PHILIPS

Data
Systems

PHILIPS

DISK BASIC

P2000

5103 991 10623 UK

CHAPTER 0

TABLE OF CONTENTS

CHAPTER 1	:	GENERAL INFORMATION ABOUT BASIC
CHAPTER 2	:	BASIC COMMANDS AND STATEMENTS
CHAPTER 3	:	BASIC FUNCTIONS
APPENDIX A	:	BASIC FILE COMMANDS AND FUNCTIONS
APPENDIX B	:	BASIC DISK I/O
APPENDIX C	:	ASSEMBLY LANGUAGE SUBROUTINES
APPENDIX D	:	MATHEMATICAL FUNCTIONS
APPENDIX E	:	ASCII CHARACTER CODES
APPENDIX F	:	CONVERTING BASIC PROGRAMS
APPENDIX G	:	SUMMARY OF ERROR CODES AND ERROR MESSAGES
APPENDIX H	:	THE VOLUME ORGANIZATION UTILITY
APPENDIX I	:	SCREEN I/O MANAGEMENT

CHAPTER 1

GENERAL INFORMATION ABOUT BASIC

1.1 INITIALIZATION

BASIC is initialized automatically at power up.

1.2 MODES OF OPERATION

When BASIC is initialized, it types the prompt "OK". "OK" means BASIC is at command level, that is, it is ready to accept commands. At this point, BASIC may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC commands and statements are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program. Multistatements may be entered in this mode.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

nnnnn BASIC statement [: BASIC statement ...] <carriage return>

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon (:).

A BASIC program line always begins with a line number, end with a carriage return and may contain a maximum of 255 characters.
It is possible to extend a logical line over more than one physical line by use of the terminal's auto-linefeed. Auto-linefeed lets you continue typing a logical line on the next physical line without typing a <carriage-return>.

1.3.1 LINE - NUMBERS

Every BASIC program line begins with a line number. Line Numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line Numbers must be in the range 0 to 65529. A Period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

1.4 CHARACTER SET

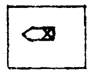
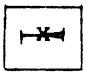
The BASIC character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in BASIC are the upper case and lower case letters of the alphabet.

The numeric characters in BASIC are the digits 0 through 9.

The following special characters and terminal keys are recognized by BASIC :

Character:	Name:	
	Blank	
;	Semicolon	
=	Equal sign or assignment symbol	
+	Plus sign	
-	Minus sign	
*	Asterisk or multiplication symbol	
/	Slash or division symbol	
	Up arrow or exponentiation symbol	↑ ↖
(	Left parenthesis	
)	Right parenthesis	
%	Percent	
#	Number (or pound) sign	
$	Dollar sign	
!	Exclamation point	
[	Left bracket	[
]	Right bracket	]
,	Comma	
.	Period or decimal point	
'	Single quotation mark (apostrophe)	
"	Double quotation mark	
:	Colon	
&	Ampersand	
?	Question mark	
	Less than	
	Greater than	
	Backslash or integer division symbol	\ /
@	At-sign	
escape	Escapes Edit Mode subcommands.	CODE
carriage return	Terminates input of a line.	↵

	Backspace. Deletes the last character typed.
TAB	Moves print position to next tab stop Tab stops are every eight columns.
	Deletes the line that is currently being typed.
SHIFT-TAB	Changes the keyboard in the so called typewriter mode vice versa. In this mode it is possible to enter lower case letters.
SHIFT-STOP	BREAKs program.
SHIFT-5 (keypad)	Suspends program.
SHIFT-START	Resumes a suspended program.

1.5 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$ 25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants	Whole numbers between and including -32768 and + 32767. Integer constants do not have decimal points.
2. Fixed Point constants	Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating Point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent).
The allowable range for floating point constants is approximately:
10^{-38} to 10^{+37}
Examples:
235.988E-7= .0000235988
2359E6 = 2359000000

(Double precision floating point constants use the letter D instead of E. See Section 1.5.1.)

