

# **DISK BASIC VER. 3.5**

## **USER'S GUIDE**

### **IMPORTANT NOTE**

Although every effort has been made to make the supplied software and its documentation as accurate and functional as possible, Southwest Technical Products Corporation and Technical Systems Consultants will not assume responsibility for any damages incurred or generated by such material. Also, Southwest Technical Products Corporation and Technical Systems Consultants reserve the right to make changes in such material at any time.

Copyright © 1978 Southwest Technical Products Corporation

All Rights Reserved



SOUTHWEST TECHNICAL PRODUCTS CORPORATION  
219 W. RHAPSODY SAN ANTONIO, TEXAS 78216

SECRET

THE BARRAGE

SECRET

SECRET

SECRET

SECRET

## SWTPC Disk BASIC

SWTPC Disk BASIC is a complete BASIC interpreter for use in both home and business applications. Features of SWTPC Disk BASIC include nine significant digit binary coded decimal addition, subtraction, multiplication and division, seven digit trigonometric functions and numerous string operations. Disk data files are also supported for manipulating or storing data.

This manual is designed to acquaint the user with the various features of SWTPC Disk BASIC—It is not designed to be a complete course on the BASIC language. This manual also assumes that the user is familiar with the supplied disk operating system (DOS) and its respective user's guide.

### Definitions

Before actually describing each BASIC function, several terms need to be defined and manual notation described.

A **command** is a BASIC operation that generally has an immediate effect on the operation of BASIC.

A **statement** is a word or group of words that directs the execution of a BASIC program.

A **function** is a BASIC operation that usually results in a numerical operation or string processing.

A **variable** is a letter, or a letter and a number, that is used to represent a numeric or string value. Variables may be named by any single alphabetic character (A-Z) or any single alphabetic character followed by a number (0-9). Variables of this type represents a numerical value.

Example: A can be equated to 3.44  
B1 can be equated to  $-7.2315 + \text{SIN}(3)$

A **string variable** is a single letter followed by a \$ that is used to represent literal (alphanumeric or text) data.

Example: A\$ can be equated to "1234" but not to 1234. (the quotation marks make 1234 a string).  
Note: a string may **not** be represented by a letter and a number such as A3\$.

When BASIC initializes, the string variable length is set equal to a maximum of 32 characters.

This manual uses the following notation conventions:

/line N/	denotes a BASIC line number such as 0090
/var/	denotes a variable name such as A3
/exp/	denotes a mathematical expression such as $3+5-2$
/rel.exp./	denotes a relational expression such as $A=5$
/string/	denotes a collection of literal alphanumeric characters enclosed by quotation marks such as "TEST1"
X	denotes a variable or expression that has a numerical result
X\$	denotes a string variable

### Restrictions on Program Lines

The following restrictions are placed on all BASIC program lines:

- 1.) Every line must have a line number ranging between 1 and 9999. Do not use line # 0.
- 2.) Line numbers are used by BASIC to order program statements sequentially.
- 3.) In any program, a line number can be used only once.
- 4.) A previously entered line may be changed by entering the same line number along with the desired statement(s). Typing a line number followed immediately by a carriage return deletes that line and line number.
- 5.) Lines need not be entered in numerical order since BASIC will automatically order them in ascending order.
- 6.) A line may contain no more than 72 characters including blanks.

- 7.) Blanks, unless within a character string and enclosed by quotation marks, are not processed by BASIC and their use is optional. Numbers can contain no imbedded blanks.

Example:

```
110 LET A=B + (3.5*5E2)
```

is equivalent to

```
110 LET A = B+(3.5*5E2)
```

- 8.) Multiple statement lines are accepted with a colon (:) used as the separator. BASIC will process the line from left to right.

Example:

```
10 A=3 : B=5 : C=A*B
```

### Data Format

The range of numbers that can be represented in this version of BASIC is 1.0E-99 to 9.999999-99E99. E99 represents  $10^{99}$  while E-99 represents  $10^{-99}$ . The E stands for exponent.

There are nine digits of significance in this version of BASIC. Numbers are internally truncated (last digits dropped) to fit this precision.

Numbers may be displayed and entered in three formats: integer, decimal and exponential.

Example:      153          34.52          136 E-2

Transcendental functions (SIN, COS, TAN, ATAN, SQR, LOG and  $\uparrow$ ) are all evaluated by a limited infinite series. For these functions accuracy is limited to seven significant digits.

### Mathematical Operators

The mathematical operators used in BASIC are as follows:

$\uparrow$	Exponentiation (raises to a power)
- (unary)	Negate (used for denoting negative numbers)
+	Addition and string concatenation
-	Subtraction
*	Multiplication
/	Division

No two mathematical operators may appear in sequence, and no operator is ever assumed. (A++B and (A+2)(B-3) are not valid). Exception: 5+3 is allowed.

Examples:

A=B $\uparrow$ C	A is evaluated to B raised to the C power
A=B+5	A is evaluated to B plus a negative 5
A=3/2	A is evaluated to 3 divided by 2

### Priority of Operations

BASIC recognizes the priority of operation in the following order:

1. Exponentiation ( $\uparrow$ )
2. Negation (-)
3. Multiplication (\*) and division (/)
4. Addition (+) and subtraction (-)

A BASIC expression is evaluated from left to right in the above priority sequence unless parenthesis are encountered. The operators within the parenthesis are evaluated first utilizing the above priority structure.

Examples:

LET A=2	
LET B=3	
LET C=4	
B $\uparrow$ 2 + C/A $\uparrow$ 2	gives a result of 10
C $\uparrow$ 2 - C/A	gives a result of 14
A * (A+B*2)-22	gives a result of 0
A $\uparrow$ A $\uparrow$ B	gives a result of 64

## String Concatenation

Although any one string variable may be a maximum of 32 characters (or whatever the length is set equal to using the STRING= command), strings may be joined up to a maximum of 126 characters for printing. The concatenation symbol is +.

```
Example: A$= (32 char. string)
        B$= (32 char. string)
        PRINT A$+B$ (prints a 64 character string)
also: A$= "HELLO"
      B$= "JOHN"
      C$= A$ + B$ (C$ still limited to 32 char.)
```

## Arrays

Sometimes it is convenient for a variable to represent several values at one time. A variable such as this type can be considered as an array and each element can be accessed independently. In referencing an array variable, the element number in the array must be specified along with the variable name. For example, say we wanted the variable A to represent 4 values. The following program would assign a different value to each element of A.

```
10 DIM A(4)      Dimension A to hold four elements
20 A(1)=1 : A(2)=2 : A(3)=3 : A(4)=4
```

As seen above, a particular element is referenced by a subscript N, such as A(N), where 1 is the first element in the array.

Two dimensional arrays are also accepted by BASIC. Two dimensional arrays are useful when working with data which is easily represented as a matrix.

```
Example: 10 DIM A (3,3)
        20 A(1,1)=1 : A(1,2)=2 : A(1,3)=3
        30 A(2,1)=4 : A(2,2)=5 : A(2,3)=6
        40 A(3,1)=7 : A(3,2)=8 : A(3,3)=9
```

gives the following matrix:

1	2	3
4	5	6
7	8	9

String variables may also be dimensioned as arrays. (A\$(5,2))

If no DIM statement is used to specifically dimension an array, a dimension of either 10 or 10 by 10 is assumed.

## Program Preparation and System Operation

At the time that BASIC is executed, BASIC will automatically determine the range of working storage. If you wish to limit the amount of memory BASIC uses, refer to Appendix D of this manual. This is normally not necessary unless external machine language subroutines are being used.

The system is then ready to accept commands or lines of statements. For example the user might enter the following program:

```
150 REM DEMONSTRATION
160 PRINT "ENTER A NUMBER";
170 INPUT A
180 LET P = A*A*3.1415926
185 PRINT
190 PRINT "THE AREA OF A CIRCLE WITH";
200 PRINT "RADIUS"; A; "IS"; P
210 STOP
```

If the user wishes to insert a statement between two others, he need only type a statement number that falls between the other two. For example:

```
183 REM THIS IS INSERTED BETWEEN 180 and 185.
```

If it is desired to replace a statement, a new statement is typed that has the same number as the one to be replaced. For example:

180 P=(A\*A)\*3.1415926 replaces the previous LET statement.

Each line entered is terminated by a Carriage Return and is not processed by BASIC until this key is depressed. BASIC then positions the print unit to the correct position on the next line.

