CHAPTER 6
BASIC

6.1 INTRODUCTION

OS/78 BASIC* is an interactive programming language used in scientific and business environments to solve mathematical problems with a minimum of programming effort. It also is used by educators and students as a problem-solving tool and as an aid to learning through programmed instruction and simulation.

In many respects the BASIC language is similar to other programming languages (such as FORTRAN IV), but BASIC is aimed at facilitating communication between the user and the computer. BASIC simply requires that you type in the computational procedure as a series of numbered statements, making use of common English words and familiar mathematical notations. Because of the small number of commands necessary and its easy application in solving problems, BASIC is an easy computer language to learn. With experience, the advanced techniques available can be added in the language to perform more intricate manipulations or to express a problem more efficiently and concisely.

6.2 MAJOR COMPONENTS OF OS/78 BASIC

The BASIC subsystem has four major components.

1. BASIC Editor (BASIC.SV)
2. Compiler (BCOMP.SV)
3. Loader (BLOAD.SV)
4. BASIC Run Time System (BRTS.SV, BASIC.AF, BASIC.SF, BASIC.FF)

The BASIC editor is used to create and edit program source files. During this process, the editor creates a file called BASIC.WS containing the current program.

Once the program has been prepared for execution, entering a RUN command causes the editor to chain to the BASIC compiler. The compiler converts the statements in BASIC.WS into relocatable binary instructions.

Following compilation, the BASIC loader is automatically requested. The loader converts the relocatable binary data output by the compiler into executable form and loads the result into memory.

The BASIC loader then chains to the BASIC Run Time System (BRTS) which executes the program. The modules BASIC.AF, BASIC.FF, and BASIC.SF are overlays to BRTS.SV.

6.3 BASIC INSTRUCTION REPERTOIRE

BASIC instructions and commands can be grouped in three categories as follows:

1. BASIC Editor commands that allow you to

   a. Create or modify a program,
   b. Execute a program,
   c. Retrieve a program from diskette, and
   d. Save a program.

*BASIC is a registered trademark of the trustees of Dartmouth College.

2. BASIC statements, comprising the BASIC language, that are the building blocks used to create and structure BASIC programs.
3. BASIC Functions, represented by subroutines, that are built into BASIC primarily to facilitate problem solving activities.

6.4 CALLING BASIC

To enter the BASIC subsystem, type

```
.BASIC
```

in response to the Monitor dot. This command invokes the BASIC editor.

6.5 BASIC EDITOR COMMANDS

6.5.1 Using the BASIC Editor

The BASIC editor incorporates all the tools and capabilities necessary to create, correct, modify, execute, save and retrieve BASIC programs. After calling BASIC, the BASIC editor responds with the displayed query

```
NEW OR OLD--
```

Editing is now continued in one of two ways:

1. Typing NEW with a file name instructs the system to initiate the creation of a new file.
2. Typing OLD with a file name instructs the system to retrieve an old file containing a previously-generated program.

Figure 6-1 summarizes user actions implemented by the BASIC editor. This figure shows two arbitrarily selected file names (MAKER/ALTER). File names may contain no more than six alphanumeric symbols.

The left side of Figure 6-1 shows the types and patterns of activities that you ordinarily pursue after typing the NEW command. The right side of the illustration shows activities frequently carried out after typing the OLD command. Note that most of the BASIC editor commands appear on both sides of the figure. Only the sequence in which they are used differs.

6.5.2 BASIC Editor Commands

This section summarizes the BASIC editor commands. Letters that are not required in naming the command are shown as lower case in the Command/Parameter description. For example, only NE is required to be recognized by the BASIC editor as the NEW command.

NOTE
The RETURN key must be pressed following each BASIC editor command.

6.5.2.1 NEW Command – The NEW command clears the memory workspace and specifies the name of the program that is to be input.

Command	**Parameters**
NEw	file [.ex]
file.ex	is the new file name and extension of the program about to be typed in. If the extension is omitted, the editor assigns .BA.

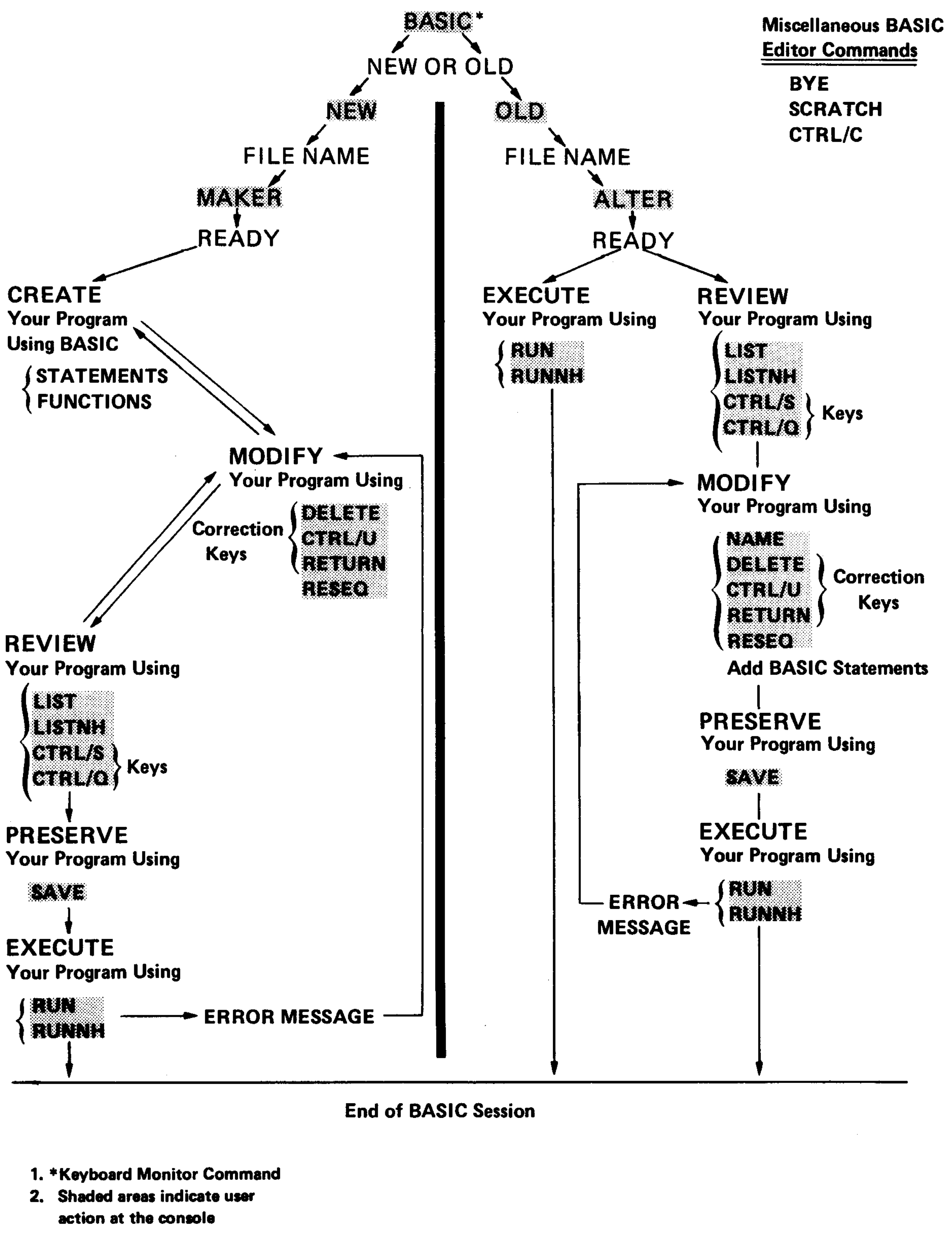
BASIC*
NEW OR OLD
NEW
FILE NAME
MAKER
READY
CREATE Your Program Using BASIC
STATEMENTS
FUNCTIONS
MODIFY Your Program Using
Correction Keys
DELETE
CTRL/U
RETURN
RESEQ
REVIEW Your Program Using
LIST
LISTNH
CTRL/S
CTRL/Q
Keys
PRESERVE Your Program Using
SAVE
EXECUTE Your Program Using
RUN
RUNNH
ERROR MESSAGE
OLD
FILE NAME
ALTER
READY
EXECUTE Your Program Using
RUN
RUNNH
REVIEW Your Program Using
LIST
LISTNH
CTRL/S
CTRL/Q
Keys
MODIFY Your Program Using
NAME
DELETE
CTRL/U
RETURN
RESEQ
Correction Keys
Add BASIC Statements
PRESERVE Your Program Using
SAVE
EXECUTE Your Program Using
RUN
RUNNH
ERROR MESSAGE
End of BASIC Session
Miscellaneous BASIC Editor Commands
BYE
SCRATCH
CTRL/C
1. *Keyboard Monitor Command
2. Shaded areas indicate user action at the console

Figure 6-1 BASIC Editor Commands and Related Uses

An alternate method is to type NEW without a file name, followed by the RETURN key. BASIC displays

```
FILE NAME --
```

in response to which the file name and extension is typed.

For example, to clear the workspace and name the new program "TEST.BA", type

```
NEW TEST
```

or

```
NE TEST.BA
```

6.5.2.2 OLD Command – The OLD command clears the memory workspace, and causes the editor to find a program on a diskette and place it into the workspace.

Command	Parameters
OLd	dev:file[.ex]
dev:file.ex	is the device, file name, and extension of the program on the disk. If the extension is omitted, BASIC assumes ".BA".

Another method is to type OLD without a file name. BASIC displays

```
FILE NAME --
```

in response to which the device, file name, and extension is typed. When no device is specified, the BASIC editor defaults to DSK (usually = SYS).

For example, to bring TEST.BA into the workspace from RXA1, type

```
OLD RXA1:TEST.BA
```

or

```
OL RXA1:TEST
```

6.5.2.3 LIST/LISTNH Commands – The LIST command displays the current program along with a header line, containing the program name, date, and the revision number of BASIC. The date is displayed only if the current date has been entered into the system.

Command	Parameters
LIst	[n]

If n is omitted all program statements in the workspace are displayed. When n is specified, line n and all subsequent lines are displayed. Type CTRL/O to terminate a listing.

For example, typing LIST (or LI) displays the program PROG.BA:

```
LIST

PROG BA        5A        26-JUL-77

10 FOR A=1 TO 5
20 PRINT A
30  NEXT A
40  END

READY
```

Typing LI 30 displays line 30 and all subsequent lines:

```
LI 30

PROG BA        5A        26-JUL-77

30 NEXT A
40 END

READY
```

The LISTNH command also displays the program statements in the workspace but without the header.

Command	**Parameters**
LISTNH	[n]
n	has the same effects as specified for the LIST command, but does not display the header.

6.5.2.4 **SAVE Command** – The SAVE command writes the program in the workspace onto the diskette as a permanently saved file. Do not confuse this command with the Monitor SAVE command.