4. Hex constants

Hexadecimal numbers with the prefix &H. Examples:

&H76
&H32F

The allowable range is:
&H0000 to &HFFFF

5. Octal constants

Octal numbers with the prefix &O or & Examples:

&O347
&1234

The allowable range is:
&O000000 to &O177777

1.5.1 Single And Double Precision Form For Fixed/Floating Point Constants

Fixed/floating point constants may be either single precision or double precision numbers. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or

2. exponential form using E, or

3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or

2. exponential form using D, or

3. a trailing number sign (#)

Examples:

Single Precision Constants	Double Precision Constants
46.8	345692811
- 7.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

1.6 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, integer,(real) single precision and double precision variables are assumed to be zero, string variables are assumed to be a zero-length string (i.e."").

1.6.1 Variable Names And Declaration Characters

BASIC variable names may be any length, however, only the first 40 characters are significant. The characters allowed in a variable name are letters and numbers and the decimal point. The first character must be a letter. Special type declaration characters are also allowed and are a significant part of the variable name -- see below.

A variable name may not be a reserved word, but BASIC allows embedded reserved words. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC commands, statements, function names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single or double precision values. The default type of a numeric variable is (real) single precision, unless otherwise specified. The type declaration characters for these variable names are as follows:

		Precision
%	Integer variable	5 digits (-32768 to +32767)
!	Single precision variable	7 digits
#	Double precision variable	16 digits

The default type for a numeric variable name is single precision.

Examples of BASIC variable names follow.

PI#	declares a double precision value
MINIMUM!	declares a single precision value
LIMIT%	declares an integer value

There is a second method by which variable types may be declared. The BASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Chapter 2.

1.6.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array variable name has as many subscripts as there are dimensions in the array. For example for V (10) the subscript range is from 0 to 10. V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.
If an array is subscripted by a single-precision or double-precision expression, the subscript is converted to integer (implicit conversion). The expression is rounded, not truncated.

1.7 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)
   Example:

   ```
   10 A% = 23.42
   20 PRINT A%
   RUN
    23
   ```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.
   Examples:

   ```
   10 D# = 6#/7         The arithmetic was performed
   20 PRINT D#          in double precision and the
   RUN                  result was returned in D#
   .8571428571428571    as a double precision value.
   ```

   ```
   10 D = 6#/7          The arithmetic was performed
   20 PRINT D           in double precision and the
   RUN                  result was returned to D (single
   .857143              precision variable), rounded and
                        printed as a single precision
                        value.
   ```

3. Logical operators convert their operands to integers and return an integer result.
   Operands must be in the range - 32768 to 32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded.
   Example:

   ```
   10 C% = 55.88
   20 PRINT C%
   RUN
    56
   ```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value.
   Example:

```
10A = 2.04
20 B# = A
30 PRINT A; B#
RUN
 2.04  2.039999961853027
```

1.8. EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC may be divided into four categories.

1. Arithmetic
2. Relational
3. Logical
4. Functional

1.8.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
↑	Exponentiation	X↑Y
-,+	Unary + or -	-X
*,/	Multiplication, Floating Point Division	X*Y X/Y
\	Integer Division	X\Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
$X+2Y$	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X 2) Y
X^{Y^Z}	X (Y Z)
$X(-Y)$	X*(-Y) or X*-Y

1.8.1.1 Integer Division And Modulus Arithmetic

Integer division is denoted by the baskslash ( ). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer. For example:

```
10 4 = 2
25.68 6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

1.8.1.2 Overflow And Division By Zero -

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then used to make a decision regarding program flow.

Operator	Relation Tested	Expression
=	Equality	X=Y
<> or ><	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<= or =<	Less than or equal to	X<=Y
>= or =>	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable).

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X) < 0 GOTO 1000
IF I MOD  J<>0 THEN K=K+1
```

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision.