If a mistake is made during type in before typing the Carriage Return, a BACKSPACE may be used to delete erroneous characters. The backspace character for BASIC is a hexadecimal ASCII 08 (Control H). BASIC assumes that the terminal automatically generates a "cursor left" when a control H is entered.

Example:

```
30 REM THIS IS A TESZ (CTL.H)T
```

The CTL.H moves the cursor back over the Z so that the result is  
TEST

If it is desired to remove a complete line that was typed in before typing the Carriage Return, the CANCEL key (hex ASCII 18, control X) may be depressed. This will delete all information that was typed in on the current command or statement line. BASIC will respond with DELETED.

Example:

```
10 FOR 1 to 10 (CTL.X)
```

```
DELETED
```

```
PATCH (CTL. X)
```

```
DELETED
```

If the user wishes to execute a program at this point, the RUN command, as described in the command section, should be entered.

### Program Abort

If, at any time, it is desired to abort a looping or otherwise malfunctioning program, BASIC has a provision for exiting the program and returning to the command (READY) mode. The abort (break) character for BASIC is a control C, hex ASCII 03. Entering one control C will immediately halt the execution of the current BASIC program and will return BASIC to the command mode. During a printout sequence, such as listing a program, typing one ESC (escape) character will cause the current printout to halt. Typing another ESC will cause printout to resume while typing a RETURN will force BASIC back to the command mode.

NOTE:When in the middle of a machine code USER routine, control C will have no effect. If necessary, the computer's RESET button can be depressed. Resetting the computer's program counter to 0103 before re-entering BASIC will keep the current BASIC program intact.

### Commands

It is possible to communicate in BASIC by typing direct commands at the terminal device. Also, certain other statements can be directly executed when they are entered without statement numbers.

Commands have the effect of causing BASIC to take immediate action. A typical BASIC language program, by contrast, is first entered statement by statement into the memory and then executed only when the RUN command is given.

When BASIC is ready to receive commands, the word READY is displayed on the terminal device. After each entry, the system will prompt with a "#".

Commands are typed without statement numbers. After a command has been executed, READY will again be displayed indicating that BASIC is ready for more input—either another command or program statements.

## APPEND

The APPEND command causes a program on tape to be loaded into memory. The APPEND command operates the same as the LOAD command except that the current BASIC program is not cleared from memory.

## CONT

A CONT (continue) command can be entered after a program has halted from a STOP command or after a program has been aborted with a control C. Between the time that the program has stopped and the time that CONT is entered no changes should be made in the program. The program will then continue with the next statement after the STOP command or wherever the program was when control C was pressed.

## DIGITS=X

The DIGITS= command sets the number of digits that will be printed to the **right** of the decimal point when displaying numeric variables. This will truncate (not round) any digits greater than the number printed, and will force "0"s if there aren't enough significant digits to fill the number of positions specified in the "DIGITS =" command.

A DIGITS=0 command resets BASIC to the floating point mode.

The DIGITS= command may also be used as a program statement.

## DOS

The DOS command causes computer control to be returned to the DOS operating system.

## LINE=X

The LINE= command is used to specify the number of print positions in a line (line length) where X is the desired number of print positions.

Example: LINE=65, LINE=80, LINE=40

Note: Each line is broken up into 16 character "zones". If the print position is within the last 25 percent of the "line" length and a "space" is printed, a C/R L/F will be output. This is so that a number or word will **not** be split up at the end of a print line. If it is desired to inhibit this feature (for precise print control) just set the line length equal to greater than 125% of the desired total print line length. This can be very important when using the TAB command.

The LINE= command can also be used as a program statement.

## LIST

LIST (line #)

LIST (line #m)-(line #n)

The LIST command causes the desired lines of the current program to be displayed on the control terminal. The lines are listed in increasing numerical order by line number. A LIST command causes all lines of the current program to be displayed, a LIST (line #) command lists only the line specified and a LIST (line #m)-(line #n) command causes all lines from m to n to be listed.

The LIST command can also be used to output lines to a terminal or printer on another port by entering #N, after LIST (such as LIST #7, 110-130) where N is the desired port number.

Examples: LIST  
LIST 30  
LIST #3, 30-70

The LIST command can also be used as a program statement.

## LOAD (file name)

The LOAD command is used to load, from disk, a previously saved program. LOAD will clear the memory of the current program and load in the desired program. The same rules apply for file names and drive specifications as in DOS.

Example:     LOAD COMPUTE

If no extension is given, .BAS is assumed. Also, if you forget to type in the file name and simply type LOAD, BASIC will prompt you for the file name.

## MON

The MON command causes computer control to be returned to the computer's monitor.

## NEW

The NEW command causes the working storage and all variables and pointers to be reset. The effect of this command is to erase all traces of the previous program from memory. This command also sets LINE equal to 48 and DIGITS equal to 0 (floating point mode).

## PORT=X

The PORT=X command defines the computer I/O Port which will serve as the 'Control Port'. "X" can be a constant, variable, or expression.

Example:     PORT = 3

**Warning**—If you define a port without a terminal as the control port, all messages (including the "Ready") will be inputted and outputted from that port. . .therefore, you will **lose** control of your program.

**NOTE:** PIA ports **require** handshaking. If handshaking is not available, then you must use the PEEK command to examine the PIA registers. Also, BASIC will always accept a break from port 1, therefore never leave port 1 without a terminal connected. Appendix G defines the correct handshaking procedure. Each port # is configured by BASIC for the specified type of interface:

PORT	TYPE OF PORT
0	ACIA
1	MODIFIED PIA (CONTROL PORT) or ACIA
2	ACIA
3	ACIA
4	PIA
5	PIA
6	PIA
7	PIA (LINE PRINTER, BY CONVENTION, such as SWTPC PR-40)

The PORT command can also be used as a program statement.

## RUN

The RUN command causes the current program resident in memory to begin execution at the first statement number. RUN always begins at the lowest statement number. RUN resets all program parameters and initializes all variables to zero.

## SAVE (file name)

The SAVE command causes the current BASIC program in memory to be saved on disk. The same rules apply for file names and drive specifications as in DOS. If no extension is specified, .BAS is assumed. If a file already exists with the chosen name, an error message will be output.

Example:     SAVE COMPUTE

If you forget to type in the file name and enter only SAVE, BASIC will prompt for the file name.

## STRING=X

The STRING= command sets the maximum allowable length of string variables. The STRING= command may be used as part of a program and must be used before any strings are referenced in a program. X may be any number between 1 and 126. STRING is initially set to 32 characters. The NEW command will not reset the string length to 32.



## TAPPEND

The TAPPEND command works the same as the TLOAD command except that the current BASIC buffer area is not cleared before the load starts.

## TLOAD

The TLOAD command is used for loading BASIC programs previously saved on cassette and paper tapes. All input/output regarding the TLOAD command will be thru the control or defined port. Appropriate reader on/off commands are automatically generated.

Example: TLOAD

If desired, the input from TLOAD can be channeled thru a port other than the control port by using the TLOAD #N command where N is the desired port number. The same rules apply for port types and handshaking as described in the PORT= command.

NOTE: Both the TLOAD and TSAVE commands assume that the punch/read device is set up to decode automatic reader/punch on/off commands. If your particular unit is not automatic, the reader or punch should be turned on manually before the carriage return is entered after typing the respective TSAVE or TLOAD command.

## TRACE ON

The TRACE ON command will cause BASIC to display the line number of the current statement being executed for every line. This can be an important debugging tool.

## TRACE OFF

The TRACE OFF command turns off the trace function.

## TSAVE

The TSAVE command is used for saving BASIC programs onto cassette or paper tape. All output from the TSAVE command will be thru the control or designated port. Appropriate punch on/off commands are automatically generated for use by the tape storage device.

Example: TSAVE

If desired, the output from TSAVE can be directed to another port by using the TSAVE #N command where N is the desired port number. The same rules apply for port types and handshaking as described in the PORT= command.

NOTE: TSAVE will dump the entire BASIC program to tape—line numbers such as TSAVE 10 - 20 can not be entered to transfer only a portion of the program to tape.

## STATEMENTS

A statement, in BASIC, is a word or a group of words that directs the execution of a BASIC program. Statements differ from commands in that they generally do not cause the computer to immediately take action by themselves. Some statements, in fact, must be used with other statements for proper operation.

**DATA N1, N2, N3, . . .**

**READ V1, V2, V3, . . .**

**RESTORE**

The DATA, READ and RESTORE statements are used in conjunction with each other as one of the methods to assign values to variables. Every time a DATA statement is encountered, the values in the argument field are assigned sequentially to the next available position of a data buffer. All DATA statements, no matter where they occur in a program, cause DATA to be combined into **one** list.