Command	**Parameters**
SAve	[dev:file.ex]
dev	is the device on which you want to store your program.
file.ex	is the file name and extension that the program will have on the diskette. If both are left out, BASIC will use the current file name and extension of the program in the workspace. If only the extension is omitted BASIC assigns .BA. If DEV: is omitted, BASIC assumes the device is DSK.

In the following example, the program "TEST.BA" is in the workspace. To store it on RXA1 under the same file name and extension, type

```
SAVE RXA1:TEST.BA
```

or

```
SA RXA1:TEST
```

6.5.2.5 RUN/RUNNH Commands – The RUN command executes the program in the workspace. Note that this command differs from the monitor RUN command.

Command	**Parameters**
RUn	(none)

Prior to program execution, this command displays a heading consisting of the file name and extension, BASIC version number and system date, assuming that the current date has been entered in the system. When program execution is completed, BASIC displays

```
READY
```

The following example shows the program PROG.BA in the workspace displaying the result of a calculation:

```
RUN
PROG BA       5A       26-JAN-76
3.179
```

The RUNNH command executes the program in the workspace, but does not display the header line.

Command	**Parameters**
RUNNH	(none)

Thus, the only difference between the RUNNH and RUN commands is that RUN prints the header and RUNNH does not display the header.

6.5.2.6 NAME Command – The NAME command allows the user to rename the program in the workspace.

Command	**Parameters**
NAme	newfil[.ex]
newfil.ex	is the new file name and extension of the program in the workspace. If the extension is omitted, the editor assigns .BA.

To change the name of the program in the workspace to PROG.BA, type

```
NAME PROG.BA
```

or

```
NA PROG
```

6.5.2.7 SCRATCH Command – The SCRATCH command erases all statements from the workspace, that is, it clears the workspace.

Command	**Parameters**
SCratch	(none)

6.5.2.8 BYE Command – The BYE command exits from BASIC and returns control to the Monitor from BASIC.

Command	**Parameters**
BYE	(none)

Typing BYE before SAVEing a newly created program will delete the program. Whenever the BASIC editor is waiting for user input, typing CTRL/C performs the same function as BYE.

6.5.3 BASIC Control Keys

This section describes the control keys that are used to correct errors, eliminate and substitute program links, and control program listings.

6.5.3.1 Correcting Typing and Format Errors (DELETE, CTRL/U) – Errors made while typing programs at the terminal are easily corrected. Pressing the DELETE key causes deletion of the last character typed. One character is deleted each time the key is pressed.

Sometimes it is easier to delete a line being typed and retype the line rather than attempt a correction using a series of DELETEs. Typing CTRL/U will delete the entire line currently being worked on and echoe "DELETED" and a carriage return-line feed. Use of the CTRL/U key is equivalent to typing DELETEs back to the beginning of the line.

6.5.3.2 Eliminating and Substituting Program Lines (RETURN) – To delete a program line that has already been entered into the computer, simply type the line number and then press the RETURN key. Both the line number and the statement(s) are removed from the program.

Change individual lines by simply retyping them in again. Whenever a line is entered, it replaces any existing line having the same line number. New lines may be inserted anywhere in the program by giving them unique line numbers.

6.5.3.3 Interrupting Program Execution – Program execution may be terminated by typing CTRL/C. BASIC responds by displaying the READY message allowing you to correct or add statements to the program.

NOTE

BASIC responds to CTRL/C with a "READY" message only if you have already given the RUN command. Typing CTRL/C when the BASIC Editor is operational causes a return to the Monitor.

6.5.3.4 Controlling Program Listings at the Terminal Console (CTRL/S, CTRL/C and CTRL/O) – For programs exceeding a single display frame (24 lines) the user may wish to stop the scrolling effect that occurs after typing the LIST/LISTNH command. Three sets of control keys are provided to do this. They are as follows:

1. CTRL/S keys. Simultaneously pressing these keys suspends listing (scrolling) of the program. However it leaves the program in a state where you may resume listing.
2. CTRL/Q keys. Simultaneously pressing these keys (after having suspended scrolling with CTRL/S), resumes the listing process.
3. CTRL/O keys. Simultaneously pressing these keys aborts the listing process and causes the BASIC editor to display READY.

6.5.4 Resequencing Programs (RESEQ)

If a program is extensively modified, you may find that some portions of the program have line numbers spaced so closely together that they do not permit any further addition of statements. Renumbering the lines in the program to provide a practical increment between line numbers can be done by using the RESEQ program. Note that the RESEQ program modifies the line numbers in GOSUB and IF-THEN statements to agree with the new line numbers assigned to program statements by RESEQ. Line lengths must not exceed 80 characters and programs may not exceed 350 lines.

Typically, the program would be used as follows:

SAVE DSK:SAMPLE BA	User saves program SAMPLE which requires renumbering.
READY	BASIC is ready for next command.
OLD DSK:RESEQ	User calls for program RESEQ.
READY	BASIC is ready for next command.
RUNNH	User runs RESEQ program.
FILE:DSK:SAMPLE.BA	Program asks for filename. User responds with device, name, and extension of program to be renumbered.
START,STEP:100,10	Program asks for a starting line number (START) and for the increment between line numbers (STEP). User requests that SAMPLE start with line number 100 and each line be incremented by 10.
READY	Renumbering is accomplished. BASIC ready for next command.
OLD DSK:SAMPLE.BA	User calls back his program.
READY	BASIC ready for next command.
LISTNH	User gets listing of program SAMPLE for further modification.

6.6 DATA FORMATS ACCEPTABLE TO BASIC

6.6.1 Numeric Information

6.6.2 Numbers

Numbers are expressed in decimal or E (exponential) format. Examples of numbers in both categories are as follows:

Decimal	**E Type**
0	10.23E27
7	6.21E+27
+69	-7.232E6
-52	2.211E-3
-3.9265	-2.2114E-4
0.123	
-0.769	

In the decimal format, the decimal point is optional for integers. That is, BASIC assumes a decimal point after the rightmost digit of an integer. In E type format, BASIC assumes a positive exponent when no plus (+) or minus (-) sign follows the E. Substitute the words "times ten to the power of" for the letter E when reading E type format numbers.

Numeric data may be input in either format. Results of computations with an absolute value outside the range +.000001<N<999999 are always output in E type format. BASIC handles six significant digits as shown by the following examples:

Value Typed In	Value Output by BASIC
.01	0.0099999
.0099	0.0099
999999	999999
100000	.100000E+007
.0000009	.899999E-006

Note in the above examples that the nature of the binary numbering system does not permit a completely accurate representation of certain decimal numbers. Hence they are output as close to the true value as the internal logic of the computer permits.

BASIC automatically suppresses the printing of leading and trailing zeros in integer numbers and all but one leading zero in decimal numbers. As can be seen from the preceding examples, BASIC formats all exponential numbers in the form:

sign .xxxxxxE(+or-)n

where x represents the number carried to six decimal places, E stands for "times 10 to the power of", and n represents the exponential value.

For example,

-.347021E+009 is equal to -347,021,000, and

.726000E-003 is equal to 0.000726

All numbers used in BASIC must have an absolute value (N) in the range:

10^-616<N<10^+616

6.6.3 Simple Variables

A simple variable in BASIC is an algebraic symbol representing a number, and is formed by a single letter or a letter followed by a digit. For example,

Acceptable Variables	Unacceptable Variables
J	2C (a digit cannot begin a variable)
B3	AB (two or more letters cannot form a variable)

Values may be assigned to variables either by indicating the values in LET statements, or by inputting the values as data via INPUT and DATA statements.

Examples:

```
10 LET I=53721
20 LET B3=456.9
30 LET X=20E9
40 INPUT Q
50 DATA 5,6,7
```

6.6.4 Subscripted Variables

In addition to simple variables, BASIC accepts another class of variables called subscripted variables. Subscripted variables provide additional computing capabilities for handling arrays, lists, tables, matrices, or any set of related variables. Variables are allowed one or two subscripts. A single letter or a letter followed by a digit forms the name of the variable. The subscript is formed by one or two integers enclosed in parentheses and separated by commas. Up to 31 arrays are possible in any program, subject only to the amount of memory available for data storage. For example, an array might be described as A(J) where J goes from 1 to 5, as follows:

A(1),A(2),A(3),A(4),A(5)

This allows reference to be made to each of the five elements in the array A. A two-dimensional array A(J,K) can be defined in a similar manner, but the subscripted variable A must always have the same number of subscripts (that is, A(J) and A(J,K) cannot be used in the same program). It is possible, however, to use the same variable name as both a subscripted and an unsubscripted variable. Both A and A(J) are valid variable names and can be used in the same program. For more information on arrays and the use of subscripts, see the description of the DIM statement (Section 6.7.10.1).

6.6.5 Arithmetic Operations

6.6.5.1 Operators – BASIC performs addition, subtraction, multiplication, division and exponentiation, as well as more complicated operations (explained in detail later in the manual). The five operators used in writing most formulas are:

Symbol Operator	Meaning	Example
+	Addition	A+B
-	Subtraction	A-B
*	Multiplication	A*B
/	Division	A/B
^(or**)	Exponentiation (Raise A to the B Power)	A^B or (A**B)

6.6.5.2 Priority – In any given mathematical formula, BASIC performs the arithmetic operations in the following order:

1. Parentheses receive top priority. Any expression within parentheses is evaluated before an unparenthesized expression.
2. In the absence of parentheses, the order or priority is:

   a. Exponentiation
   b. Multiplication and Division (of equal priority)
   c. Addition and Subtraction (of equal priority)

3. If either sequence 1 or 2 above does not clearly designate the order of priority, then the evaluation of expressions proceeds from left to right.

The expression A+B-C is evaluated from left to right as follows:

1. A+B   = step 1
2. (result of step 1)-C = answer

The expression A/B*C is also evaluated from left to right since multiplication and division are of equal priority:

1. A/B   = step 1
2. (result of step 1)*C = answer

6.6.5.3 Parentheses – Parentheses may be used to change the order or priority because expressions within parenthesis are always evaluated first. Thus, by enclosing expressions appropriately, the order of evaluation can be controlled. Parentheses may be nested, that is, enclosed by one or more sets of parentheses. In this case, the expression within the innermost parentheses is evaluated first, and then the next innermost, and so on, until all have been evaluated. Consider the following example:

```
A=7*((B^2+4)/X)
```

The order of priority is:

1. B^2   = step 1
2. (result of step 1)+4 = step 2
3. (result of step 2)/X = step 3
4. (result of step 3)*7 = answer

Parentheses also prevent any confusion or doubt as to how the expression is evaluated. For example,

```
A*B^2/7+B/C+D^2
((A*B^2)/7)+((B/C)+D^2)
```

Both of these formulas will be executed in the same way. However, the second example may be easier to understand. Spaces may also be used to increase readability. Since the BASIC compiler ignores spaces, the two statements:

```
10 LET B=D^2+1
10LETB=D^2+1
```

are identical, but spaces in the first statement provide ease in reading.