For example:

```
IF D<200  AND F>4 THEN 80
IF I < 10 OR K > 0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.). If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands. True is represented by -1, false by 0.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator maybe used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
- 1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC's intrinsic functions are described in Chapter 3.

BASIC also allows "user defined" functions that are written by the programmer.

1.8.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE"
20 B$="NAME"
30 C$="NEW " + A$ + B$
40 PRINT A$ + B$
50 PRINT C$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=                                =      =
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA"    "AB"
"FILENAME" = "FILENAME"
"X&"    "X#"
"kg"    "KG"
"SMYTH"    "SMYTHE"
B$  "9/12/78"      where  B$ = "8/12/78"
"B"    "AB"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT key.

Rubout has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type .

A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC will automatically replace the old line with the new line.

For more sophisticated editing capabilities see EDIT.

To delete the entire program that is currently residing in memory, enter the NEW command. NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If BASIC detects an error that causes program execution to terminate, an error message is printed. For a complete list of BASIC error codes and error messages, see Appendix C.

CHAPTER 2

BASIC COMMANDS AND STATEMENTS

All of the BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Format:	Shows the correct format for the instruction. See below for format notation.
Purpose:	Tells what the instruction is used for.
Remarks:	Describes in detail how the instruction is used.
Example:	Shows sample programs or program segments that demonstrate the use of the instruction.

<u>Format Notation</u>

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.
2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.
4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).
6. Items separated by a vertical bar (|) are mutually exclusive; choose one.

2.1 AUTO

Format: AUTO [<line number>[,<increment>]]

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will delate the current line and generate the next line number. If a text is entered before typing carriage return, the text of the current line is replaced and the next line number is generated.

AUTO is terminated by typing SHIFT-STOP. The line in which SHIFT-STOP is typed is not saved. After SHIFT-STOP is typed, BASIC returns to command level.

Example:

AUTO 100, 50	Generates line numbers 100, 150, 200 ...
AUTO	Generates line numbers 10, 20, 30, 40 ...

2.2 CALL

Format: CALL <variable name> [<argument list>]

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an assembly language subroutine. (See also the USR function).

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. argument list contains the arguments that are passed to the assembly language subroutine.

The CALL statement generates the same calling sequence used by BASIC.

Example:

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
       .
       .
       .
```

2.3 CHAIN

Format: CHAIN [MERGE] <filename>[,[<line number exp>]

Purpose: To call a program and pass variables to it from the current program.

Remarks: <filename> is the name of the program that is called. Example:

```
CHAIN"PROG1"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. Example:

```
CHAIN"PROG1",1000
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. Example:

```
CHAIN"PROG1", 1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE"OVERLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. Example:

```
CHAIN MERGE"OVERLAY2",1000,DELETE 1000-5000
```

The line numbers in <range> are affected by the RENUM command.

NOTE: If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statement containing shared variables must be restated in the chained program.

2.4 CLEAR

Format: CLEAR [, expression-1 , expression-2]

Purpose: - to undo all declarations of variables
- to make a redivision of the available memory

Remarks: expression-1 specifies the upper boundary of memory to be used by BASIC. Beyond this boundary, applications can place machine code programs or any other information. (e.g. by POKE statements)

expression-2 is the size of the stackarea to be used by the BASIC system. The minimum value is about 100, but should be specified greater if the application has nested constructions like FOR...NEXT loops, recursive subroutines or complex expressions. (Recommended: greater than 400)

Examples:

```
CLEAR

CLEAR ,&HC000, 400
```

2.5 CLOSE

Format: CLOSE [[#]<file number>[,[#]<file number>..]]

Purpose: To conclude I/O to a disk file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: See appendix B.

2.6 COMMON

Format: COMMON <list of variables>

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
```

2.7 CONT

Format: CONT

Purpose: To continue program execution after a SHIFT-STOP SHIFT-STOP has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usuallay used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break, and the program reports "Can't Continue".

2.8 DATA

Format: DATA <list of constants>

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s).

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement. If the program tries to access DATA via the READ statement after the last DATA item, the program reports an "out of data" error.

DATA statements may be reread from the beginning by use of the RESTORE statement.