READ statements cause values in the data buffer to be accessed sequentially and assigned to the variables named in the READ statement. They start with the first data element from the first data statement, then the second element, to the end of the first data statement, then to the first element of the second data statement, etc., each time a READ command is encountered. If a READ is executed, and the DATA statements are out of data, an error is generated.

Numeric and string data may be intermixed, however it must be called in the appropriate order.

**Note:** String data need **not** be enclosed within quotes (") as the comma (,) acts as the delimiter. However, if the string contains a (,), then it **must** be delimited by quotes (").

**Example:**

```
10 DATA 10,20,30,56.7,"TEST,ONE",1.67E30,8, HELLO
20 READ A,B,C,D,E$,F,G5,F$
```

**Note:** DATA STATEMENTS may be placed anywhere within the program.

**Example:**

```
110 DATA 1,2,3.5
120 DATA 4.5,7,70
130 DATA 80,81
140 READ B,C,D,E
```

is the equivalent of:

```
10 LET B=1
20 LET C=2
30 LET D=3.5
40 LET E=4.5
```

The RESTORE statement causes the data buffer pointer, which is advanced by the execution of READ statements, to be reset to point to the first position in the data buffer.

**Example:**

```
110 DATA 1,2,3.5
120 DATA 4.5,7,70
130 DATA 80,81
140 READ B,C
150 RESTORE
160 READ D,E
```

In this example, the variables would be assigned values equal to:

```
100 LET B=1
101 LET C=2
102 LET D=1
103 LET E=2
```

There are also versions of READ and RESTORE which are used for the manipulation of disk data files. These statements are discussed in the Disk Data Files section.

### **DIM/var/ (exp) or /var/ (exp), /var/(exp) or /var/(exp,exp)**

The DIM statement allocates memory space for an array. In this version of BASIC, one or two dimension arrays are allowed and the maximum array size is 255 x 255 elements. All array elements are set to zero by the DIM statement.

If an array is not explicitly defined by a DIM statement, it is assumed to be defined as an array of 10 elements (or 10 x 10 if two elements are used) upon first reference to it in a program.

**Caution:** The dimension of an array can be determined only once in a program, implicitly and explicitly. Also only the variables A thru Z (followed by \$) may be dimensioned for strings.

Example: DIM A(10), C(R5+8), D(30,A\*3), A7(20), C\$(30), Z\$(5)  
but not A6\$(5)

The DIM statement can also be used in the direct execution mode.

### **END**

The END statement causes the current BASIC program to stop executing. When an END statement is seen, BASIC will return to the command mode. In this version of BASIC, END may appear more than once and need not appear at all.

### **FOR /var/ = /exp 1/ TO /exp 2/ STEP /exp/ NEXT /var/**

The FOR and NEXT statements are used together for setting up program loops. A loop causes the execution of one or more statements for a specified number of times. The variable in the FOR. . .TO statement is initially set to the value of the first expression (exp1). Subsequently the statements following the FOR are executed. When the NEXT statement is encountered, the values of the named variable is added to the value specified by the STEP expression in the FOR. . . TO statement, and execution is resumed at the statement following the FOR. . .TO. If the addition of the STEP value develops a sum that is **greater than** the TO expression (exp2) or, if STEP is negative, a sum **less than** the TO expression (exp2), the next instruction executed will be the one following the NEXT statement. If no STEP is specified, a value of one is assumed. If the TO value is initially less than the initial value, the FOR. . .NEXT loop will still be executed once.

Example: 110 FOR I=.5 TO 10  
120 INPUT X  
130 PRINT I,X,X/5.6  
140 NEXT I

Although expressions are permitted for the initial, final, and STEP values in the FOR statement, they will be evaluated only the first time the loop is entered. They are not re-evaluated.

It is not possible to use the same indexed variable in two loops if they are nested.

When the statement after the NEXT statement is executed, the variable is equal to the value that caused the loop to terminate, not the TO value itself. In the above example, I would be equal to 9.5 when the loop terminates.

### **GOSUB /line #/**

A subroutine is a sequence of instructions which perform some task that would have utility in more than one place in a BASIC program. To use such a sequence in more than one place, BASIC provides subroutines and functions.

A subroutine is a program unit that receives control upon execution of a GOSUB statement. Upon completion of the subroutine, control is returned to the statement following the GOSUB by execution of a RETURN statement in the subroutine.

Example: 10 A=3  
20 GOSUB 100  
30 PRINT B  
40 END  
100 LET B= SIN(A)  
110 RETURN

## GOTO /line #/

The GOTO statement directs BASIC to execute the statements on the specified line unconditionally. Program flow continues from the line specified by /line/.

Example: 150 GOTO 270

This statement may be used in the direct execution mode.

## IF /relational exp/ THEN /statement n/

### IF /relational exp/ THEN /BASIC statement/ (Direct)

The IF statement is used to control the sequence of program statements to be executed, depending on specific conditions. If the /relational expression/ given in the IF is "true", then control is given to the line number declared after the THEN. If the relational expression is "false", program execution continues at the line following the IF statement.

Example: 10 IF 5>2 THEN 100

It is also possible to provide a BASIC statement after the THEN in the IF statement. If this is done and the relational expression is true, the BASIC statement will be executed and the program will continue at the statement or line following the IF statement.

Example: 10 IF 5>2 THEN LET B=7

When evaluating relational expressions, arithmetic operations take precedence in their usual order, and the relational operators are given equal weight and are evaluated last.

Example: 5+6\*5>15\*2 evaluates to be true

### The Relational Operators

= Equal  
<> Not Equal  
< Less Than  
> Greater Than  
<= Less Than or Equal  
>= Greater Than or Equal

Examples: 110 IF A<B+3 THEN 160  
180 IF A=B+3 THEN PRINT "VALUE A", A  
190 IF A=B THEN T1=B

NOTE: If an IF test fails on a multiple statement line, the remainder of the line will not be executed.

Example: 10 IF 5<2 THEN 100 : PRINT 3  
20 END

Control will go to line 20 and "3" will not be printed

The relational operators = (equal) and <> (not equal) may also be used on strings.

Example: 110 IF Y\$ = "YES" THEN 100

The < (less than) and > (greater than) comparisons may also be used on strings, but **only** when the number of characters in each of the strings being compared is the same. The > and < operators compare strings by evaluating the ASCII value of the characters starting from the first (leftmost) character. When a character in one string is found to be not equal to its respective character in the other string, the greater than or less than operation is made either true or false depending on the ASCII values of these two characters.

Example: IF "AAABA" > "AAAAB" THEN 100

The first non-equal character in the comparison is the B in "AAABA". The > operator then compares this B to the respective character in the other string (an A). Since the ASCII value of B is greater than that of A, the operation evaluates to "yes, greater than".

Example: "A" > "B" FALSE  
"B" > "A" TRUE  
"ABCDE" < "ABCDF" TRUE  
"ABC" > "ABCD" ILLEGAL, LENGTHS NOT EQUAL  
"BZZ" > "CZZ" FALSE

**INPUT /var/**  
**INPUT /var/, /var/, /var/, . . .**  
**INPUT #N, var**  
**INPUT "/PROMPT"/, /var/**

The INPUT statement allows users to enter data from the terminal during program execution.

Example:     **INPUT X** - Inputs one numeric value  
              **INPUT X\$** - Inputs one string value  
              **INPUT X,Y,Z,B\$** - Multiple inputs may be entered, separated by  
                                  ",". If the expected number of values are not entered,  
                                  another "?" will be generated.  
              **INPUT "ENTER VALUE",X** - Prints the message in quotes, then a  
                                  "?", and waits for input. It stores the inputted  
                                  value in X.

When the program comes to an INPUT statement, a question mark is displayed on the terminal device. The user then types in the requested data separated by commas and followed by a carriage return. If insufficient data is given, the system prompts the user with '?'. If no data is entered, or if a non-numeric character is entered, the system prompts "RE-ENTER". However, for string variables a null return will be considered as valid data. Caution: for input A\$,B\$,C\$—a null response would create three null variables. Only constants can be given in response to an INPUT statement.

The INPUT can also be used to issue a prompting message before the question mark appears.

Example:     **10 INPUT "INPUT A\$", A\$**  
              **20 PRINT A\$**  
              would give the following results  
              **INPUT A\$ ? 66** (user types this 66 in)  
              **66**

INPUT may also be used with the #N, directive for input from ports other than the control port.

**LET /var/=/exp/**

The LET statement is used to assign a value to a variable. The use of the word LET is optional unless you are in the direct mode.