6.6.5.4 Rules for Exponentiation – The following rules apply in evaluating the expression A^B.

1. If B=0, then A^B=1	3^0=1
2. If A=0 and B>0, then A^B=0	0^2=0
3. If A=0 and B<0, then A^B=0 and a DV error message is displayed	0^-2=0
4. If B is an integer >9, then A^B=A_1*A_2*A_3. . .*A_n, where n=B	3^5=3*3*3*3*3=243
5. If B is an integer <0 then A^B=1/A_1*A_2*A_3. . .*A_n), where n=B	3^-5=1/243
6. If B is a decimal (non-integer) and A>0, then A^B=EXP(B*LOG(A))	2^3.6=$e^{B\ln A}$=$e^{3.6\ln 2}$
7. If B is a positive or negative decimal (non-integer) and A>0, the program halts and an EM error is displayed. <	-3^2.6 is illegal.

6.6.5.5 Relational Operators – A program may require that two values be compared at some point to discover their relation to one another. To accomplish this, BASIC makes use of the following relational operators:

=	equal
<	less than
=< or <=	less than or equal to
>	greater than
=> or >=	greater than or equal to
>< or <>	not equal to

Depending upon the result of the comparison, control of program execution may be directed to another part of the program. Relational operators are used in conjunction with the IF-THEN statement.

The meaning of the (=) sign should be clarified. In algebraic notation, the formula X=X+1 is meaningless. However, in BASIC (and most computer languages), the equal sign designates *replacement rather than equality*. Thus, this formula is actually translated "add one to the current value of X and store the new result back in the same variable X". Whatever value has previously been assigned to X will be combined with the value 1. An expression such as A=B+C instructs the computer to add the values of B and C and store the result in a third variable A. The variable A is not being evaluated in terms of any previously assigned value, but only in terms of B and C. Therefore, if A has been assigned any value prior to its use in this statement, the old value is lost; it is replaced instead by the value of B+C.

6.6.6 String Information

The previous sections dealt only with *numerical information*. However, BASIC also processes alphanumerical information called strings. A string is a sequence of characters, each of which is a letter, a digit, a space, or some character other than a statement terminator (backslash or carriage return).

6.6.6.1 String Character Set – The character set recognized by BASIC is as shown below. The decimal code for each character is also shown.

Code Number		Code Number		
0	@	18	R	
1	A	19	S	
2	B	20	T	
3	C	21	U	
4	D	22	V	
5	E	23	W	
6	F	24	X	
7	G	25	Y	
8	H	26	Z	
9	I	27	[	(left bracket)
10	J	28	\	(back slash)
11	K	29	]	(right bracket)
12	L	30	¯	(exponent sign)
13	M	31	_	(underscore)
14	N	32		(space)
15	O	33	!	(exclamation point)
16	P	34	"	(double quotes)
17	Q	35	#	

Code Number		**Code Number**	
36	$	50	2
37	%	51	3
38	& (ampersand)	52	4
39	' (apostrophe)	53	5
40	(	54	6
41	)	55	7
42	*	56	8
43	+	57	9
44	, (comma)	58	:
45	- (hyphen or minus sign)	59	;
46	. (period)	60	< (left-angle bracket)
47	/ (slash or division sign)	61	=
48	0	62	> (right-angle bracket)
49	1	63	?

6.6.6.2 String Conventions – Strings may be used as constants or variables. String constants are enclosed in quotes. For example,

"THIS IS A STRING CONSTANT"

A string variable consists of a letter followed by a dollar sign ($) or a letter and a single digit followed by a dollar sign. A$ and A1$ are both legitimate string variable names while 2A$ and AA$ are not.

A string variable may contain at most eight characters unless it has been dimensioned with the DIM statement. Quotation marks may be included in strings by indicating two quotation marks in succession. For example, the string A"B would appear in a program as:

```
10 LET A$="A""B"
```

The following lines:

```
10 LET A$="""QUOTE"""
20 PRINT A$
99 END
```

will result in this display:

```
"QUOTE"
```

It is important to recognize the real, structural difference between strings and numerical data. This number

2

is not identical to this string:

"2"

Numerical data may not be used where strings are required, and vice-versa.

6.6.6.3 **String Concatenation** – Strings can be concatenated, that is, connected like links in a chain, by using the ampersand (&). For example, as a result of the following lines, the next statement executed will be line 460:

```
400 LET A1$ = "AB"
410 LET B1$ = "CD"
420 LET C1$ = "ABCD"
430 IF C1$=A1$&B1$ GOTO 460
```

The ampersand can be used to concatenate string expressions wherever a string expression is legal, with the exception that information cannot be stored by means of a LET statement in concatenated string variables. That is, concatenated string variables cannot appear to the left of the equal sign in a LET statement. For example, this statement is legal

```
LET A$=B$&C$
```

while this statement is not:

```
LET A$&B$=C$
```

6.6.7 **Format Control Characters**
In OS/78 BASIC, a terminal line is formatted into five fixed zones (called print zones) of 14 columns each. A program such as:

```
1 LET A=2.3
2 LET B=21
3 LET C=156.75
4 LET D=1.134
5 LET E=23.4
10 PRINT A,B,C,D,E
15 END
```

where the control character comma (,) is used to separate the variables in the PRINT statement, will cause the values of the variables to be displayed, using all five zones. For example,

```
RUNNH

2.3           21            156.75        1.134         23.4

READY
```

It is not necessary to use the standard five zone format for output. The control character semicolon (;) causes the text or data to be output immediately after the last character printed.

The following example program illustrates the use of the control characters in PRINT statements.

```
5  READ A,B,C,D,E,F
10 PRINT A,B,C,D,E,F
15 PRINT
20 PRINT A;B;C;D;E;F
25 DATA 4,5,6
30 DATA 16,25,36
```

```
RUNNH
 4              5           6             16            25
 36

 4 5 6 16 25 36

READY
```

As this example illustrates, when more than five variables are listed in the PRINT statement, OS/78 BASIC automatically moves the sixth number to the beginning of the next line.

6.6.8 Files

Files are referenced symbolically by a name of up to six alphanumeric characters followed, optionally, by a period and an extension of two alphanumeric characters. The extension to a file name is generally used as an aid for remembering the format of a file.

A fixed length file is one which is already in existence. That is, it has been created and CLOSEd. The length of a fixed length file is equal to the number of blocks in the file and cannot be changed.

A variable length file is a newly created file. Until the file is CLOSEd, it is equal in length to the largest free space on the device. When the file is CLOSEd it becomes a fixed length file equal in length to the actual number of blocks it occupies. Unless the file is CLOSEd, the CHAIN, STOP or END statements will cause a loss of the file.

6.7 BASIC STATEMENTS

BASIC statements are the principal components of BASIC programs. The general format of BASIC statements is:

```
xxx     Word Type  Parameters
```

where

xxx is the line number;

Word Type is the statement (instruction) type; and

Parameters are the variables used in conjunction with the statement type.

Each statement starts with a line number followed by the word statement type. Spaces have no significance in BASIC language statements except in messages or literal strings which are displayed or printed out. Thus, spaces may, but need not, be used to modify a program and make it more readable.

Multiple statements may be placed on a single line by separating each statement from the preceding statement with a backslash. For example:

```
10 LET A=5\LETB=.2\LETC=3\PRINT "ENTER DATA"
```

All of the statements in line 10 will be executed before BASIC continues to the next line. Only one statement number at the beginning of th entire line is necessary. However, it should be remembered that program control cannot be transferred to a statement within a line, only to the first statement of the line in which it is contained. This consideration is important when using control statements or loop statements (see related descriptions).

6.7.1 Statement Line Numbers (Sequencing)

Failure to assign a line number results in the message:

```
WHAT
```

Each line of the program begins with a line number of 1 to 5 digits that serves to identify the line as a statement. The largest allowable line number is 99999.

A common programming practice is to number lines by fives or tens, so that additional lines may be inserted in a program without the necessity of renumbering lines already present. Renumbering a program can be accomplished by using the RESEQ program (Section 6.5.4).

6.7.2 The PRINT Statement

The PRINT statement is used to perform calculations and display results. It is also used to display alphanumeric (string) messages.

PRINT Statement Form

Line Number	**Statement**	**Parameter**
xxx	PRINT	expressions

Where expressions may be numbers, variables, strings or arithmetic expressions, separated by commas or semicolons. When used without an expression, a blank line will be output on the terminal.

In BASIC, a line is formatted into five fixed zones (called print zones) of 14 columns each. If the expressions in a PRINT statement are separated by commas, each will begin at a 14-column interval. That is, the first expression will be displayed starting at the first position, the second at the fifteenth position, and so forth. If more than five variables are involved, the display will automatically continue at the beginning of the next line.

If this format is not desired, you may separate expressions with the semicolon (;), causing the text or data to be output immediately after the last character printed (see Example 3).

If the last expression in a PRINT statement is followed by a comma or semicolon, the next display called for by the program will begin on the same line (if there is room). This also applies to the question mark displayed by the INPUT statement (see Examples 2 and 4).

Any algebraic expression in a PRINT statement will be evaluated using the current value of the variables.

Regardless of format (integer, decimal, or E-type), BASIC prints numbers in the form:

sign number space

where the sign is either minus (-) or blank (for plus) and a blank space always follows the number (see Example 3).

Strings and numeric expressions may be combined in a single PRINT statement (see Example 3).

To output an alphanumeric message, enclose the expression in quotation marks. To print a quotation mark (") put two quotation marks ("") in the expression (see Example 5).

The following examples illustrate the use of the PRINT statement.

Example 1:

The following lines

```
40 LET A=1
50 LET B=2
55 LET C=3
60 PRINT A,B,C
99 END
```

will cause each variable to begin at a 14-column interval as follows:

```
1             2             3
```

Example 2:

The following lines

```
110 LET A$="PRINTING"
120 LET B$= "STRINGS"
130 PRINT A$,
140 PRINT B$
199 END
```

will display the following:

```
PRINTING      STRINGS
```

Example 3:

The following lines:

```
10 A=5
20 PRINT "NUMBER";A;"AND";6
99 END
```

will cause this display:

```
NUMBER 5 AND 6
```

Blanks appear before the 5 and 6 because they are positive. The blank following the 5 accounts for the blank that BASIC always generates after a number.

Example 4:

The following lines

```
60 PRINT "NUMBER OF YEARS";
70 INPUT N
99 END
```

will display (the number 9 is typed in by the user) the following:

```
NUMBER OF YEARS
?9
```

Example 5:

The following lines

```
80 PRINT "MESSAGE"
90 PRINT "A""B"
100 PRINT """QUOTE"""
199 END
```

will cause the following display:

```
MESSAGE
A"B
"QUOTE"
```

6.7.3 Information Entry Statements (DATA, READ)
The DATA and READ statements are always used together. The DATA statement is used to set up a list of numeric or string values. These values are accessed by the READ statement which assigns those values to variables in a program.

6.7.3.1 DATA Statement Format – The DATA statement sets up a list of values to be used by the READ statement.

DATA Statement Form

Line Number	Statement	Parameters
xxx	DATA	Values

where values are numeric and/or string entries separated by commas. The DATA statement serves as a source of input variables for the program. The variables are accessed for processing by the READ statement.