2.9 DEF FN

Format: DEF FN <name>[(<parameter list>)]=<function definition>

Purpose: To define and name a function that is written by the user.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separeted by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

In BASIC, user-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
410 DEF FNAB (X,Y)=X 3/Y 2
420 T=FNAB(I,J)
```

Line 410 defines the function FNAB.
The function is called in line 420.

2.10 DEFINT/SNG/DBL/STR

Format: DEF<type> <range(s) of letters>
where <type> is INT, SNG,DBL, or STR

Purpose: To declare variable types as integer, single precisions, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable.

If no type declaration statements are encountered, BASIC assumes all variables without declaration characters are single precision variables.

Examples:

10 DEFDBL L-P — All variables beginning with the letters L,M,N, O and P will be double precision variables.

10 DEFSTR A — All variables beginning with the letter A will be string variables.

10 DEFINT I-N, W-Z — All variables beginning with the letters I, J, K, L, M, N, W, X, Y and Z will be integer variables.

2.11 DEF USR

Format: DEF USR [<digit>]=<integer expression>

Purpose: To specify the starting address of an assembly language subroutine.

Remarks: <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Appendix C, Assembly Language Subroutines.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessarily.

Example:

```
.
.
.
200 DEF USR0=24000
210 X=USR0 (Y↑2/2.89)
.
.
.
```

2.12 DELETE

Format: DELETE[<line number>] [-<line number>]

Purpose: To delete program lines.

Remarks: BASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples:

DELETE 40	Deletes line 40
DELETE 40-100	Deletes lines 40 through 100, inclusive
DELETE-40	Deletes all lines up to and including line 40

2.13 DIM

Format: DIM <list of subscripted variables>

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless specified otherwise with the OPTION BASE statement.
The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A (20)
20 FOR I=O TO 20
30 READ A (I)
40 NEXT I
    .
    .
    .
```

2.14 EDIT

Format: EDIT <line number>

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions.

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting Edit Mode

NOTE

In the descriptions that follow, <ch> represents any character, text represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ or Escape (see text) represents the CODE key.

1. Moving the Cursor

Space	Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.
Rubout	In Edit Mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I	I<text> inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Rubout or Delete key on the terminal may be used to delete characters to the left of the cursor. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell is typed and the character is not printed.
X	The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, type CODE or Carriage Return.

3. Deleting Text

D	[i]D deletes i characters to the right of the cursor. The deleted characters are echoed between quotes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.
H	H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

4. Finding Text

S	The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of

the line. All characters passed over during the search are printed.

K The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in quotes.

5. Replacing Text

C The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. Ending and Restarting Edit Mode

<cr> Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.

E The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q The Q subcommand returns to BASIC command level, without saving any of the changes that were made to the line during Edit Mode.

L The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

If BASIC receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell and the command or character is ignored.

Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type Carriage Return (or the E subcommand), BASIC reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first exit Edit Mode with the Q subcommand. BASIC will return to command level, and all variable values will be preserved.

NOTE

> Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

2.15 END

Format: END

Purpose: To terminate program execution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:

```
520 IF K <1000 THEN END ELSE GOTO 20
```

2.16 ERASE

Format: ERASE <list of array variables>

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Duplicate definition" error occurs.

Example:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

2.17 ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ...

```
IF ERR = <error code>THEN ...

IF ERL = <line number>THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. BASIC's error codes are listed in Appendix G.

2.18 ERROR

Format: ERROR <integer expression>

Purpose: 1) To simulate the occurrence of a BASIC error; or 2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of integer expression equals an error code already in use by BASIC (see Appendix C), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by BASIC's codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15         (you type this line)
String too long (BASIC types this line)
Ok
```

Example 2:

```
    .
    .
    .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET"; B
130 IF B > 5000 THEN ERROR 210
    .
    .
    .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
    .
    .
    .
```

2.19 FIELD

Format: FIELD[#]<file number>,<field width> AS <string variable>...

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to string variable. For example,

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET).

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 256.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example: See Appendix B.

NOTE: Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

2.20 FOR ... NEXT

Format:

```
FOR <variable>=x TO y [STEP z]
.
.
.
NEXT [<variable>] [,<variable>...]
```

where x, y and z are numeric expressions.