Example:     **100 LET B=827**  
              **110 LET C=87E2**  
              **120 R=(X\*Y)/2\*A**  
              **130 C\$="THIS IS TEXT"**

The equal sign does not mean equivalence as in ordinary mathematics. It is the replacement operator. It says: replace the value of the variable named on the left with the value of the expression on the right. The expression on the right can be a simple numerical value or an expression composed of numerical values, variables, mathematical operators, and functions.

**ON /exp/ GOTO /line (s)/**

**ON /exp/ GOSUB /line(s)/**

This statement transfers control to the line or subroutine as defined by the value of /exp/. The expression will be evaluated, truncated (chopped off after the decimal point) and control then transferred to the nth statement number (where n is the integer value of the expression).

Example:     **ON N GOTO 110, 300, 500, 900**  
Means:        **If N <1** You will get an error  
              **If N=1** GOTO 110  
              **If N=2** GOTO 300  
              **If N=3** GOTO 500  
              **If N=4** GOTO 900  
              **If N>4** You will get an error

Example:     **ON (N+7)\*2 GOSUB 1000,2000**  
(see GOTO and GOSUB for a further description of these statements)



## FUNCTIONS

Functions are similar to BASIC statements except that they usually relate to mathematical or string processing operations.

### ABS (X)

The ABS (X) function returns the "Absolute Value" of X.

Example:           ABS (3.44)=3.44  
                    ABS (-3.44)=3.44

### ATAN (X)

The ATAN (X) function returns the angle, in radians, whose tangent is X.

### ASC (string or string var)

The ASC (string or string variable) function returns the **decimal** ASCII numeric value of the **first** ASCII character within the string. Literals must be enclosed by quotes while string variables are not.

Example:           ASC("?")       gives 63  
                    ASC("A")       gives 65  
                    ASC("B")       gives 66  
                    ASC("Z")       gives 90  
                    ASC("5")       gives 53  
                    LET B\$="5" → >ASC(B\$)       gives 53

### CHR\$ (X)

The CHR\$ (X) function returns a single character string equivalent to the decimal ASCII numeric value of X. This is the **inverse** of the ASC function.

Example:           CHR\$(63)       gives a ?  
                    CHR\$(65)       gives a A  
                    CHR\$(66)       gives a B  
                    CHR\$(53)       gives a 5

### COS(X)

The COS(X) function returns the cosine of the angle X. X must be in radians.

### DEF FN/letter/(/variable/)=/exp/

This function allows the user to create his very own functions. The /letter/ is any alphabetic character. This names the function (i.e., you could have, say, three functions named FNA, FNB, and FNC). The /variable/ is a non-subscripted numeric variable. This is essentially a "dummy" variable (or place holder). . . This will be apparent shortly. The "Expression" is any valid expression. Note that the "variable" **must** be enclosed within parenthesis.

For example, study the following sample program:

```
10 DEF FNA(X)=3.14*X↑ 2
20 DATA 5,6,7,0
30 READ X
40 IF X=0 THEN END
50 PRINT FNA(X)
60 GOTO 30
```

RUN

78.5  
113.04  
153.86

READY

As you can see, the dummy variables were replaced with the variables you actually wished to use at the time the function was used.

Note: You may **not** define the same function greater than once per program, and a function **must** be defined before it is called.

#### EXP(X)

The EXP(X) function returns the base of natural logarithms raised to the Xth power (this is the inverse of LOG(X)) and is the equivalent of 2.718282 raised to the Xth power.

#### INT(X)

The INT(X) function returns the greatest integer **less than** X.

Example: INT(4.354)=4

Now note this one: INT(-4.354)=-5

#### LEFT\$(X\$,N)

The LEFT\$(X\$,N) function returns a string of characters from the leftmost to the Nth character in X\$.

Example: X\$="ONE,TWO,THREE"

LET A\$=LEFT\$(X\$,6)

A\$ NOW EQUALS "ONE,TW"

#### LEN(X\$)

The LEN(X\$) function returns the number of characters contained in string X\$.

Example: LEN("TESTING")=7

LEN("TEST ONE")=8

Note: The space **does** count.

Hint: LEN(STR\$(X)) = The number of print positions required to print the number X.

#### LOG(X)

The LOG(X) function returns the natural logarithm of the number X.

#### MID\$(X\$,X,Y)

The MID\$(X\$,X,Y) function returns a string of characters from X\$ beginning with the Xth character from the left, and continuing for Y characters from that point. Y is **optional**. If Y is **not** specified, then the string returned is from the Xth character of the string through the end of the string.

Example: X\$="ONE,TWO,THREE"

A\$=MID\$(X\$,3,10)

A\$ NOW EQUALS "E,TWO,THRE"

#### PEEK(X)

The PEEK(X) function returns, in **decimal**, the value contained in **decimal**, not octal, memory location X.

Example: LET A=PEEK(255)

A will now contain the decimal value contained in memory location 25510.

#### POKE(I,J)

The POKE(I,J) function takes the decimal, not octal, value of J and places it in decimal, not octal, location I. For example, POKE (255, 10) will store a decimal 10 in decimal memory location 255.

Warning: This function can cause system program failure if improperly used.



## POS

The POS function returns in decimal, not octal, the current position of the print head or cursor. The first position (left margin) is position #1.

## RIGHT\$(X\$,N)

The RIGHT\$(X\$,N) function returns a string of characters from the Nth position to the left of the rightmost character through the rightmost character.

Example:                   X\$='ONE,TWO,THREE'  
                              A\$=RIGHT\$(X\$,9)  
                              AS NOW EQUALS "TWO,THREE"

## RND AND RND(X)

The RND(X) function produces a set of uniformly distributed pseudo-random numbers. If X (the seed) is 0, then each time RND(X) is accessed, a different number between 0 and 1 will be returned. If  $X \neq 0$  then a specific random number will be returned each time (the same number each time). RND can be called without an argument, in which case it works as if one had used an argument of 0.

Example:                   10 LET A=RND  
                              20 LET B=RND(5)

If you require random numbers other than between 0 and 1, then:

PRINT INT ( (B-A+1)\*RND(0)+A)

will yield random numbers ranging between A & B.

## SGN(X)

The SGN(X) function returns the 'sign' (+, -, or 0) of X. The SGN of a negative number will yield a -1, the SGN of a positive number will yield 1 and the SGN of 0 gives 0.

Example:                   SGN(4.5)=1  
                              SGN(-4.5)=-1  
                              SGN(0)=0  
                              SGN(-0)=0

## SIN(X)

The SIN(X) function returns the sine of the angle X. X must be in radians.

## SQR(X)

The SQR(X) function returns the square root of X. X must be greater than or equal to 0 (X must be positive).

## STR\$(X)

The STR\$(X) function returns the string value of a numeric value. This is the inverse of the VAL function.

Example:                   A=34567  
                              LET A\$=STR\$(A)  
                              A\$ NOW EQUALS "34567"

## TAB(X)

The TAB(X) function will move the print position to the "Xth" position to the right of the left margin. If the print position is **already** to the right of the position specified in the TAB command, **no** spaces will be left and printing (if any) will commence. The first print position (left margin) is position #1.

The TAB function can be used with the PRINT statement to cause data to be printed in exact locations. The argument of TAB may be an expression.

Example:

```
      5  X=3
     10  PRINT TAB(2); X; TAB( );X*X; TAB( ); X*X*S
will print
      3      9      27
```

## TAN(X)

The TAN(X) function returns the tangent of the angle X. X must be in radians. (360 degrees =  $2\pi$  radians  $\pi = 3.141592654$ )

## USER(X)

The USER (X) function is a BASIC function that enables a user to call a special machine language subroutine. The syntax of the USER function is of the form LET /var/=USER (/var.1/) such as LET A = USER(X). The use of the USER function assumes that the programmer is familiar with assembler level programming.

When the USER function is executed in a program, the numeric value of the variable X is stored in a special BCD (binary coded decimal) format in a seven byte series somewhere in the computer's memory. BASIC then keeps track of where this series is stored so that the machine language routine can access it. After storing this series, BASIC then looks at hex memory locations 0067 and 0068. The computer is then instructed to execute a "Jump to Subroutine" to the hex address stored in hex memory locations 0067 and 0068. To avoid accidental misuse of the USER function, 0067 and 0068 will initialize to a location which contains a hex 39, a return from subroutine. Locations 0067 and 0068 can be changed using the POKE function prior to using USER.