There may be any number of DATA statements in a program. They are not executed and may be placed anywhere within the program. However, BASIC treats them all together as a single list.

Both string data and numeric data may be intermixed in a single DATA statement.

String data in a DATA list must always be enclosed by quotation marks.

For example, the three DATA statements

```
30 DATA "FIRST"
120 DATA 2, "SECOND"
240 DATA 3, "THIRD", 4
```

are equivalent to the following statement:

```
240 DATA "FIRST",2,"SECOND",3,"THIRD",4
```

6.7.3.2 READ Statement Format – The READ statement accesses variables defined by the DATA statement and assigns those values to variables in a program.

READ Statement Form

Line Number	**Statement**	**Parameters**
xxx	READ	variables

where variables are names corresponding to values contained in DATA statements. The following examples illustrate the use of the READ statement.

Variable names must be separated by commas.

A READ statement may contain numeric and string variable names intermixed, to correspond to numeric and alphanumeric values in the DATA list. However, since values are taken from the DATA list in sequential order, you must insure that the values in the list are in the correct sequence to correspond to the variable names of the same format (numeric or string).

A READ statement may have more or fewer variables than there are values in any one DATA statement. The READ statement causes BASIC to search all available DATA statements in the order of their line numbers until values are found for each variable in the READ. A second READ statement will begin reading values where the first one stopped.

The READ statement is always used in combination with the DATA statement.

Example 1:

The following lines

```
10 READ A,B,C
20 DATA 1,2,3
```

will set variable A equal to 1, B equal to 2, and C equal to 3.

Example 2:

These statements

```
50 READ C,D$,E,F$(K)
60 DATA 5, "AAA",12, "WORD"
```

will set:

C=5
D$="AAA"
E=12
F=$="WORD"

Example 3:

The following program will display the first and third variables in a DATA list:

```
50 READ A$,B$,C$
60 PRINT A$,C$
70 DATA "DIS","XXX","PLAY"
99 END
```

The screen will show:

```
    DISPLAY
```

Example 4:

The following program uses DATA statements to supply both a variable number of scores and variable score values to an average calculation routine.

```
    100 PRINT 'NUMBER'
    110 PRINT 'OF SCORES','AVERAGE'
    120 PRINT
    125 READ N1
    127 FOR I=1 TO N1
    130 READ N
    135 IF N=0 GOTO 999
    140 LET S=0
    150 FOR K=1 TO N
    160 READ T
    170 LET S=S+T
    180 NEXT K
    190 PRINT N,S/N
    200 NEXT I
    890 DATA 5
    900 DATA 3,82,88,97
    910 DATA 5,66,78,71,82,75
    920 DATA 4,82,86,100,91
    930 DATA 4,72,82,73,82
    940 DATA 6,61,73,67,80,84,79
    999 END

    RUNNH

    NUMBER
    OF SCORES AVERAGE

     3        89
     5        74.4
     4        89.75
     4        77.25
     6        74

     READY
```

6.7.4 LET Statement

The LET statement assigns the value of an algebraic expression to a variable. Use of the word LET is optional.

LET Statement Form

Line Number	**Statement**	**Parameters**
xxx	LET	v = expression

where

v is a variable; and

expression is a number, another variable, a string (enclosed in quotes), or an arithmetic expression.

The following examples illustrate the use of the LET statement.

Example 1:

The numeric variable A is set equal to 5 by the statement

```
10 LET A = 5
```

Example 2:

The string variable A$ is set equal to "XYZ" by the statement

```
10 A$ = "XYZ"
```

Example 3:

An element (3,2) in array A is set equal to an element (1,4) in array B by the statement

```
10 LET A(3,2) = B(1,4)
```

NOTE

The LET statement does not necessarily imply an equality. LET means "evaluate the expression to the right of the equal sign and assign this value to the variable on the left". Thus, the statement

```
L=L+1
```

means "set L equal to a value one greater than it was before".

6.7.5 Loops (FOR and NEXT Statement)

6.7.5.1 FOR Statement Format – The FOR statement is used in combination with the NEXT statement to specify program loops.

FOR Statement Form

Line Number	Statement	Parameters			
xxx	FOR	V = x to y [STEP z]			
		LOOP INDEX	INDEX INITIAL VALUE	INDEX TERMINAL VALUE	INDEX STEP VALUE

where

V is a variable that serves as the INDEX for the loop. The index value is incremented (or decremented) each time the loop is executed.

x	is an expression (numerical value, variable name, or mathematical expression) indicating the initial value of the index, that is, the value it will have before the loop is executed the first time;
y	is the terminal value of the index. When v is "beyond" y, the loop will not be repeated; and
z	is the step value, that is, it is the value that is added to the index each time the loop is executed. Variable z may be assigned a minus value in cases where it is desired to decrement the index. If "STEP Z" is omitted, a value +1 is assumed.

The x, y, and z values are expressions: these expressions are evaluated upon first encountering the loop. This includes setting the index equal to the initial value. Therefore, the program can later jump back to the same FOR statement any number of times to re-start the loop. If the x, y, and z values are unchanged, the loop will be repeated the same number of times each time the program executes the FOR statement.

A variable used as an index in a FOR statement must not be subscripted.

The block of instructions to be executed repeatedly will immediately follow the FOR statement. After the last of these instructions there must be a NEXT statement whose parameter is the same as the index of the loop.

If the initial value is "beyond" the terminal value, the loop will never execute because an initial check is made of the starting and terminal values before the loop is executed. "Beyond", as used here, means that the initial value is not equal to the terminal value and adding the step value will only increase the difference. If the step value is negative "beyond" means "less than". If the step value is positive, " beyond" means "greater than".

The value of the loop index can be changed within the loop, thus influencing the number of times the loop is executed.

A program can have one or more loops within a loop. This is called "nesting loops", and is often used with subscripted variables.

It is possible to exit from a loop without the index reaching the terminal value by using an IF statement. Control may transfer into a loop only if that loop was left earlier without being completed.

The following examples illustrate the use of the FOR statement.

Example 1:

The following program

```
10 DIM C(2,3)
20 FOR A=1 TO 3
30 FOR B=1 TO 2
40 READ C(B,A)
50 PRINT C(B,A), B;A
60 NEXT B
70 NEXT A
80 DATA 1,2,3,4,5,6
90 END
```

will display:

```
1    1 1
2    2 1
3    1 2
4    2 2
5    1 3
6    2 3
```

Lines 30 through 60 of the above program represent a loop nested within another loop (line 20 through 70). The FOR statement on line 30 will be executed three times. Each time, its loop statements (lines 40 through 60) will be repeated twice.

Example 2:

The following lines

```
10 LET B=2\C=3
20 FOR A=C-B TO B^C STEP C
30 PRINT A
40 NEXT A
50 PRINT A
99 END
```

will display:

```
1
4
7
7
```

When the FOR statement was encountered, it interpreted the initial value as 1 (3 minus 2), the terminal value as 8 (2 to the third power) and the step value as 3.

Example 3:

The following loop

```
10 FOR I=10 TO 1 STEP -1
20 NEXT I
30 PRINT I
99 END
```

will display:

```
1
```

Example 4:

The following loop

```
10 FOR D=1 TO 5
20 LET D=D+4
30 NEXT D
99 END
```

will only be executed once.

Example 5:

The loops in the following program will never be executed:

```
10 FOR D=5 TO 1
20 PRINT D
30 NEXT D
10 FOR D=1 TO 5 STEP -1
20 PRINT D
30 NEXT D
99 END
```

6.7.5.2 NEXT Statement Format – The NEXT statement defines the end of a program loop.

NEXT Statement Form

Line Number	Statement	Parameters
xxx	NEXT	v (variable)

where v represents the index value used in the corresponding FOR statement.

A NEXT statement is never used without a FOR statement. The variable value must be identical to the index value (v) used in the corresponding FOR statement. The variable parameter may not be subscripted.

6.7.6 Control Statements (GOTO, IF-THEN, GOSUB, RETURN)

Normally the statements in a BASIC program are executed in the sequence they appear in a program. Control statements allow this execution sequence to be redirected.

6.7.6.1 Unconditional Branch (GOTO Statement) – The GOTO statement is used to transfer control to another line statement in a program. BASIC then continues execution at the line number referred to in the GOTO statement.

GOTO Statement Form

Line Number	Statement	Parameter
xxx	GOTO	n

where n is the line number of the statement to which control should be transferred.

For example, the following program employs two separate GOTO statements to redirect program control.

```
10 GOTO 40
20 PRINT "SECOND"
30 STOP
40 PRINT "FIRST"
50 GOTO 20
99 END
```

When executed, the program displays:

```
FIRST
SECOND
```

NOTE

When the program reaches the GOTO statement, the statements immediately following will not be executed; instead, execution is transferred to the statement beginning with the line number indicated.

6.7.6.2 Conditional Branch (IF-THEN Statement) – The IF-THEN statement tests a condition and redirects program control if that condition is true. That is, if the condition specified is true, the IF-THEN statement effectively executes a GOTO statement for the line number specified. When the condition is false, the next statement is executed.

Line Number	Statement	Parameters
xxx	IF	v1 relation v2 THEN n

where

v1, v2	are variables, numbers, strings, or expressions to be compared;
relation	is the relational operator to be used in comparing v1 and v2 (See Section 6.6.5.4);
THEN	may be replaced with GOTO if desired (either THEN or GOTO is acceptable, but one of the two must be present); and
n	is the number of the statement to which the program will jump if the relationship described is true.

The following examples illustrate the IF-THEN statement.

Example 1:

After executing the first two instructions

```
20 LET A=5
30 IF A =>2 GOTO 100
40 PRINT 'NO'
100 PRINT 'HERE'
```

control is transferred to line 100.

Example 2:

After executing the following instructions

```
20 LET A=5
30 IF A=2 GOTO 100
40 PRINT 'NO'
99 END
```

the console will display:

```
NO
```

After executing the following instructions

```
10 LET A$="*"
20 IF A$= "Z" GOTO 99
30 PRINT "YES"
99 END
```

the program will display:

```
YES
```

because the ASCII code for "*" is not equal to the ASCII code ASCII for a "Z".

Strings may be used in IF-THEN statements, but comparisons are based on the positions of the characters in the string sequence. The question mark (?) is the highest alphanumeric character and the at sign (@) is the lowest.

The two strings described in the IF-THEN statement are compared one character at a time, from left to right, until the ends of the strings are reached or until an inequality is found.

If the strings are of unequal length, BASIC lengthens the shorter string by adding spaces to the right until both are of equal length. If "AB" is compared to a four-character string, it will be treated as "AB(space) (space".

When using numerical expressions in the IF-THEN statement, the test for equality may not always work due to the nature of the arithmetic used by the computer. One way to get around this is to compare the absolute value of the difference between the operands to a very small number. For example, instead of

```
20 IF A=B THEN 50
```

use

```
20 IF ABS(B) < 0001 THEN 50
```

6.7.6.3 Branch to Subroutine (GOSUB Statements) – A subroutine is a group of statements that perform a processing operation at more than one point in a program.