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
1  20
3  30
5  40
7  50
9  60
Ok
```

Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
RUN
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

In this example, the loop executes ten times. The final value of the loop variable is always set before the initial value is set.

2.21 GET

Format: GET[#]<file number>[,<record number>]

Purpose: To read a record from a random disk file into a random buffer.

Remarks: <file number> is the number under which the file was OPENed. If record number is omitted, the next record (after the last GET) is read into the buffer. The lowest possible record number is 1.

Example: See Appendix B.

2.22 GOSUB...RETURN

Format: GOSUB <line number>

.
.
.

RETURN

Purpose: To branch to and return from a subroutine.

Remarks: <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine. If the <line number> is not valid, a "Undefined line number in xx" error is reported.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN ";
60 PRINT " PROGRESS "
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.23 GOTO

Format: GOTO <line number>

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after line number . If the <line-number> is not valid, a "undefined line number in xx" error is reported.

Example:

```
LIST
10 READ R
20 PRINT "R=";R,
30 A = 3.14*R↑2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5, 7, 12
Ok
RUN
R = 5                AREA = 78.5
R = 7                AREA = 153.86
R = 12               AREA = 452.16
?Out of data in 10
Ok
```

2.24 IF...THEN[...ELSE] AND IF...GOTO

Format: IF <expression> THEN <statement(s)> | <line number>

[ELSE <statement(s)> | <line number>]

Format: IF <expression> GOTO <line number>

[ELSE <statement(s)> | <line number>]

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of expression is not zero, the THEN or GOTO clause is executed. ("true "is represented by a non-zero value).THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of expression is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement.

Nesting of IF Statements

IF ...THEN...ELSE ... statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X Y THEN PRINT "GREATER" ELSE IF Y<X
    THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"
    ELSE PRINT "A<>C"
```

will not print "A<>C", when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1:

```
200 IF I THEN GET #1,I
```

This statement GETs record number I if I is not zero.

Example 2:

```
100 IF (I<20) AND (I>10) THEN DB= 1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
          .
          .
          .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

2.25 INPUT

Format: INPUT [<"prompt string">;] <list of variables>

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

The data that is entered is assigned to the variable(s) given in variable list . The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks. However, if the string is surrounded by quotation marks, the quotation marks are stripped off).

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Examples:

```
10  INPUT X
20  PRINT X "SQUARED IS" X↑2
30  END
RUN
?  5      (The 5 was typed in by the user
          in response to the question mark.)
 5  SQUARED IS 25
Ok
```

```
LIST
10 PI= 3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R↑2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS ? 7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS  171.946

WHAT IS THE RADIUS ?
etc.
```

2.26 INPUT#

Format: INPUT#<file number>,<variable list>

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example: See Appendix B.

2.27 KILL

Format: KILL <filename>

Purpose: To delete a file from disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files and sequential data files.

Example: 200 KILL "MYJOB. BAS"

See also Appendix B.

2.28 LET

Format: [LET] <variable>=<expression>

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e. the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110  LET D=12
120  LET E=12*2
130  LET F=12*4
140  LET SUM=D+E+F
        .
        .
        .

        or

110 D=12
120 E=12*2
130 F=12*4
140 SUM=D+E+F
        .
        .
        .
```

2.29 LINE INPUT

Format: LINE INPUT [<"prompt string">;]<string variable>

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing SHIFT-STOP. BASIC will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

2.30 LINE INPUT#

Format: LINE INPUT#<file number>,<string variable>

Purpose: To read an entire line (up to 254 characters), without delimeters, from a sequential disk data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is espacially useful is each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION?;C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES 234,4 MEMPHIS
LINDA JONES  234,4   MEMPHIS
Ok
```

2.31 LIST

Format: LIST [<line number1>][-][<line number2>]

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC always returns to command level after a LIST is executed.

Listing is terminated either by the end of the program or by typing SHIFT STOP.

The format allows the following options:

1. LIST
   or
   LIST -
   All of the program currently in the memory is listed at the terminal.

2. LIST <line number1>
   Only the specified line number is listed.

3. LIST <line number1> -
   The program is listed beginning at that line.

4. LIST - <line number2>
   All lines from the beginning of the program through that line are listed.

5. LIST <line number1> - <line number2>
   The entire range is listed.

Examples:	LIST	Lists the program currently in memory.
	LIST 150-	Lists all lines from 150 to the end.
	LIST -1000	Lists all lines from the lowest number through 1000.
	LIST 150-1000	Lists lines 150 through 1000, inclusive.
	LIST 500	Lists line 500.

2.32 LLIST

Format: LLIST [<line number1>][-][<line number2>]

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes a 132-character wide printer.

BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST.

Example: See the examples for LIST.

2.33 LOAD

Format: LOAD <filename>[,R]

Purpose: To load a file from disk into memory.

Remarks: <filename> is the name that was used when the file was SAVEd. (The default extension .BAS is supplied.)

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example: LOAD "MYJOB",R

2.34 LPRINT AND LPRINT USING

Format: LPRINT [<list of expressions>]

LPRINT USING <string exp>; <list of expressions>

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer.

LPRINT assumes a 132-character-wide printer.

2.35 LSET AND RSET

Format:

```
LSET <string variable>=<string expression>
RSET <string variable>=<string expression>
```

Purpose: To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions.

Examples:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC($)
```

See also Appendix B.

NOTE: LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

2.36 MERGE

Format: MERGE <filename>

Purpose: To merge a specified disk file into the program currently in memory.

Remarks: <filename> is the name used when the file was SAVEd. (The default extension .BAS is supplied.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC always returns to command level after executing a MERGE command.

Example: MERGE "MYJOB"

2.37 MID$

Format: MID$ (<string exp1>, n[,m])=<string exp2>

where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Purpose: To replace a portion of one string with another string.
MID$ may also be used as a function that returns a substring of a given string (see section 3.24).

Remarks: The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

Example:

```
10 A$="KANSAS CITY, MO"
20 MID$ (A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

2.38 NAME

Format: NAME <old filename> AS <new filename>

Purpose: To change the name of a disk file.

Remarks: <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the newname.

Example:

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

2.39 NEW

Format: NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

2.40 NULL

Format: NULL <integer expression>

Purpose: To set the number of nulls to be printed at the end of each line.

Remarks: <integer expression> should be 0 or 1 for Teletype-compatible CRTs. <integer expression> should be 2 or 3 for 30 cps hard copy printers. The default value is 0.

Example:

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X>50 GOTO 800
      .
      .
      .
```

Two null characters will be printed after each line.

2.41 ON ERROR GOTO

Format: ON ERROR GOTO <line number>

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine. The error trapping is disabled by the CLEAR command.

Example:

```
10 ON ERROR GOTO 1000
```

2.42 ON...GOSUB AND ON...GOTO

Format: ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of expression is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example: 100 ON L-1 GOTO 150,300,320,390

2.43 OPEN

Format: OPEN <mode>,[#]<file number>,<filename>,[<reclen>]

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

"O"	specifies sequential output mode
"I"	specifies sequential input mode
"R"	specifies random input/output mode

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to the BASIC rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 256 bytes.

NOTE: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example:

```
10 OPEN "I",2,"INVEN"
```

See also Appendix B.

2.44 OPTION BASE

Format: OPTION BASE n
where n is 1 or 0

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.
If more than one OPTION BASE statement is specified in a program, a "Duplicate Definition" error occurs, until a CLEAR is specified.

2.45 OUT

Format: OUT I,J
where I and J are integer expressions in the range 0 to 255.

Purpose: To send a byte to a machine output port.

Remarks: The integer expression I is the port number, and the integer expression J is the data to be transmitted. (See INP Function for input port handling, Section 3.15)

Example:

```
100 OUT 32, 100
```

2.46 POKE

Format: POKE I, J
where I and J are integer expressions

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. I must be in the range 0 to 65536.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. (See PEEK Function, Section 3.27)

POKE and PEEK are useful for efficient data storage, loading machine code subroutines, and passing arguments and results to and from machine code or assembly language subroutines.

Example:

```
10 POKE &H5A00, &HFF
```

2.47 PRINT

Format: PRINT [<list of expressions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the lines into print zones of 14 character positions each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 10 (-6) is output as .000001 and 10 (-7) is output as 1E-7. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-16 is output as .0000000000000001 and 1D-17 is output as 1D-17.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X↑5
30 END
RUN
 10            0            -25            3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X↑2 "AND";
30 PRINT X "CUBED IS" X↑3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261
?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5   10  10   20   15   30   20   40   25   50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.48 PRINT USING

Format: PRINT USING <string exp>; <list of expressions>

Purpose: To print strings and/or numbers using a specified format, possibly intermixed with text.