After the computer jumps to the location pointed to by 0067 and 0068 it is up to the machine language program to perform its special function or to manipulate the data previously stored in the seven byte BCD series. To find out where this series is located, hex memory locations 005D and 005E should be checked by the machine program. 005D and 005E contain a pointer to the **location** of the seven byte series. They do not contain the actual location of the series.

For example, say that locations 005D, 005E contain the address IDB1 This means that locations IDB1 and IDB2 contain the **address** of the seven byte series. If the series was stored beginning at 242B, then the locations would be set up as follows:

```
005D  1D
005E  B1

IDB1  24
IDB2  2B
```

242B Start of seven byte series.

The actual number that was stored in the seven byte series is stored in a special BCD format as follows:

for + numbers	for - numbers
BYTE 1 (sign) (D <sub>9</sub> )	(sign) ( $\overline{D}_9$ )
BYTE 2 (D <sub>8</sub> ) (D <sub>7</sub> )	( $\overline{D}_8$ ) ( $\overline{D}_7$ )
BYTE 3 (D <sub>6</sub> ) (D <sub>5</sub> )	( $\overline{D}_6$ ) ( $\overline{D}_5$ )
BYTE 4 (D <sub>4</sub> ) (D <sub>3</sub> )	( $\overline{D}_4$ ) ( $\overline{D}_3$ )
BYTE 5 (D <sub>2</sub> ) (D <sub>1</sub> )	( $\overline{D}_2$ ) ( $\overline{D}_1$ )
BYTE 6 (Exponent in hex)	(Exponent in hex)
BYTE 7 00	00

Where D's are digits and  $\bar{D}$ 's are the digits complemented.

The sign half-byte denotes whether or not the number is positive or negative. A sign of 0 denotes + while a 9 denotes -. The actual number digits are located in half-bytes D<sub>1</sub> - D<sub>9</sub>. The exponent byte tells BASIC where to put the decimal point. Notice that this number is hexadecimal and not BCD.

For example, the number 1234.5678 would be stored as follows:

Byte 1	01
Byte 2	23
Byte 3	45
Byte 4	67
Byte 5	80
Byte 6	04
Byte 7	00

The number is stored as .12345678 with an exponent of 4 which moves the decimal point 4 places to the right giving 1234.5678. The 0 half-byte in byte 1 denotes a positive number.

Now look at the number -1234.5678. Negative numbers are more complicated and must be handled with great care.

Byte 1	98
Byte 2	76
Byte 3	54
Byte 4	32
Byte 5	20
Byte 6	04
Byte 7	00

Notice that the first 9 in byte 1 denotes a negative number and that all digits D<sub>1</sub> - D<sub>9</sub> are complemented. The complement of a digit is defined a 9- (the digit) with the complement of 0 still being 0. In the above example, the digits that were stored were not 12345678 but rather (9-1) (9-2) (9-3) (9-4) (9-5) (9-6) (9-7) and (1+9-8). The **last significant digit** not including any trailing 0's must have 1 added to its complement before storing in the BCD series. In the example -1234.5678 (the same as -1234.56780) the last **significant** digit is 8; therefore, 1 must be added to its complement.

The number -7.20008000 would be stored as:

Byte 1	92	
Byte 2	79	
Byte 3	99	
Byte 4	20	(the last significant digit is 8)
Byte 5	00	
Byte 6	01	
Byte 7	00	

The end of the machine language program should contain a hex 39, a RTS. This will transfer control back to BASIC. BASIC will then assign the numeric value of the number in the seven byte string to the variable A in the example A=USER(X).

## VAL(X\$)

The VAL(X\$) function returns the numeric constant equivalent to the value in X\$. This is the **inverse** of the STR\$ function.

Example: VAL("12.3")=12.3  
VAL("5E4")=5000  
VAL("TWO")=GENERATES AN ERROR. "TWO" cannot be equaled to a numeric constant.  
VAL("-12.3")=-12.3

## Special Disk Commands and Disk Data Files

Below is a description of several commands which allow the user to interface with the various files stored on a disk.

### CAT (drive number)

The CAT command can be used to display all of the files on a particular drive. All files are listed, not just BASIC files. Only the names are displayed—additional information on a file's length and other disk information may be obtained by using the CAT command of DOS.

Examples:           CAT  
                  CAT 1

### CHAIN (file name), (optional line number)

#### CHAIN A\$, (option line number)

The CHAIN command can be used to call one BASIC program from another program. CHAIN will force the extension .BAS on the file name, even if another is given. If no line number is specified, program execution of the program called will begin on the first line of the program. If specified, execution will begin on the given line.

Example:           25 CHAIN MONEY 110

In the above example, the current BASIC program would be deleted from memory and the program MONEY.BAS loaded. Execution on line 110 of MONEY would then begin.

String variables may also be used in the file specification:

Example:           10 A\$="MONEY"  
                  20 CHAIN A\$, 110

### KILL (file name)

#### KILL A\$

The KILL command is used to delete a file from a disk. Care must be exercised when using the KILL command since it does not prompt with an "Are you sure" question. The same rules apply for file names and drive specifications as in DOS. The default extension for KILL is .DAT.

Example:           KILL COMPUTE.BAS

If desired, KILL may be used as a program statement and the file name may be specified as a string variable.

Example:           10 A\$= "COMPUTE.BAS"  
                  20 KILL A\$

### RENAME (file 1), (file 2)

#### RENAME A\$, B\$

The RENAME command may be used to rename the various files on a disk. The default extension is .DAT.

Example:           RENAME TEST, JUNK

The above example would rename the file TEST.DAT to the name JUNK.DAT. RENAME may also be used as a program statement and may be used with string variables.

Example:           10 A\$= "TEST"  
                  20 B\$= "JUNK"  
                  30 RENAME A\$,B \$

The same rules apply for file names and drive specifications as in DOS.

## Disk Data Files

SWTPC Disk BASIC contains the necessary statements to manipulate sequential disk data files. Data files give the user the ability to access large amounts of data on disk whenever necessary. These data files are very useful when working with things such as inventory and payroll data.

Working with disk data files is similar to using the DATA and READ statements described earlier. When beginning a new file, the file must be "opened". This "opening" essentially equates a file number that BASIC can understand to a file name that the file management system of DOS can understand. After a file is opened, the desired data can then be "written" on this file. If no more manipulation is desired, the file is "closed" (line designated file number is disassociated with the file). The file may later be re-opened and the data read from it by a BASIC program.

Below is a description of each of the file commands in BASIC and a sample program showing their use.

### OPEN #/file no./, /file name/

The OPEN command prepares a file on disk to be used for either input or output. No actual disk operation takes place when executing the OPEN statement.

BASIC programs essentially refer to files by file number rather than file name. The function of the OPEN statement is to equate a recognizable file number to a given file name. When using the OPEN statement, the /file no./ must be the number assigned to the file and must be from 0 to 9. What you choose for a file number is completely arbitrary, but each file that is open at any one given time must have a unique file number. The /file name/ specification is the name of the file as it appears on the disk. The same rules apply for /file name/ as do in DOS.

Example:           10 OPEN #1, DATA.DAT  
                  OPEN #1, DATA.DAT, #2, JUNK

If no extension is given on the file name, the extension .TXT is assumed.

Note:     Each file number that is opened requires 198 bytes. Re-using the same file number, after closing a file, in subsequent OPEN statements will save the allocation of new memory space.

### CLOSE #/file no./, #/file no./ . . .

The CLOSE statement is used to "close" an open file. The file number that is closed must have previously been opened by the OPEN statement. The CLOSE statement, in effect, disassociates the previously assigned file number with the file name. Files should always be closed when file manipulation is finished.

Example:           10 CLOSE #1, #2

### READ #/file no./, /variable list/

The READ #/file no./ statement is similar to the READ statement described earlier and is used to retrieve data off of a disk file to be used in a BASIC program. For example, a READ #1, A,B will transfer the first entry of file number 1 into variable A and the second entry into variable B. Each time a read is done from a file an internal pointer is incremented so that the next read will access the next value in the file. String and numeric variables may be intermixed in /variable list/ and their format must match with that of the file being read.

Example:     Suppose the file PYRL.DAT contains data in the following format:

	(EMPLOYEE #)	(NAME)	(HOURLY SALARY)	(HOURS WORKED)
	numeric	string	numeric	numeric
Such as	1	ADAMS	3.25	40
	2	BROWN	6.00	40
	3	JONES	4.87	40
			etc.	