The GOSUB statement is used to branch to the subroutine and the RETURN statement redirects control back to the main body of the program.

GOSUB Statement Form

Line Number	**Statement**	**Parameters**
xxx	GOSUB	n

where n is the line number of the first line of the subroutine.

When the program encounters a GOSUB statement, the following action occurs:

1. BASIC internally records the number of the statement following the GOSUB statement.
2. Control is transferred to statement number n.

GOSUB is always used with the RETURN statement.

The RETURN statement is the last statement in a subroutine. When the program encounters the RETURN statement, control transfers back to the statement following the GOSUB.

A subroutine can call another subroutine. This is called "nesting" (see Example 2). Programs may be written to transfer control from one statement to another in the same subroutine, or to a statement in a different subroutine. When a RETURN is encountered, control returns to the statement following the last GOSUB that was executed. Subroutines may not be nested more than ten levels deep (Example 2 shows two levels of nesting).

The following examples illustrate the GOSUB statement.

Example 1:

The following program

```
100 LET A8="HI"
110 LET B$="THERE"
120 LET C$=" "
130 LET V$=A$
140 GOSUB 1000
150 LET V$=A$&C$&B$
160 GOSUB 1000
170 STOP
1000 REM PRINT V$
1010 PRINT V$
1020 RETURN
9999 END
```

will display:

```
HI
HI THERE
```

Example 2:

The following program, showing subroutine nesting,

```
100 FOR I=1 TO 3
110 LET V=I
120 GOSUB 1000
130 LET X=2*I
140 GOSUB 500
150 NEXT I
200 STOP

500 REM CALCULATE
510 LET V=X+2
520 GOSUB 1000
530 RETURN

1000 REM PRINT VALUE
1010 PRINT "THE VALUE IS";
1020 PRINT V
1030 RETURN
9999 END
```

will display:

```
THE VALUE IS 1
THE VALUE IS 4
THE VALUE IS 2
THE VALUE IS 6
THE VALUE IS 3
THE VALUE IS 8
```

6.7.6.4 Return from Subroutine (RETURN Statement) – The RETURN statement is used to redirect control from a subroutine.

RETURN Statement Form

Line Number	**Statement**	**Parameters**
xxx	RETURN	(none)

RETURN is always used with the GOSUB statement.

When the RETURN is encountered, it causes control to transfer to the statement following the last GOSUB executed.

6.7.7 Program Termination Statements (END, STOP)

BASIC is equipped with two statements, the END and STOP statements, that can be used to terminate program execution.

6.7.7.1 END Statement Format – The END statement terminates execution of the program. It informs the BASIC compiler that it has reached the last line of the program.

END Statement Form

Line Number	**Statement**	**Parameters**
xxx	END	None

NOTE

Only one END statement may appear in a program and it must be the last statement in the program.

6.7.7.2 STOP Statement Format – The STOP statement is used to terminate the execution of a program.

STOP Statement Form

Line Number	**Statement**	**Parameters**
xxx	STOP	None

Note that there may be several STOP statements in a program.

6.7.8 The INPUT Statement

The INPUT statement permits the operator to specify data during execution of a program.

INPUT Statement Form

Line Number	**Statement**	**Parameters**
xxx	INPUT	Variable List

where variables can be a single variable or a list of variables.

The INPUT Statement will cause the program to pause during execution, display a question mark, and wait for you to enter a value and press the RETURN key. If there are several variables involved, the program will expect you to type in a value corresponding to each variable. If you press the RETURN without having done this, the system will display another question mark and wait for the rest of the data. When the values have all been typed in, the program will continue with the variable names now equivalent to the values typed in. The first variable will equal the first entry, the second will equal the second entry, etc. (See Example 1).

The following values are recognized as acceptable when inputting numeric data:

+ or - sign
digits 0 through 9
the letter E
leading spaces (ignored)
(first decimal point)

All other characters are treated as delimiters for separating numeric data. That is, when the system encounters a character other than those specified, it will consider that it has come to the end of the entry relating to the variable it is currently processing and will apply any characters typed in after that to the following variable, if any (see Example 1).

When inputting numeric data, two delimiters read in succession imply that the data between delimiters is 0 (see Example 2).

In response to an INPUT statement, you can provide more values than are requested by the INPUT statement. The remaining or unused values are saved for subsequent use by the next INPUT statement. The question mark is not displayed until the program is out of data.

When inputting string data all characters are recognized as part of the string. Quotation marks are not typed in unless they are deliberately meant to be part of the string.

Each string requested by an INPUT statement must be terminated by a carriage return which acts as the data delimiter. This is necessary since all characters except for the carriage return are recognized as part of the data string.

String variables are assumed to be eight characters long unless otherwise described in a DIM statement.

The following examples illustrate the use of the INPUT statement.

Example 1:

If, in response to this statement:

```
100 INPUT A,B,C,D,E
```

you type

```
?-2,3.7A4E3 9<+1
```

the variables will have the following values:

A:-2
B:3.7
C:4000(4E3=4X10^3=4000)
D:9
E:1(numbers are assumed to be positive unless they are specified to be negative).

The delimiters in the above line are the comma, the letter "A", the space after the number 3, the left-angle bracket (<), and the RETURN.

Example 2:

If, in response to the same statement, you type

```
?-2, 3.7, 4E3,,1
```

The results will be identical except that variable "D" will have the value 0.

Example 3:

If, in response to the statement:

```
50 INPUT R$
```

you type

```
?  "A,C>=+7
```

the string variable R$ will have the value:

"A,C>=+7

Example 4:

If, in response to:

```
40 LET A=5
50 INPUT B(A)
```

you type

```
? 7
```

B (5) will now have the value of 7.

6.7.9 The REMark Statement

The REM statement is a nonexecutable statement used to insert comments into the source program.

REMARK Statement Form

Line Number	Statement	Parameters
xxx	REM	comments

6.7.10 Ancillary Statements (DIMension, RESTORE, DEFine, RANDOMIZE and CHAIN)

BASIC has five additional statements that are used as helping statements and fall in no particular category. They are described in the following paragraphs.

6.7.10.1 DIMension Statement Format – The DIMension statement is used to describe subscripted variables and to define the length of strings.

DIMension Statement Form

Line Number	**Statement**	**Parameters**
xxx	DIM	v(n[,m])

where v is the name of the subscripted variable.

If the variable name (v) is numeric, and

1. m is omitted, then n+1 is the number of elements in an array or vector (see Example 1).
2. m is specified, then n+1 is the number of rows in a two-dimensional array (see Example 2).
3. if v is numeric, then m+1 is the number of columns in a two-dimensional array (see Example 2);

If the variable name (v) is alphanumeric, and

1. m is omitted, then n is the length of the string. This describes a single string, not a list, and cannot exceed 80 (see Example 3).
2. m is specified, then n+1 is the number of strings in the list. This is a one dimensional array (see Example 4).
3. if v is alphanumeric, then m is the length of each string in a one-dimensional list. It cannot exceed 80 (see Example 4).

The parameters n and m must be integer constants. They are limited in size only by the amount of available memory.

If a numeric variable is used in the program with a subscript but is not defined in a DIM statement, BASIC assigns it an array size of ten.

BASIC assumes a maximum string length of 8 characters unless the variable appears in a DIM statement.

Two-dimensional string variables are not permitted.

When the variable is used in other statements, it is not permitted to have subscripts whose values are higher than those in the DIM statement.

The first element of every array is automatically assumed to have a subscript of zero. Therefore, the number of boxes in a one dimensional array is n+1. The number of boxes in a two-dimensional array is (n+1)*(m+1). The first element in an array is v(0) or v(0,0) (see Example 2). However, the "zero" elements can be disregarded in programming unless the user wishes to conserve memory.

More than one array can be defined in a single DIM statement (see Example 7).

In general, wherever you can use a single variable name in a statement, you can use a subscripted one. Exceptions are noted in descriptions of the individual statements (such as the index of the FOR statement).

The following examples illustrate the use of DIMension statement.

Example 1:

The following statement

```
10 DIM A(5)
```

describes six numeric elements as follows:

A(0)	A(1)	A(2)	A(3)	A(4)	A(5)

To store a 5 in A(3) and then display it, type

```
20 LET A(3)=5
30 PRINT A(3)
```

Example 2:

The following statement

```
10 DIM A(3,5)
```

describes 24 numeric elements (4X6=24) as follows:

SIX COLUMNS

	A(0,0)	A(0,1)	A(0,2)	A(0,3)	A(0,4)	A(0,5)
FOUR	A(1,0)	A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
ROWS	A(2,0)	A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
	A(3,0)	A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)

Example 3:

The following statement

```
10 DIM C$(12)
```

describes one string, 12 characters long:

C$

Example 4:

The following statement

```
10 DIM D$(3,20)
```

describes 4 strings, each 20 characters long:

D$(0)   D$(1)   D$(2)   D$(3)

Example 5:

The following program will fill the array in Example 4 from a DATA list:

```
10 DIM D$(3,20)
20 FOR Y=0 TO 3
30 READ D$(Y)
40 NEXT Y
50 FOR Z=0 TO 3
60 PRINT D$(Z)
70 NEXT Z
80 DATA "ZERO","ONE","TWO","THREE"
99 END
```

and display each element:

```
ZERO
ONE
TWO
THREE
```

Example 6:

After these statements:

```
10 DIM B(3,5)
20 FOR G=1 TO 3
30 LET B(G,0)=G
40 NEXT G
50 FOR H=2 TO 5
60 LET B(0,H)=H
70 NEXT H
99 END
```

The area diagrammed in Example 2 would appear as follows:

			B(0,2)	B(0,3)	B(0,4)	B(0,5)
			2	3	4	5
B(1,0)	1					
B(2,0)	2					
B(3,0)	3					

Example 7:

The following statement dimensions both the one-dimensional array A and the two-dimensional array B:

```
10 DIM A(20), B(4,7)
```

6.7.10.2 RESTORE Statement – The RESTORE statement allows the program to go back to the beginning of a DATA list (paragraph 6.7.3.1) after using the list in a READ statement (paragraph 6.7.3.2).

RESTORE Statement Form

Line Number	**Statement**	**Parameters**
xxx	RESTORE	None

If it is desired to use the same data more than once in a program, RESTORE makes it possible to recycle through the DATA list beginning with the first value in the first DATA statement.

The RESTORE statement may be used in programs where DATA statements convey numeric or string data to READ statements.

The same variable names may be used the second time through the data since the values are being read as though for the first time.

The following example illustrates the use of the RESTORE statement.

The following lines

```
10 READ A,B,C,D
20 PRINT A;B;C;D
30 RESTORE
40 READ E,F,G,H
50 PRINT E;F;G;H
60 DATA 1,2,3,4
99 END
```

will cause this display:

```
1 2 3 4
1 2 3 4
```

6.7.10.3 DEFine Statement – This statement allows you to add functions to a program.

DEFine Statement Form

Line Number	**Statement**	**Parameters**
xxx	DEF	FNa(x)=expression

where

a is the capital letter used for identifying the function;

x is a dummy variable and must be the same on both sides of the equal (=) sign; and

expression defines the function by indicating the calculation process (function) involved.