Remarks and Examples: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) that is comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

" n spaces " Specifies that 2+n characters from the string are to be printed. If the double quotation marks are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "    ";A$;B$
50 PRINT USING "    ";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK": B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78
```

```
PRINT USING "###.##";987.654
987.66
```

```
PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20     5.30    66.79    0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

\- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
-68.95      +2.40     + 55.60      - 0.90

PRINT USING "##.##-   ";-68.95,22.449,-7.01
68.95-      22.45      7.01-
```

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#   ";12.39,-0.9,765.1
*12.4     *-0.9       765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
   $ 456.78
```

**$ The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be astersk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING  "**$##.##";2.34
***$2.34
```

, A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,"; 1234.5
1234.50,
```

↑↑↑↑ Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.## ↑↑↑↑"; 234.56
  2.35E+02

PRINT USING ".#### ↑↑↑↑-";888888
  .8889E+06

PRINT USING "+.## ↑↑↑↑";123
+.12E+03
```

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

```
10 A$=" JOHN ###.## ! PETER &     "
20 B$="***" : C=123.457
30 LPRINT USING A$;C;B$
40 LPRINT USING A$;C,B$,B$
50 LPRINT USING A$;C;B$;B$,C
RUN
 JOHN 123.46 * PETER
 JOHN 123.46 * PETER ***
 JOHN 123.46 * PETER ***      JOHN 123.46
```

2.49 PRINT# AND PRINT# USING

Format: PRINT#⟨filenumber⟩,[USING⟨string exp⟩;]⟨list of exps

Purpose: To write data to a sequential disk file.

Remarks: ⟨filenumber⟩ is the number used when the file was OPENed for output. ⟨string exp⟩ is comprised of formatting characters as described in Chapter 2, PRINT USING. The expressions in ⟨list of expressions⟩ are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blacks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34).

For example let A$="CAMERA, AUTOMATIC" andB$=" 93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC  93604-1
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC   93604-1" to B$. To separate these strings properly on the disk, write double quotes to the disk image using CHR$(34). The statement

```
PRINT#1, CHR$(34);CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC""    93604-1"
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and "   94603-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

For more examples using PRINT#, see Appendix B.

See also WRITE#.

2.50 PUT

Format: PUT [#]<file number>[,<record number>]

Purpose: To write a record from a random buffer to a random disk file.

Remarks: <file number> is the number under which the file was OPENed. If record number is omitted, the record will have the next available record number (after the last PUT). The lowest possible record number is 1.

Example: See Appendix B.

2.51 RANDOMIZE

Format: RANDOMIZE [<expression>]

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC suspends program execution and asks for a value by printing

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)?
3 (user types 3)
 .88598  .484668  .586328  .119426  .709225
Ok
RUN
Random Number Seed (-32768 to 32767)?
4 (user types 4)
 .803506  .162462  .929364  .292443  .322921
Ok
RUN
Random Number Seed (-32768 to 32767)?
3 (same sequence as first RUN)
 .88598  .484668  .586328  .119426  .709225
Ok
```

NOTE: The following construction should be used to generate a really random start value.

```
10 RANDOMIZE 256 * PEEK(&H6011) + PEEK(&H6010) - 32768
```

2.52 READ

Format: READ <list of variables>

Purpose: To read values from a DATA statement and assign them to variables.

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statment(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement

Example 1:

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
LIST
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$, S$, Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$, S$, Z
Ok
RUN
CITY          STATE          ZIP
DENVER,       COLORADO       80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

2.53 REM

Format: REM <remark> or '<remark>

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Example:

```
     .
     .
     .
120  REM CALCULATE AVERAGE VELOCITY
130  FOR I=1 TO 20
140  SUM=SUM + V (I)
     .
     .
     .
```

or,

```
     .
     .
     .
120 FOR I=1 TO 20     : 'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
     .
     .
     .
```

or,

```
120 ' This is a remark
```

2.54 RENUM

Format: RENUM [[<new number>][,[<old number>][,<increment>]]]

Purpose: To renumber program lines.

Remarks: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON....GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10,20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 300,,50	Renumbers the entire pro- pram. The first new line number will be 300. Lines will increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

2.55 RESTORE

Format: RESTORE [<line number>]

Purpose: To allow DATA statements to be reread from a specified point.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
     .
     .
     .
```

2.56 RESUME

Formats:

```
RESUME

RESUME 0

RESUME NEXT

RESUME <line number>
```

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement causes a "RESUME without error" message to be printed, if no error trap routine is specified or no error has occured.