The following program could be used to work on the file:

```
10 OPEN #, PYRL.DAT
20 READ #1, N, N$, S, T           READS DATA ON EMPLOYEE 1
30 PRINT N, N$, $, T, S*T
40 READ #1, N, N$, S, T           READS DATA ON EMPLOYEE 2
50 PRINT N, N$, S, T, S*T
    etc.
```

The output on the screen would be as follows:

```
1      ADAMS      3.25      40      130
2      BROWN      6.00      40      240
    etc.
```

Notice that the READ operation starts at the beginning of a file and increments its way through as data is read.

Note: If the receiving element is a string variable, it will receive the data from the file up to a maximum string length. The line input buffer for a single item from a file is 72 characters. If an attempt is made to read a string variable from the file that is longer than the string length limit of the receiving string variable, the item will be truncated at the receiver's limit. If the input string variable length is greater than the 72 character buffer limit, the buffer input processing will be terminated after 72 characters.

Both the READ and WRITE commands are "line" oriented. For example, say that data was written to a file with the following command:

```
10 WRITE #1, A, B, A$
```

This imaginary "line" then consists of the amount of space that A, B and A\$ take up. When reading from a file, a READ statement can not read more than one "line" at a time.

Example: 20 READ #1, A, B, A\$

would read the entire "line" and enter the correct values for A, B, and A\$. A statement such as:

```
20 READ #1, A, B, A$, C$, C
```

would assign the correct values to A, B and A\$ but would assign the value of  $\emptyset$  to C and set C\$ to a null since these variables attempted to read past the "line" length defined by the WRITE command. Also a statement such as:

```
20 READ #1, A, B
```

would correctly read the values of A and B with the remainder of the data on this "line" (A\$) not being used. The next READ such as 26 READ #1, A, B, A\$ would start reading on the next "line".

#### SAMPLE FILE STRUCTURE

```
A1      B1      A$1
A2      B2      A$2
A3      B3      A$3
```

└──defined line length──┘



Any reads attempted from this area would set the desired numeric variables to  $\emptyset$  and string variables to a null.

If the receiving element is a numeric variable, the input is scanned for a "break" character (a comma or a null) and that portion of the input, up to the break character, is then processed by a validation routine which verifies the number as being a valid numeric variable. If the number is invalid, an ERROR 3 message will result.

**RESTORE #/file no./, #/file no./, . . .**

The RESTORE statement causes the "where to read from" pointer for the file number specified to be set to the location of the first element of the file. The file number may be that of a file that is open for either reading or writing—the restore statement will first close the file and then re-open it for reading.

### SCRATCH #/file no./,#/file no./,...

The SCRATCH statement is used to remove an existing file from the current disk directory and then re-enter it into the directory. The file will then be re-opened for output (write). The SCRATCH statement performs the functions of delete and open for write. The old file is lost from the disk and a new file with the same name is prepared to receive data. Care should be exercised in using this statement since it will destroy the designated data file.

Example: 35 SCRATCH #5

### WRITE #/file no./, /variable list/

The WRITE statement is essentially the same as a READ statement allowing data to be written on a disk file. The file must have been previously opened for writing by either the OPEN or SCRATCH statements.

Example: 10 LET A=5  
20 LET C=6  
30 LET S\$="TEST"  
40 OPEN #1, TEST.DAT  
50 WRITE #1, A, C, S\$  
60 CLOSE #1

In the above program a file will be created by the name TEST.DAT (WRITE will create a file on disk if none exists) and the values 5 6 and TEST will be entered in the file. If the file specified currently exists on the disk, an error will result on the first execution of the WRITE statement. To insure availability of file write access, the SCRATCH statement should be used.

Note: WRITE must be followed by a list of variables only.

10 WRITE #1, 5, 6, "TEST" is not valid.

### EOF (X) X= file number

The EOF(X) command can be used to determine if the end of a file has been reached. X is the desired file number. EOF will return a 0 if not at the end and a 1 if the end has been reached.

Example: 25 IF EOF (3) = 1 PRINT "END OF FILE"

## Random Access Data Arrays

In addition to sequential disk data files described earlier, this version of disk BASIC also supports what can be called random access data arrays. Random access data arrays gives the programmer the ability to individually and selectively access any element in a given array on disk in almost exactly the same manner as is done in memory, like in the following example:

10 DIM A\$(5,5) Create a 5 x 5 string matrix  
20 LET A\$(1,2)="THIS IS 1,2" Selectively reference a particular location.

A 5 x 5 reference like the one above will easily fit in the computer's memory, but a 100 by 100 array will not. Disk random access data arrays allow these large arrays to be stored and accessed on disk rather than in memory.

## Using Random Access Arrays

In order to use a random access array (RAA) a name and number must be associated with the array. The file management system of the DOS will treat the RAA just like a file; therefore, the OPEN statement described earlier can be used.

Example: 10 OPEN #1, DATA.RAN

This associates the file DATA.RAN with the file #1 which is subsequently recognized by BASIC. By opening the file you are essentially specifying the area of "virtual memory" (disk addressed) to be used. If the specified file does not exist on the disk, the OPEN statement will automatically create it.

Once a file has been opened on disk, it must be dimensioned with a special form of the DIM statement. The general form of this special dimension statement is:

```
DIM    #/file no./,/var/(exp)  or
```

```
DIM    #/file no./,/var/(exp,exp)
```

where /var/ is a numeric or string variable name and exp is an expression defining the size of the array.

For example, a 100 x 100 numeric array could be dimensioned as DIM #1,A(100,100) and a string array could be dimensioned as DIM #1,A\$(100,100). Any dimension of up to 65535 can be specified, however, keep the size as small as possible to conserve disk space. The DIM statement essentially formats the required disk space and can take several minutes to finish the first time depending on the size of the array.

Nothing special is required to access an element in a RAA once the file is opened and dimensioned—referencing an element in the disk array is exactly the same as accessing an element in a memory stored array.

```
Example: 10 OPEN #1, DATA.RAN
          20 DIM #1, A$(100,100)
          30 LET A$(5,35)="ELEM 5.35"
```

Once access to the RAA is finished, the file should be closed by using the CLOSE statement described earlier.

### The Format of Random Access Arrays

Knowing how the actual data is stored on disk in a RAA can help in writing programs which are optimized for both disk space usage and speed.

All information that is stored on disk is stored in the form of "sectors". Each sector contains 252 usable bytes of storage. In this version of BASIC each floating point number requires 6 bytes of disk storage; therefore, 42 numbers can be stored in one sector. The number of sectors required to store a 100 by 100 numeric array would be:

$$\frac{100 \times 100}{42} = 238.1 \rightarrow 239 \text{ sectors.}$$

42

The amount of space required for a string variable depends on the string length which is set to using the STRING=L command. If STRING=32, then each string variable will hold 32 characters and will require 32 bytes. The number of string variables stored per sector can be expressed as  $252 \div L$  where L is the string length. For example, using a string length of 32,  $252 \div 32$  gives 7.875 string variables per sector. A fraction of variable cannot be stored in a sector so only 7 variables/sector are allowed and  $252 - (7)(32) = 28$  bytes per sector are wasted. Careful selection of string length will help conserve disk use.

When addressing two dimensional random access arrays, it is very important that the dimensions be accessed in the correct "order" for speed optimization. Take, for example, the 3 x 3 array A. A would be stored on disk in the following manner:

```
A(1,1)    row 1    col 1
A(1,2)    row 1    col 2
A(1,3)    row 1    col 3
A(2,1)                etc.
A(2,2)
etc.
```

Obviously it is much faster to do searches by first referencing a row than incrementing by columns. Example:

CORRECT REFERENCE ORDER	INCORRECT ORDER
A(1,1)	A(1,1)
A(1,2)	A(2,1)
A(1,3)	A(3,1)
A(2,1)	A(1,2)
A(2,2)	A(2,2)
A(2,3)	A(3,2)



Two example program parts and the required times for each show the importance:

**CORRECT WAY**

```
10 FOR I=1 to 100
20 FOR J=1 to 100
30 X= A(I,J)
40 IF I x J=10000 THEN END
50 NEXT J
60 NEXT I
```

Time reqd = 50 sec.

**INCORRECT WAY**

```
10 FOR J=1 to 100
20 FOR I=1 to 100
30 X= A(I,J)
40 IF I x J=1000 THEN END
50 NEXT I
60 NEXT J
```

Time reqd = 3 minutes

**Other Notes**

There are several other notes worth mentioning concerning random access arrays:

- 1.) All references to array elements are located relative to the start of file address. No description (name, dimension, type, etc.) is stored within the file. After originally creating an array care should be excised not to later reference it with a dimension larger than the original one.