There must be a DEF statement for each function used in the program.

The DEF statement must appear before the first use of the function it defines.

If there is more than one variable involved in the function, BASIC will identify them by their position.

Up to 14 different arguments may be used.

Up to 26 FN functions may be defined in a single program (FNA, FNB . . .FNZ).

The following examples illustrate the use of the DEFine statement.

Example 1:

The statement

```
    10 DEF FND(S)=S^2
```

will cause the later statement

```
    50 LET R=FND(4)
```

to be evaluated as R=16. BASIC locates the DEF statement for the function FND, substitutes the 4 for the variable (S) in the expression (S^2) and calculates the value of FND(4) to be 4^2=16.

NOTE
A variable that is used as a dummy argument in a DEF FNa statement can also be used elsewhere in the program.

Example 2:

This program:

```
    10 DEF FNH(N,P)=2*P+N
    20 LET X=4\LET Y=5
    30 PRINT FNH(X,Y)
    40 END
```

will display:

```
    14
```

BASIC takes the first value in the function (4) as "N", because "N" appears first in the DEF statement. It takes the second value (5) as "P", because "P" is in the second position.

    DEF FNH (N,P) = 2*P+N

first position   second position

    PRINT FNH(X,Y)

6.7.10.4 RANDOMIZE Statement – The RANDOMIZE statement is used with the RND function to generate a different set of numbers each time the program is run.

RANDOMIZE Statement Form

Line Number	Statement	Parameters
xxx	RANDOMIZE	None

6.7.10.5 CHAIN Statement – The CHAIN statement allows one program to execute another program. It can be used to divide large programs into a number of smaller programs that are to be written and stored separately.

CHAIN Statement Form

Line Number	Statement	Parameters
xxx	CHAIN	"dev:file.ex"

where "dev:file.ex" is the device and the file name of the program to be executed.

When BASIC encounters a CHAIN statement in a program, it stops execution of that program, retrieves the program named in the CHAIN statement from the specified device and file, compiles the CHAINed program (if necessary) and begins execution of that program.

If the program was started in the editor, when execution of all programs in the chain is complete, the workspace will contain the original program.

All output files that are opened in the original program must be closed before the CHAIN statement is encountered.

A BASIC language program may only CHAIN to another BASIC language program, and the program it CHAINs to may not have an extension of ".SV" (see the following example).

A compiled program may also execute CHAIN statements, but it can only CHAIN to other compiled programs which have ".SV" extensions.

In the following example, the program "SECOND" will CHAIN to the program "FIRST".

```
NEW FIRST

READY
10 PRINT "TARGET"
20 END
SAVE SYS:FIRST                    Enter FIRST and store it.

READY
NEW SECOND

READY
10 PRINT "ORIGINAL"
20 CHAIN "SYS:FIRST.BA"
30 END
SAVE SYS:SECOND                   Enter SECOND and store it.

READY
RUNNH                             Execute the programs.
ORIGINAL                          Displayed by SECOND.
TARGET                            Displayed by FIRST.
```

6.7.11 File Handling Statements
The file capability provided by OS/78 BASIC allows writing to or reading from the peripheral devices of the system.

6.7.11.1 FILE# Statement – The FILE# statement defines and opens a file.

FILE# Statement Form

Line Number	Statement	Parameters
xxx	FILE	t # n: "dev:file.ex"

where

t	must be one of the following: (blank) – for an input string file V – for an output string file N – for an input numeric file VN – for an output numeric file;
n	is the number you are assigning to the file (It must be 1, 2, 3, or 4 and can be a numeric variable); and
"dev:file.ex"	is the standard device, file name and extension; It must be either a string variable or the string itself in quotation marks.

A file must be opened before it can be used. The only exception is the terminal, which is always available for use.

The n in this statement is the number you must use in all FILE# statements that refer to the file. The terminal is always FILE#0.

The following examples illustrate the use of the FILE# statement.

Example 1:

The following statement describes file number 1 to be the string file HPRDAT.AS on RXA1 and opens it for output.

```
10 FILEV#1:"RXA1:HPRDAT.AS"
```

Example 2:

The following statement describes file number 2 to be the numeric file DATA.NU on RXA1 and opens it for output.

```
10 FILEVN#2:"RXA1:DATA.NU"
```

Example 3:

The following statement describes file number 3 to be the string file TEST.AB on RXA1 and opens it for input.

```
10 FILE#3:"RXA1:TEST.AB"
```

Example 4:

The following statement describes file number 4 to be the numeric file FILA.CD on RXA1 and opens it for input.

```
10 FILEN#4:"RXA1:FILA.CD"
```

6.7.11.2 PRINT# Statement – The PRINT# statement writes data into files.

PRINT# Statement Form

Line Number	**Statement**	**Parameters**
xxx	PRINT#	n:expressions

where

n	is the file number (It may be a numeric variable); and
expressions	depends on file type, numeric or string as discussed below.

As long as PRINT# is used for only numbers or numeric variables separated by commas or semicolons (or RETURN at the end of a PRINT# line), BASIC converts commas to spaces and does not write the carriage return and line feed to numeric files. The only thing written out will be a "list" of numbers separated by spaces. Each time INPUT# is used, it will read another number from the list (see Example 1).

When dealing with string files, symbols such as RETURNs, semicolons, and so forth, are used in the PRINT# statement in the same way they are used in the PRINT statement. The PRINT# statement works exactly the same way that the PRINT statement does, except that the line goes to the file designated instead of to the terminal.

The important difference here is that string files involve lines, while numeric files involve individual numbers. That is, each INPUT# will read a line (see Example 2).

If PRINT# is used for numerics to a string file, BASIC will convert them to strings. If an attempt is then made to INPUT# them into numeric variables, BASIC will convert them back to numerics. However, the RETURN and line feed at the end of a line will be converted to zeros. This can be dealt with by adding two extra variables to the INPUT# statement.

The following examples illustrate the use of the PRINT# statement.

Example 1:

The following lines

```
10 FILEVN#1:"SYS:TST.XX"
20 PRINT#1:1,2
30 PRINT#1:3,4,
40 PRINT#1:5,6
50 CLOSE#1
60 FILEN#1:"SYS:TST.XX"
70 FOR X=1 TO 6
80 INPUT#1:Z
90 PRINT Z
100 NEXT X
199 END
```

will display

```
1
2
3
4
5
6
```

Example 2:

The following lines

```
10 PRINT "A", "B"
20 PRINT "C"; "D";
30 PRINT "E"
40 END
```

will display

```
A         B
CDE
```

The same display will also be caused by

```
5 DIM J$(30)
10 FILEV#2:"RXA1:PROG.XX"
20 PRINT #2:"A", "B"
30 PRINT #2:"C"; "D";
40 PRINT #2:"E"
50 CLOSE#2
60 FILE#2:"RXA1:PROG.XX"
70 INPUT#2:J$
80 PRINT J$
90 INPUT#2:J$
100 PRINT J$
199 END
```

6.7.11.3 INPUT# Statement – The INPUT# statement reads data from a file.

INPUT# Statement Form

Line Number	Statement	Parameters
xxx	INPUT#	n:variables

where

n	is the file number of the file being read (it may be a variable); and
variables	is the list of variables into which data will be read. Each variable is separated by a comma.

Normally, data from numeric files is read into numeric variables and data from string files is read into string variables.

It is possible however to write numerics into a string file and then read them into either numeric or string variables, depending on how it is desired to use them. If numbers are read from a string file into string variables, they will be in string form and subject to the same rules as other strings.

Numbers read from a string file into numeric variables will be converted to numerics. Line feeds are ignored in this case.

The following examples illustrate the use of the INPUT# statement.

Example 1:

To read two strings from RXA1:FIL.DA, enter

```
10 FILE#1:"RXA1:FIL.DA"
20 INPUT#1:A$,B$
```

A$ will contain the first string, B$ the second string.

Example 2:

To read five numbers from RXA1:TST.XX, enter

```
10 FILEN#3:"RXA1:TST.XX"
20 INPUT#3:A,B,C,D,E
```

Example 3:

The following program writes numerics to a string file and reads them back as numerics:

```
10 FILEV#1:"SYS.FILA.ZZ"
20 FOR I=1 TO 5
30 PRINT#1:I
40 NEXT I
50 CLOSE#1
60 FILE#1:"SYS:FILA.ZZ"
70 FOR I=1 TO 5
80 INPUT#1:J,C,L
90 PRINT J
100 NEXT I
110 END
```

It will display:

```
1
2
3
4
5
```

6.7.11.4 RESTORE# Statement – The RESTORE statement resets the file data printer back to the beginning of the file, that is, RESTORE effectively performs a file close followed immediately by a file open so that the first data element in the file can be reread.

RESTORE# Statement Form

Line Number	Statement	Parameters
xxx	RESTORE#	n

where

n is the number of the file to be reset. It may be a number or a numeric variable.

For example, if RXA1:FILB.LM is a numeric file containing the numbers 1 through 9, the instructions

```
100 FILEN#3:"RXA1:FILB.LM"
110 FOR I=1 TO 3
120 INPUT #3:Z
130 PRINT Z
140 NEXT I
150 RESTORE#3
160 INPUT#3:Z
170 PRINT Z
199 END
```

will display:

```
1
2
3
1
```

NOTE
If n is zero, the DATA list in the program is reset.

6.7.11.5 CLOSE# Statement – The CLOSE# statement finishes the processing of a file and allows it number to be assigned to another file in a FILE# statement.

FILE CLOSE# Statement Form

Line Number	Statement	Parameters
xxx	CLOSE#	n

where

n is the number of the file to be closed. It may be a variable.

For example, in the following program

```
50 FILEV #1:"SYS:TEST.XX"
60 PRINT #1:"A","B","C","D"
70 CLOSE #1
80 FILE #1:"RXA1:FILD.DA"
90 INPUT #1:J$
99 END
```

The CLOSE# statement at line 70 finished the processing of file SYS:TEST.XX and allowed its number, 1, to be assigned to RXA1:FILD.DA in line 80.

NOTE
All output files must be CLOSED before any of the following is executed.

CHAIN	END
STOP	CTRL/C

Failure to do so results in loss of file.

6.7.11.6 IF END# Statement – The IF END# statement determines if the End of File (EOF) marker has been read from a file, and if so, branches to the line specified.

IF END# Statement Form

Line Number	Statement	Parameters
xxx	IF END#	n THEN m

where:

n	is the file number of the file in question (it may be a variable); and
m	is the line number to which control passes when the end of the file is detected.

This statement works only for string files.

The IF END# statement should come immediately after the PRINT# or INPUT# statement for that file. If, as a result of the IF END# statement, control passes to line m, it means that the last PRINT# or INPUT# was not successful. That is, nothing was actually read from or written to the file as a result of the last INPUT# or PRINT# statement.