Example:

```
10 ON ERROR GOTO 900
   .
   .
   .
   .
900 IF (ERR=230)AND(ERL=90)THEN PRINT "TRY
AGAIN": RESUME 80
   .
   .
   .
```

2.57 RUN

Format 1: RUN [<line number>]

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number.

Example: RUN

Format 2: RUN <filename>[,R]

Purpose: To load a file from disk into memory and run it.

Remarks: <filename> is the name used when the file was SAVEd. (The default extension .BAS is supplied.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example: RUN "NEWFIL",R

See also Appendix B.

2.58 SAVE

Format: SAVE <filename>[,A | ,P]

Purpose: To save a program file on disk.

Remarks: <filename> is a quoted string that conforms to the BASIC requirements for filenames. (The default extension .BAS is supplied.) If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples:

```
SAVE"COM2",A
SAVE"PROG",P
```

See also Appendix B.

2.59 STOP

Format: STOP

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

Example:

```
10 INPUT A,B,C
20 K=A↑2*5.3:L=B↑3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

2.60 SWAP

Format: SWAP <variable>,<variable>

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
LIST
10 A$=" ONE " : B$=" ALL" : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

2.61 TRON/TROFF

Format: TRON

TROFF

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINTJ;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1   10   20
[50][60][30][40] 2   20   30
[50][60][70]
Ok
TROFF
Ok
```

2.62 VARPTR

Format 1: VARPTR (<variable name>)

Format 2: VARPTR (#<file number>)

Action: Format 1: Returns the address of the first byte of data identified with <variable name>. A value must be assigned to variable name prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(O)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Warning: All simple variables should be assigned before calling VARPTR for an array, because the addressed of the arrays change whenever a new simple variable is assigned.

Format 2: Returns the starting address of the disk I/O buffer assigned to <file number>.

Examples:

```
100 XX=USR(VARPTR(Y))
```

or

```
10 REM SEE WARNING
20 DEFINT A-Z
30 DIM X(10)
40 X(8)=31
50 A=VARPTR(X(8))
60 I = 111 : REM CAUSES A SHIFT OF THE ARRAY
70 LPRINT A
80 LPRINT PEEK (A+1);PEEK(A)
90 LPRINT
100 A=VARPTR(X(8))
110 LPRINT A
120 LPRINT PEEK(A+1);PEEK(A)
RUN
-27865
 0  0

-27859
 0  31
```

2.63 WAIT

Format: WAIT <port number>, I[,J]
where I and J are integer expressions

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example:

```
100 WAIT 32,2
```

2.64 WHILE...WEND

Format:

```
WHILE <expression>
   .
   .
[<loop statements>]
   .
   .
WEND
```

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statement> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115         FLIPS=0
120         FOR I=1 TO J-1
130                 IF A$(I)>A$(I+1) THEN
                            SWAP A$(I),A$(I+1):FLIPS=1
140         NEXT I
150 WEND
```

NOTE: WHILE...WEND is not implemented in most BASICs.

2.65 WIDTH

Format: WIDTH [LPRINT] <integer expression>

Purpose: To set the printed line width in number of characters for the terminal or line printer.

Remarks: If the LPRINT options is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If < integer expression> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor of the print head, as given by the POS or LPOS function, returns to zero after position 255.

2.66 WRITE

Format: WRITE [<list of expressions>]

Purpose: To output data at the terminal

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement without leading and trailing blanks.

Example:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
Ok
```

2.67 WRITE#

Format: WRITE#<filenumber>,<list of expressions>

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

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

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.