Example:     DIM #1, A\$(10)     original  
              DIM #1, B\$(5)     ok  
              DIM #1, C\$(30)    DON'T DO!

- 2.) The same file name should never be opened under two different file numbers. Don't, for example:

```
10 OPEN #1, DATA.RAN
60 OPEN #2, DATA.RAN
```

- 3.) The sequential file commands READ, RESTORE, SCRATCH, WRITE and EOF have no use with random access arrays.

- 4.) The same random access array reference should not be used on both sides of an "equate" expression.

Example:     10 DIM #1, A(100)  
              20 LET A(95)=A(13) DON'T DO!

do this instead:

```
10 DIM #1, A(100)
20 LET B= A(13)
30 LET A(95)=B
```

## APPENDIX A

### ERROR MESSAGES

If, during the operation of BASIC, a mistake was made by the programmer, BASIC will output one of the following error messages:

ERROR #	DEFINITION
1.	Oversize variable (over 255) in TAB, CHR, subscript or "ON"
2.	Input error
3.	Illegal character or variable
4.	No ending " in print literal
5.	Dimensioning error
6.	Illegal arithmetic
7.	Line number not found
8.	Divide by zero attempted
9.	Excessive subroutine nesting (max is 8)
10.	RETURN W/O prior GOSUB
11.	Illegal variable
12.	Unrecognizable statement
13.	Parenthesis error
14.	Memory full
15.	Subscript error
16.	Excessive FOR loops active (max is 8)
17.	NEXT "X" W/O FOR Loop defining "X"
18.	Misnested FOR-NEXT
19.	READ statement error
20.	Error in ON statement
21.	Input Overflow (more than 72 characters on INput line)
22.	Syntax error in DEF statement
23.	Syntax error in FN error, or FN called on Function not defined
24.	Error in STRING Usage, or mixing of numeric and string variables
25.	String Buffer Overflow, or String Extract (in MID\$,LEFT\$, or RIGHT\$) too long
26.	I/O operation requested to a port that does not have an I/O card installed.
27.	VAL function error—string starts with a non-numeric value.
28.	LOG error—an attempt was made to determine the log of a negative number.
29.	File not open
30.	Illegal file number—must by 0-9
31.	Illegal file name
32.	File number in use
33.	Attempt to write to a file not open for write
34.	Attempt to read from a file not open for read
35.	The same virtual array reference not allowed on both sides of the = sign
36.	Can't exit while printing

## APPENDIX B

### Disk Error Messages

During any disk operation, there are several possible disk error messages:

ERROR #	DEFINITION	ERROR #	DEFINITION
0	NO ERROR	12	DELETE PROTECTED
1	ILLEGAL FUNCTION CODE	13	ILLEGAL FCB
2	FILE BUSY	14	ILLEGAL DISK ADDRESS
3	FILE EXISTS	15	DRIVE NUMBER ERROR
4	NO SUCH FILE	16	NOT READY
5	DIRECTORY ERROR	17	ACCESS DENIED
6	TOO MANY FILES	18	STATUS ERROR
7	DISK FULL	19	INDEX RANGE ERROR
8	END OF FILE	20	FMS INACTIVE
9	READ ERROR (CRC)	21	ILLEGAL FILE NAME
10	WRITE ERROR (CRC)	22	CLOSE ERROR
11	WRITE PROTECTED		

When a disk error is encountered, the output will be similar to the form "DISK ERROR #11". An additional error message may also be output at this time and will be of the form:

ERROR #F (file #) IN LINE # line no.)

If, for example, the following program was attempted with a write protected disk, the error message would be shown:

```
10 OPEN #1, DATA.DAT
20 WRITE #1, A, B
30 GOTO 3
```

would give

```
DISK ERROR #11
ERROR #F1 IN LINE #20
```

If an error #FF in line # (line no) message is encountered then an attempt was made to KILL or RENAME a non-existent file or a special file control block error was detected.

**APPENDIX C**  
**ASCII Hexadecimal to Decimal Conversion Table**

CHARACTER	HEXADECIMAL	DECIMAL	CHARACTER	HEXADECIMAL	DECIMAL	CHARACTER	HEXADECIMAL	DECIMAL
NUL	00	000	+	2B	043	V	56	086
SOH	01	001	,	2C	044	W	57	087
STX	02	002	-	2D	045	X	58	088
ETX	03	003	.	2E	046	Y	59	089
EOT	04	004	/	2F	047	Z	5A	090
END	05	005	0	30	048	(	5B	091
ACK	06	006	1	31	049	\	5C	092
BEL	07	007	2	32	050	)	5D	093
BS	08	008	3	33	051	^	5E	094
HT	09	009	4	34	052	~	5F	095
LF	0A	010	5	35	053	`	60	096
VT	0B	011	6	36	054	a	61	097
FF	0C	012	7	37	055	b	62	098
CR	0D	013	8	38	056	c	63	099
SO	0E	014	9	39	057	d	64	100
SI	0F	015	:	3A	058	e	65	101
DLE	10	016	;	3B	059	f	66	102
DC1	11	017	<	3C	060	g	67	103
DC2	12	018	=	3D	061	h	68	104
DC3	13	019	>	3E	062	i	69	105
DC4	14	020	?	3F	063	j	6A	106
NAK	15	021	@	40	064	k	6B	107
SYN	16	022	A	41	065	l	6C	108
ETB	17	023	B	42	066	m	6D	109
CAN	18	024	C	43	067	n	6E	110
EM	19	025	D	44	068	o	6F	111
SUB	1A	026	E	45	069	p	70	112
ESC	1B	027	F	46	070	q	71	113
FS	1C	028	G	47	071	r	72	114
GS	1D	029	H	48	072	s	73	115
RS	1E	030	I	49	073	t	74	116
US	1F	031	J	4A	074	u	75	117
SP	20	032	K	4B	075	v	76	118
!	21	033	L	4C	076	w	77	119
"	22	034	M	4D	077	x	78	120
#	23	035	N	4E	078	y	79	121
\$	24	036	O	4F	079	z	7A	122
%	25	037	P	50	080	{	7B	123
&	26	038	Q	51	081	/	7C	124
'	27	039	R	52	082	}	7D	125
(	28	040	S	53	083	~	7E	126
)	29	041	T	54	084	DEL	7F	127
*	2A	042	U	55	085			

## Appendix D

### Memory Map

0000 - 00FF	Input buffer and temporary variable storage
0100 - 2698	BASIC interpreter (may vary slightly—check contents of 014E for your version)
0100	Cold start address
0103	Warm start address
002A-002B	Contains the next available memory after the current BASIC source program.
002E-002F	Contains the starting address of the BASIC source program.
005D-005E	Contains the address of the current arithmetic value in use during a USER call.
0067 - 0068	Contains the pointer for USER
014E-014F	This location tells BASIC where to start allocating memory for program storage. This address may be changed by the user if desired to allow space for USER routines.
0150	Contains the number of the port which BASIC will initialize to, currently 01.
0153	Contains the ASCII value of what is output to the terminal when a BACKSPACE is entered. Currently a null for terminals generating an automatic cursor left on back-space.
0154	Contains the ASCII value of what BASIC interprets as a BREAK character, currently a 03 (CTL. C.).

Below is a list of the I/O jumps in BASIC for the various ports. For each port the first is the "output character in accumulator A" jump, the second receives input and places it in accumulator A and the third is the initialization routine for a particular type port (ACIA or PIA). This I/O can be changed at the discretion of the user if desired.

```
*PORT 0
0106 7E 04 6C JMP TAB OUTACI
0109 7E 04 5D JMP INACIA
010C 7E 03 B2 JMP ACIINZ

*PORT 1
010F 7E AD 18 JMP OUTEEE
0112 7E AD 15 JMP INEEE
0115 7E 19 BD JMP DUMRTS

*PORT 2
0118 7E 04 6C JMP OUTACI
011B 7E 04 5D JMP INACIA
011E 7E 03 B2 JMP ACIINZ

*PORT 3
0121 7E 04 6C JMP OUTACI
0124 7E 04 5D JMP INACIA
0127 7E 03 B2 JMP ACIINZ

*PORT 4
012A 7E 04 82 JMP OUTPIA
012D 7E 04 77 JMP INPIA
0130 7E 03 BD JMP PIAINZ

*PORT 5
0133 7E 04 82 JMP OUTPIA
0136 7E 04 77 JMP INPIA
0139 7E 03 BD JMP PIAINZ

*PORT 6
013C 7E 04 82 JMP OUTPIA
013F 7E 04 77 JMP INPIA
0142 7E 03 BD JMP PIAINZ

*PORT 7
0145 7E 04 82 JMP OUTPIA
0148 7E 04 77 JMP INPIA
014B 7E 03 BD JMP PIAINZ
```

**APPENDIX E**  
**Notes for Speeding up BASIC**

1. Subscripted variables take considerable time; whenever possible use non-subscripted variables.
2. Transcendental functions (SIN, COS, TAN, ATAN, EXP, LOG) are slow because of the number of calculations involved, so use them only when necessary. Also, the ↑ operator uses both the LOG and the EXP functions, so use the format A\*A to square a number.
3. BASIC searches for functions and subroutines in the source file, so place often called routines at the start of the program.
4. BASIC searches the symbol table for a referenced variable, and variables are inserted into the symbol table as they are referenced; therefore, reference a frequently called variable early in the program so that it comes early in the symbol table.
5. Numeric constants are converted each time they are encountered, so if you use a constant often, assign it to a variable and use the variable instead.