For Example,

The following lines

```
 10 FILEV#1:"SYS:PROGA.BB"
 20 PRINT#1:"A"
 30 PRINT#1:"B"
 40 CLOSE#1
 50 FILE#1:"SYS:PROGA.BB"
 60 INPUT#1:A$
 70 IF END#1 THEN 100
 80 PRINT A$
 90 GOTO 60
100 PRINT "END OF FILE"
110 CLOSE#1
199 END
```

will display

```
A
B
END OF FILE
```

6.8 BASIC FUNCTIONS

BASIC functions are standard subroutines incorporated into the BASIC Run Time System (BRTS) to aid computations and text handling.

Function calls consist of a three letter (all capitals) name followed by an argument in parentheses. The argument may be a number, variable, expression, or another function. Generally, functions may be used anywhere a number or variable is legal in a mathematical expression.

Most functions compute a value based on the value of the argument or arguments involved. They are said to return this value. For example, SQR(A) "returns" the square root of A.

Functions may return either strings or numbers. Functions that return strings have names ending in a dollar sign (STR$, SEG$), while functions returning numbers have names that do not end in a dollar sign (SGN, VAL).

6.8.1 Arithmetic Functions (ABS, INT, EXP, RND, SGN, SQR)

6.8.1.1 BASIC ABS Function – The ABS function returns the absolute value of an expression.

Format

    ABS(X)

where

X	is a number, numeric variable, or numeric expression

6.8.1.2 BASIC EXP Function – The EXP function calculates the value of e raised to the X power, where e is equal to 2.71828. That is, EXP(X) is equivalent to 2.71828 X.

Format

    EXP(X)

where

X	is a number, numeric variable, expression, or another function.

6.8.1.3 BASIC INT Function – The INT function returns the value of the largest integer not greater than the argument.

Format

    INT(X)

where

X	is a number, numeric variable, expression, or another function.

NOTE
This function can be used to round numbers to the nearest integer by specifying INT(X+.5).

For example, the function INT(34.67) has the value 34; the functions INT(34.67+.5) and INT(34.37+.5) have these values of 35 and 34, respectively; and these functions INT(-23) and INT(-14.39) have these values of -23 and -15, respectively.

6.8.1.4 BASIC RND Function – The RND function produces random numbers between (but not including 0 and 1.

Format

    RND(X)

where

X	is a dummy variable in this function.

Every time this function is encountered in a statement, it will produce a different set of decimal numbers. However, the program is RUN again, the same set of numbers will be produced (see Example 1).

If this repetition is undesirable, it can be changed with the RANDOMIZE statement.

For example, the following program is run twice with identical results:

```
10 FOR A = 1 TO 5
20 PRINT RND(X)
30 NEXT A
40 END

RUNNH
0. 361572
0. 332764
0. 633057
0. 350342
0. 670166

READY
RUNNH
0. 361572
0. 332764
0. 633057
0. 350342
0. 670166

READY
```

6.8.1.5 BASIC SGN Function – The SGN function creates a value based on the sign of the argument.

Format

SGN(X)

where

X is a number, numeric variable, numeric expression, or another function.

NOTE
The value of the SGN function will be 1 if the argument is any positive number, 0 if the argument is zero, and -1 if the argument is negative.

6.8.1.6 BASIC SQR Function – The SQR function computes the positive square root of an expression.

Format

SQR(X)

where

X is a number, numeric variable, numeric expression, or another function.

NOTE

If the argument is negative, the absolute value of the argument is used.

6.8.2 Trigonometric Functions (ATN, COS, LOG, SIN)

6.8.2.1 BASIC ATN Function – The ATN function calculates the angle (in radians) whose tangent is given as the argument of the function.

Format

    ATN(X)

where

X	is a number, numeric variable, expression, or another function, representing the tangent of an angle.

6.8.2.2 BASIC COS Function – The COS function is used to calculate the cosine of an angle specified in radians.

Format

    COS(X)

where

X	is a number, numeric variable, expression, or another function, representing the size of an angle in radians.

6.8.2.3 BASIC LOG Function – The LOG function calculates the natural logarithm of X (to the base e).

Format

    LOG(X)

where

X	is a number, numeric variable, expression, or another function.

6.8.2.4 BASIC SIN Function – The SIN function is used to calculate the sine of an angle specified in radians.

Format

    SIN(X)

where

X	is a number, numeric variable, expression or another function, representing the size of an angle in radians.

6.8.3 String Handling Functions (ASC, CHP$, DAT$, LEN, POS, SEG$, STR$, TAB, VAL)

6.8.3.1 BASIC ASC Function – The ASC function converts a one character string to its code number (see CHR$).

Format

ASC(X)

where

X   is a one character string.

To find what will be returned for any character, look for the character in the "CHARACTER" column of Table 6-1. The number to the left of it is the decimal equivalent.

Table 6-1 Decimal/Character Conversions

Decimal	Character	Decimal	Character
0	@	32	(space)
1	A	33	!
2	B	34	”
3	C	35	#
4	D	36	$
5	E	37	%
6	F	38	&
7	G	39	’
8	H	40	(
9	I	41	)
10	J	42	*
11	K	43	+
12	L	44	,
13	M	45	-
14	N	46	.
15	O	47	/
16	P	48	0
17	Q	49	1
18	R	50	2
19	S	51	3
20	T	52	4
21	U	53	5
22	V	54	6
23	W	55	7
24	X	56	8
25	Y	57	9
26	Z	58	:
27	[	59	;
28	\	60	<
29	]	61	=
30	^	62	>
31	_	63	?

For example,

The following program

```
10 LET A$="*"
20 PRINT ASC("P"), ASC(A$),ASC("9")
30 END
```

will display

```
16        42        57
```

6.8.3.2 BASIC CHR$ Function – The CHR$ function converts a code number (modulo 64) to its equivalent character in the 64 character set.

Format

CHR$(X)

where

X is a number, numeric expression, or numeric variable ($0 < X < 63$).

To find what will be returned for any number, look for the number in the "DECIMAL" column of the preceding conversion table. The equivalent character will be to the right of the number.

For example, the following line

```
10 PRINT CHR$(1),CHR$(40)
```

will display:

```
A   (
```

6.8.3.3 BASIC DAT$ Function – The DAT$ function returns the current system date.

Format

DAT$(X)

where

X is a dummy variable in this function.

The date is returned as an eight-character string of the form:

MM/DD/YY

If the date has not been specified with the Monitor DATE command, no characters will be returned.

For example, the following lines

```
10 LET D$ = DAT$(X)
20 PRINT D$
```

will display:

```
07/20/77
```

if that date was entered with the Monitor DATE command.

6.8.3.4 BASIC LEN Function – The LEN function returns the number of characters in a string.

Format

LEN(X$)

where

X$	is a string or string variable. It may be several concentrated strings and/or variables.

6.8.3.5 BASIC POS Function – The POS function returns the location of a specified group of characters in a string.

Format

POS(X$, Y$, Z)

where

X$	is the string to be searched (it may be a string variable or string constant);
Y$	is the series of characters you are searching for (it may be a string variable or a string constant); and
Z	is the position in the string at which you want to begin the search.

This function searches X$ for the first occurrence of Y$. The search begins with the Zth character in X$.

Z may not be less than zero or greater than the length of string X$.

If Y$ contains no characters, the function returns a one.

If X$ contains no characters, it returns a zero.

If Y$ is not found, it returns a zero.

For example, the following lines

```
10 DIM B2$(12)
20 B2$ = "ABCDEFGHIDEF"
30 PRINT POS (B2$, "DEF", 7)
```

will display:

```
10
```

6.8.3.6 BASIC SEG$ Function – The SEG$ function returns the sequence of characters between two positions in a string.

Format

SEG$(X$,Y,Z)

where

X$	is the string containing the characters to be returned (it may be a string variable or a string constant);
Y	is the position of the first character to be returned; and
Z	is the position of the last character to be returned.

If Y is less than 1, it is set to 1.

If Y is greater than the length of X$, no characters are returned.

If Z is less than 1, no characters are returned.

If Z is greater than the length of X$, it is set equal to the length of X$.

If Z is smaller then Y, no characters are returned.

For example, the following lines

```
10 DIM B2$(12)
20 B2$ = "ABCDEFGHIDEF"
30 PRINT SEG$(B2$,3,5)
```

will display:

```
CDE
```

6.8.3.7 BASIC STR$ Function – The STR$ function converts numbers to strings.

Format

STR$(X)

where

X	is a numeric expression

NOTE
The string that is returned is in the form in which numbers are output in BASIC without leading or trailing blanks.

6.8.3.8 BASIC TAB Function – The TAB function allows you to position characters anywhere on the terminal line.

Format

TAB(X)

where

X is the position (from 1 to 80) in which the next character will be displayed.

This function may only be used in a PRINT or PRINT# statement.

Positions on the line are considered by BASIC to be numbered from 1 to 80 across the screen from left to right.

Each time the TAB function is used, positions are counted from the beginning of the line, not from the current position of the cursor.

If X is less than the current position of the cursor, the display starts at the current position.

If X is greater than 80, the display will begin at the first position of the next line.

In order to keep track of the cursor, BASIC maintains a "column count", which represents the position of the cursor at any given time. As the cursor moves across the screen, BASIC adds to the column count. When the cursor returns to the first position, the column count is reset to 0. The activity of the TAB function is based on this count.

There are circumstances in which the column count does not coincide with the position of the cursor. For example, the PNT(07) function will add 1 to the count without moving the cursor. Also, the PNT(13) function will return the cursor to the first position on the screen without setting the column count to zero. The user, therefore, must take this into account when using the TAB function after PNT functions. The column count will be corrected the next time there is a "normal" return to column one.

For examples, the following lines

```
60 LET B=5
70 PRINT "A";TAB(B);"C"
```

will cause this display:

```
A   C
```

6.8.3.9 BASIC VAL Function – The VAL function converts a string to numeric data.

Format

VAL(X$)

where

X$ is a string constant or string variable made up of those values that BASIC recognizes as acceptable when inputting numeric data:

+ or - sign
digits 0 through 9
the letter E
leading spaces (ignored)
. (first decimal point)

NOTE

A string, even though it is composed of digits, is not numeric data. It cannot be used in calculations or as the argument of a mathematical function (SQR, ABS, EXP, and so forth), without first being converted by the VAL function.

6.8.4 Display Console Control Function (PNT)

BASIC PNT Function – The PNT function is used to perform special actions on the terminal, such as sounding the buzzer, erasing the screen, and moving the cursor.

Format

PNT(X)

where

X	is the value of the character to be output.

