an upper-case letter of the alphabet. When this occurs, 3 NOPs will be generated in place of the statement.

O Opcode Error. The opcode field is not a proper opcode or pseudo-op. 3 NOPs will be generated in place of the statement.

Symbols in ATETBL:

ATETBL is an abbreviation of the symbol table from an assembly of ATE, the non-I/O portion of DISKATE. It correlates with the version of DISKATE supplied, but in different versions of DISKATE the symbols in ATETBL may not have the same value. Any reference to such a symbol should take its value from ATETBL itself and not a listing of any previous version of ATETBL.

BEGIN The beginning of ATE. This location is an entry point which performs initialization. The stack and some internal variables are initialized. The command O,Z,Y is executed, a message is printed on the terminal, and the command GO STARTUP is executed.

RENT The entry point to ATE that avoids initialization. Only the stack is initialized. If DISKATE has been called by another machine language program, jumping to this location will cause the return address to be lost.

ATECOMS An entry point which executes an arbitrary string of DISKATE commands and then returns. HL must contain the address of the beginning of the command string. A jump to ATECOMS is written at memory location Ø when DISKATE is loaded from the disk. ATECOMS may be invoked by loading the address of a command string in HL and then

executing the instruction RST 0.

WHAT — The entry point to the DISKATE error exit. A machine language program which detects an error can return to DISKATE by jumping to this entry point.

READER — A subroutine which reads one line from the terminal. DE must point to a buffer which begins with a string terminated by a 0 byte. This string will be printed on the terminal as a prompt. The characters are stored in the buffer beginning immediately after the 0 at the end of the prompt. The byte in the B register must be 2 greater than the maximum number of characters which can be stored in the buffer. (This number should not include the length of the prompt string.)

VCHK — A subroutine which determines if an argument follows the invokation of a machine language program. Returns with the Z flag OFF if an argument is present.

CVALS — A subroutine which evaluates an interval argument after the invokation of a machine language program. The lower address is placed in HL and the upper address in DE. In case of error it exits via WHAT rather than returning to the machine language program that called it.

VALUS — A subroutine similar to CVALS which evaluates an interval argument after the invokation of a machine language program. This routine will return to the calling program even in case of error. Returning with the Z flag OFF indicates an error. When called the lower and upper addreses respectively of the initial reference interval must be supplied in HL and DE. The initial reference interval need not be given if it is known the argument will contain no matching symbols.

OUT — A subroutine which prints the character in A at the terminal. The I/O device number is whatever has been established by an IO command, and is 0 at power-up. Before printing the character the panic detect routine is called. All registers and flags are preserved, a line feed is supplied automatically after a carriage return. The internal print head counter is updated. When the terminal width is reached a carriage return and line feed are printed.

INECO — A subroutine which obtains a character from the terminal and echoes it via the OUT routine above. All flags and registers are preserved except A, which contains the input character. ESC and backspace are treated like any other character.

PHLSB A subroutine which prints the value in HL in the current base, including leading Ø's. The value is printed in split form with 3 digits per byte for all bases other than 16. If the base is 16 4 digits are printed.

PHLDC A subroutine which prints the value in HL in base 1Ø suppressing leading Ø's.

USRCT An address giving the beginning of space provided inside ATE for a user command table.

ROMEND An address giving the upper address of space provided inside ATE for a user command table.

UCTAD The address of a 16 bit value giving the address of the beginning of the user command table. If this is not changed by personalization, the contents of UCTAD..UCTAD+1 on power up will be the same address as USRCT.

ASPC The address of the assembler program counter.

STCTR The address of the assembler storage counter.

BOSAP The address of a 16 bit value giving the lower address of the source area.

SYMTB The address of a 16 bit value giving the beginning address of the symbol table.

END The address in memory of the last byte of code in the module ATE.

EOSAP The address of a 16 bit value giving the upper address of the source area.

BOFP The address of a 16 bit value giving the lower address of the current file.

EOFP The address of a 16 bit value giving the upper address of the current file.

TABA The address of a I6 bit value giving the Ø ending the symbol table. (Not consulted when values are defined for a new symbol.)

CHPTR The address of the entry pointer.

P1 The address of a 16 bit value giving the lower address the last interval computed and the value of <.

P2 The address of a 16 bit value giving the upper address

the last interval computed and the value of >.

RECAD The address of a 16 bit value giving the lower address the area of memory occupied by the most recently read record from disk.

RECND The address of a 16 bit value giving the upper address the area of memory occupied by the most recently read record from disk.

ERSAV The address of a 16 bit value giving the address of the most recent program execution error, or the memory address at which a printout was interrupted -- i.e. the value of ?.

PHD The address of 1 byte value giving the column in which the print head is waiting.

MAXBS A 16 bit constant giving the maximum number of blocks allowed on a disk drive.

DISKBUF The beginning address of the ATE disk buffer. After an I command a copy of the entire directory will begin at this location. The first 1Ø24 bytes of this buffer are used only during disk commands. The rest is shared by other buffers.

RAMEND The end of RAM used internally by ATE. Memory above this address is free to the user.

# Index

! following a page number indicates the page number is part of the System Reference Summary. Non alphabetic characters are given in order of their ASCII codes.