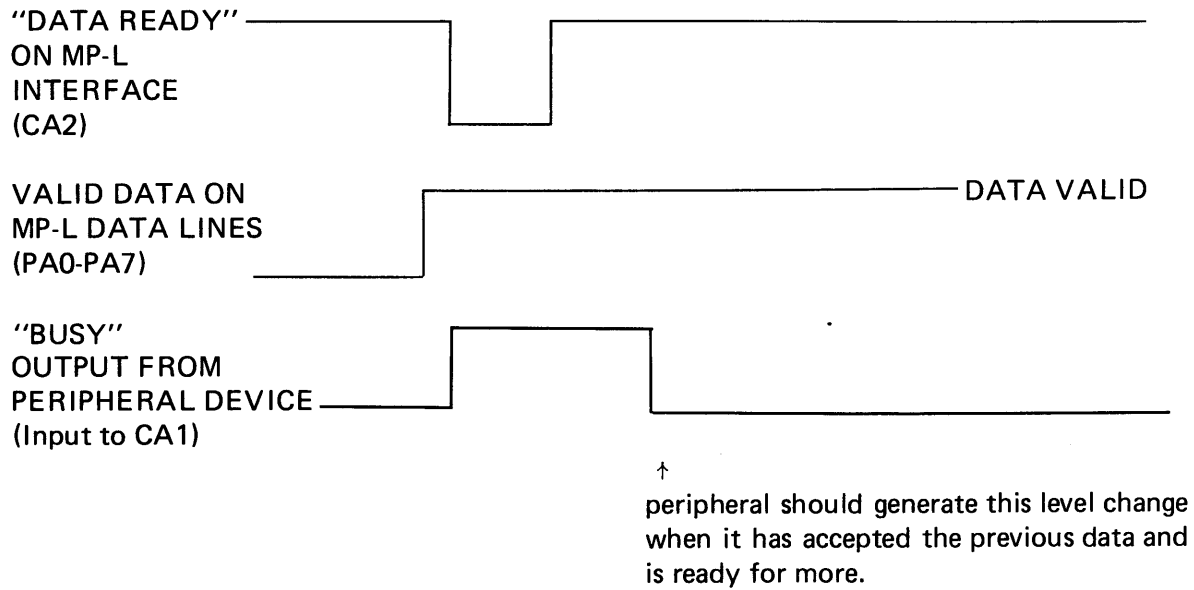
**APPENDIX F**  
**Notes on Memory Usage in BASIC**

1. REM statements use space, so use them sparingly.
- 2.a. Each non-subscripted numeric variable takes 8 bytes.  
b. Each non-subscripted string variable takes 34 bytes.  
c. Each numeric array takes 6 bytes plus 6 bytes for each element.  
d. Each string array takes 6 bytes plus 32 bytes for each element.
3. An implicitly dimensioned variable creates 10 elements (or 10 by 10). If you do not intend to use all 10 elements, save memory by explicitly dimensioning the variable.
4. Each BASIC line takes 2 bytes for the line number, 2 bytes for the encoded key word, 1 byte for the end of line terminator, 1 byte for the line length, plus one byte for each character following the key word. Memory can be saved by using as few spaces as possible.
5. BASIC reserves the uppermost 256 bytes of memory for stack and buffer use.

## APPENDIX G

### Parallel Interface Handshaking

The parallel interface drivers are written for a conventional handshaking scheme used by many printer manufacturers and is the same as that on a SWTPC PR-40 printer. Handshaking timing is set up as follows:



## Instruction Set Summary

Commands	Statements	Functions
APPEND	CHAIN	ABS(X)
CAT	CLOSE	CHR \$(X)
CONT	DATA	COS(X)
DIGITS=&	READ	DEF FN(X)
DOS &	RESTORE	EOF(X)
KILL &	FOR /exp/ to /exp/	EXP(X)
LINE= &	NEXT	INT(X)
LIST &	GOSUB	LEFT \$(X\$,N)
LOAD	DIM *	LEN(X\$)
MON	END	MID\$(X\$,S.Y)
NEW	GOTO *	PEEK(X)
PORT= &	IF /rel. exp./ THEN /line/	POKE (I,J)
RENAME &	INPUT *	POS
RUN	ON /exp/ GOT /line(s)/	RIGHT(X\$,N)
SAVE	ON /exp/ GOSUB /line(s)/	RND(X)
STRING= &	PRINT *	SGN(X)
TAPPEND	REM	SIN(X)
TLOAD	RETURN	SQR(X)
TRACE ON &	STOP	STR\$(X)
TRACE OFF &	WRITE	TAB(X)
TSAVE		TAN(X)

\* Denotes statements that may be used in the DIRECT mode  
 & Denotes commands that may be used as program statement

### MATH OPERATORS

- ↑ Exponentiate
- (unary) Negate
- \* Multiplication
- / Division
- + Addition, string concatenation
- Subtraction

### RELATIONAL OPERATORS

- = Equal (numeric and string)
- <> Not Equal (numeric and string)
- < Less Than
- > Greater Than
- <= Less Than or Equal
- >= Greater Than or Equal

- LINE NUMBERS —May be from 1 thru 9999
- VARIABLES —May be single character alphabetic or single character alphabetic followed by one integer 0 thru 9 or \$
- BACKSPACE —Is a Control H
- LINE CANCEL —Is a Control X (CANCEL)
- PROGRAM ABORT—Typing a Control C should bring BASIC back to the READY mode regardless of what the BASIC program is doing (except USER programs).
- LINES —Each line may contain multiple statements. Each statement is separated from the other with a : .



# INDEX

	Page		Page
ABS.....	13	MID\$.....	14
APPEND.....	5	MON.....	6
ASC.....	13	NEW.....	6
ATAN.....	13	NEXT.....	9
CAT.....	18	ON.....	11
CHAIN.....	18	OPEN.....	19
CHR\$.....	13	PEEK.....	14
CLOSE.....	19	POKE.....	14
COMMAND.....	1	PORT.....	6
CONCATENATION.....	3	POS.....	15
CONT.....	5	PRINT.....	12
COS.....	13	PRIORITY.....	2
DATA.....	8	RANDOM ACCESS FILES.....	21
DEF.....	13	READ.....	8, 19
DIGITS.....	5	RELATIONAL OPERATORS.....	10
DIM.....	9	REM.....	12
DISK ERRORS.....	25	RENAME.....	18
DOS.....	5	RESTORE.....	8, 20
END.....	9	RETURN.....	12
EOF.....	21	RIGHT\$.....	15
ERRORS.....	24	RND.....	15
EXP.....	14	RUN.....	6
FILES.....	19	SAVE.....	6
FOR.....	9	SCRATCH.....	21
FUNCTION.....	1	SGN.....	15
GOSUB.....	9	SIN.....	15
GOTO.....	10	SQR.....	15
HANDSHAKING.....	29	STATEMENT.....	1
IF-THEN.....	10	STOP.....	12
INPUT.....	11	STRING=.....	6
INT.....	14	STR\$.....	15
KILL.....	18	TAB.....	16
LEFT\$.....	14	TAN.....	16
LEN.....	14	TAPPEND.....	7
LET.....	11	TLOAD.....	7
LINE.....	5	TRACE ON.....	7
LINE RESTRICTIONS.....	1	TRACE OFF.....	7
LIST.....	5	TSAVE.....	7
LOAD.....	5	USER.....	16
LOG.....	14	VAL.....	17
MATHEMATICAL OPERATORS.....	2	VARIABLE.....	1
MEMORY MAP.....	27	WRITE.....	21