Special actions that can be performed by the PNT function are as follows:

PNT(07)	sounds buzzer
PNT(08)	moves cursor one space to left
PNT(09)	moves to next tab stop (Tab stops are set every 8 spaces.)
PNT(10)	moves cursor down one line and scrolls if required
PNT(13)	moves cursor to left margin of current line
PNT(27);"A"	moves cursor up one line
PNT(27);"C"	moves cursor right one position
PNT(27);"H"	moves cursor to upper lefthand corner of screen ("home" position)
PNT(27);"J"	erases from cursor position to end of screen
PNT(27);"K"	erases line from cursor to right margin
PNT(27);"["	stops display of any new lines when screen is full. Each time the operator presses the SCROLL key, another line will be displayed. If he presses SHIFT and SCROLL, twelve new lines will be displayed. This is called Hold Screen Mode.
PNT(27);CHR$(28)	turns off Hold Screen Mode.

NOTE

The PNT function may only be used in a PRINT or PRINT# Statement.

6.8.5 Trace Function

BASIC TRC Function – The TRC function causes BASIC to print the line number of each statement in the program as it is executed.

Format

v = TRC(X)

where

v can be any letter (it is a dummy argument and has no purpose except to occupy that position on the line); and

x is 1 to turn the function on and 0 to turn it off.

This function is used to follow the progress of a program and help in tracking down errors.

When BASIC encounters TRC(1) in a program, it displays the line number of each line in the program as it is executed. The line numbers are displayed between percent signs.

When TRC(0) is encountered, the function is turned off and normal program operation resumes.

Certain types of statements are not recorded by TRC: namely, DATA, DEF, DIM, END, GOTO, NEXT, RANDOMIZE, REM, and STOP.

For example, the following program

```
60 T=TRC(1)
70 GOSUB 90
80 GOTO 140
90 PRINT "IN OUTER SUB"
100 GOSUB 120
110 RETURN
120 PRINT "IN INNER SUB"
130 RETURN
140 T= TRC(0)
150 END
```

will display:

```
% 70 %
% 90 %
IN OUTER SUB
% 100 %
% 120 %
IN INNER SUB
% 130 %
% 110 %
% 140 %
```

6.9 SUMMARY OF BASIC EDITOR COMMANDS

Command	Function
BYE	Exits from the editor and returns control to the monitor
LIst	Displays the program statements in the workspace with a header
LISTNH	Displays the program statements in the work space, without a header
NAme	Renames the program in the workspace
NEW	Clears the workspace and tells the editor the name of the program the user is about to type
OLd	Clears the workspace, finds a program on the disk, and puts in into the workspace
RUn	Executes the program in the workspace, after displaying a header
RUNNH	Executes the program in the workspace, without displaying a header
SAve	Puts the program in the workspace on a disk
SCratch	Erases all statements from the workspace

6.10 SUMMARY OF BASIC STATEMENTS

Statement	Function
CHAIN	Executes another program Example: 40 CHAIN "SYS:PROG.BA"
CLOSE#	Closes a file Example: 100 CLOSE#1
DATA	Sets up a list of values to be used by the READ statement Example: 240 DATA "FIRST",2,3
DEF	Defines functions Example: 10 DEF FND(S)=S+5
DIM	Describes a string and/or any subscripted variables Example: 50 DIM B(3,5),D$(3,72)
END	Terminates program compilation and execution Example: 100 END
FILE#	Defines and opens a file Example: 20 FILEVN#2:"RXA1:DATA.NV"

Statement	Function
FOR	Describes program loops (used with NEXT) Example: 60 FOR X=1 TO 10 STEP 2
GOSUB	Transfers control to a subroutine (used with RETURN) Example: 50 GOSUB 100
GOTO	Transfers control to another statement Example: 100 GOTO 50
IF	Tests the relationship between two variables, numbers, or expressions Example: 20 IF A=0 THEN 50
IF END#	Tests for the end of a string file Example: 60 IF END#3 THEN 100
INPUT	Accepts data from the terminal Example: 80 INPUT A,B,C
INPUT#	Reads data from a file Example: 50 INPUT#1:A$
LET	Assigns a value to a variable Example: 90 LET A$="XYZ"
NEXT	Indicates the end of a program loop (used with FOR) Example: 140 NEXT I
PRINT	Displays data on the screen Example: 200 PRINT A,"X";6
PRINT#	Writes data to a file Example: 180 PRINT#1:J
RANDOMIZE	Causes the RND function to produce a different set of numbers each time the program in run Example: 10 RANDOMIZE
READ	Sets variables equal to the values in DATA statements Example: 50 READ A$,B
REM	Inserts comments into the program Example: 30 REM COMPUTE EARNINGS
RESTORE	Sets program READ statements back to the beginning of the DATA list Example: 85 RESTORE

Statement	Function
RESTORE#	Resets a file pointer back to the beginning of that file Example: 130 RESTORE#3
RETURN	Returns control from a subroutine (used with GOSUB) Example: 115 RETURN
STOP	Terminates program execution Example: 40 STOP

6.11 SUMMARY OF BASIC FUNCTIONS

Command	Function
ABS(X)	Returns the absolute value of an expression Example: 10 LET X=ABS(-66) will assign X a value of 66
ASC(X$)	Converts a one character string to its code number Example: 20 PRINT ASC("B") will display 2
ATN(X)	Calculates the angle (in radians) whose tangent is given as the argument Example: 30 LET X=ATN(.57735) will assign X a value of 0.523598
CHR$(X)	Converts a code number to its equivalent character Example: 40 PRINT CHR$(1) will display A
COS(X)	Returns the cosine of an angle specified in radians Example: 50 LET Y=COS(45*3.14159)/180 will assign Y a value of 0.707108
DAT$(X)	Returns the current system date Example: 60 PRINT DAT$(X) will display the system date, such as 07/20/77
EXP(X)	Calculates the value of e raised to a power, where e is equal to 2.71828 Example: 30 IF Y>EXP(1.5) GOTO 70 will go to line 70 if Y is greater than 4.48169
INT(X)	Returns the value of the nearest integer not greater than the argument Example: 60 LET X=INT(34.67) will assign X the value 34

Command	**Function**
LEN(X$)	Returns the number of characters in a string Example: 10 PRINT LEN ("DOG") will display 3
LOG(X)	Calculates the natural logarithm of the argument Example: 10 PRINT LOG(959) will display 6.86589
PNT(X)	Outputs non-printing characters for terminal control Example: 50 PRINT PNT(13) will move the cursor to the left margin of the current line
POS(X$,Y$,Z)	Returns the location of a specified group of characters (Y$) in a string (X$) starting at a character position (Z) Example: 60 LET V= POS("ABCDBC", "BC",4) will assign V a value of 5
RND(X)	Returns a random number between (but not including) 0 and 1 Example: 70 PRINT RND(X) will display a decimal number, such as 0.361572
SEG$(X$,Y,Z)	Returns the sequence of characters in a string (X$) between two positions in the string (X,Y) Example: 30 LET R$=SEG$("ABCDEF",2,4) will assign R$ a value of BCD
SGN(X)	Returns 1 if the argument is positive, 0 if it is zero, and -1 if it is negative Example: 200 PRINT 5*SGN(-6) will display -5
SIN(X)	Returns the sine of an angle specified in radians Example: 30 LET B=SIN(30*3.14159/180) will assign B a value of 0.5
SQR(X)	Returns the positive square root of an expression Example: 40 PRINT SQR(16) will display 4
STR$(X)	Converts a number into a string Example: 120 PRINT STR$(1.76111124) will display the string 1.76111

Command	**Function**
TAB(X)	Positions characters on a line Example: 70 PRINT "A";TAB(5);"B" will display A B
TRC(1)	Causes BASIC to display the line number of each statement in the program as it is executed Example: 10 V=TRC(1) will display the line number of each statement executed until a TRC (0) is encountered
VAL(X$)	Converts a string to a number Example: 90 PRINT VAL("2.46111")*2 will display 4.92222

6.12 BASIC ERROR MESSAGES

6.12.1 Compiler Error Messages

The following error messages are generated by the BASIC compiler:

CH	ERROR IN CHAIN STATEMENT	NM	MISSING LINE NUMBER
DE	ERROR IN DEF STATEMENT	OF	OUTPUT FILE ERROR
DI	ERROR IN DIM STATEMENT	PD	PUSHDOWN STACK OVERFLOW
FN	ERROR IN FILE NUMBER OR NAME	QS	STRING LITERAL TOO LONG
FP	INCORRECT FOR STATEMENT	SS	BAD SUBSCRIPT OR FUNCTION ARG
FR	ERROR IN FUNCTION ARGS	ST	SYMBOL TABLE OVERFLOW
IF	ERROR IN IF STATEMENT	SY	SYSTEM INCOMPLETE
IC	I/C ERROR	TB	PROGRAM TOO BIG
LS	MISSING EQUALS SIGN IN LET	TD	TOO MUCH DATA IN PROGRAM
LT	STATEMENT TOO LONG	TS	TOO MANY CHARS IN STRING
MD	MULTIPLY DEFINED LINE NUMBER	UD	ERROR IN UDEF STATEMENT
ME	MISSING END STATEMENT	UF	FOR STATEMENT WITHOUT NEXT
MO	OPERAND EXPECTED, NOT FOUND	US	UNDEFINED STATEMENT NUMBER
MP	PARENTHESIS ERROR	UU	USE STATEMENT ERROR
MT	OPERAND OF MIXED TYPE	XC	CHARS AFTER END OF LINE
NF	NEXT STATEMENT WITHOUT FOR		

6.12.2 Run-Time System Error Messages

The following error messages are generated by the BASIC run-time system:

BO	NO MORE BUFFERS AVAILABLE	GS	TOO MANY NESTED GOSUBS
CI	IN CHAIN, DEVICE NOT FOUND	IA	ILLEGAL ARG IN UDEF
CL	IN CHAIN, FILE NOT FOUND	IF	ILLEGAL DEV:FILENAME
CX	CHAIN ERROR	IN	INQUIRE FAILURE
DA	READING PAST END OF DATA	IO	TTY INPUT BUFFER OVERFLOW
DE	DEVICE DRIVER ERROR	LM	TAKING LOG OF NEGATIVE NUMBER
DC	NO MORE ROOM FOR DRIVERS	OE	DRIVER ERROR WHILE OVERLAYING
DV	ATTEMPT TO DIVIDE BY ZERO	OV	NUMERIC OR INPUT OVERFLOW
EF	LOGICAL END OF FILE	PA	ILLEGAL ARG IN POS
EM	NEGATIVE NUMBER TO REAL POWER	RE	READING PAST END OF FILE

EN	ENTER ERROR	SC	CONCATENATED STRING TOO LONG
FB	USING FILE ALREADY IN USE	SL	STRING TOO LONG OR UNDEFINED
FC	CLOSE ERROR	SR	READING STRING FROM NUMERIC FILE
FF	FETCH ERROR	ST	STRING TRUNCATION ON INPUT
FI	CLOSING OR USING UNOPENED FILE	SO	SUBSCRIPT OUT OF RANGE
FM	FIXING NEGATIVE NUMBER	SW	WRITING STRING INTO NUMERIC FILE
FN	ILLEGAL FILE NUMBER	VR	READING VARIABLE LENGTH FILE
FO	FIXING NUMBER > 4095	WE	WRITING PAST END OF FILE
GR	RETURN WITHOUT GOSUB		