

THE WANG PROFESSIONAL COMPUTER

BASIC
Language
Guide

WANG

The Wang Professional Computer BASIC Language Guide

1st Edition — December 1982
Copyright © Wang Laboratories, Inc., 1982
Portions reproduced with permission of Microsoft, Inc.
700-8019

WANG

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL: 617/459-5000, TWX 710-343-6769, TELEX 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase, lease, or license agreement by which this software package was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of the software package, the accompanying manual, or any related materials.

NOTICE:

All Wang Program Products are licensed to customers in accordance with the terms and conditions of the Wang Laboratories, Inc. Standard Program Products License; no ownership of Wang Software is transferred and any use beyond the terms of the aforesaid License, without the written authorization of Wang Laboratories, Inc., is prohibited.

WANG

PREFACE

The purpose of the Wang Professional Computer BASIC Language Guide is to acquaint you with Wang Professional Computer Interpretive BASIC. This guide is designed to be used by programmers who are familiar with BASIC. Programmers who are unfamiliar with BASIC should consult any standard BASIC textbook for an introduction to the language.

Chapter 1 introduces you to the PC Interpretive BASIC. Chapter 2 shows you how to enter and exit PC Interpretive BASIC. Chapter 3 tells you how to edit program text using the PC Interpretive BASIC full screen editor. Chapter 4 describes statements used in program entry and manipulation. Chapter 5 explains the commands used in transferring programs to and from disk. Chapter 6 discusses statements used in such program control transfers as subroutines, conditional statements, loops, and error traps. Chapter 7 describes the statements used in formatting data. Chapter 8 provides you with information on data file input/output and its associated statements. Chapter 9 describes each of the arithmetic, string, and logical functions. Chapter 10 discusses instructions specifically used in debugging programs. Chapter 11 explains the instructions available for producing graphic images. Chapter 12 explains the PC Interpretive BASIC instructions that create music and other sounds.

In addition to this manual, you may find it helpful to also refer to the Wang Professional Computer Introductory Guide and the Wang Professional Computer Compiled BASIC Reference Manual.

This guide uses specific conventions to indicate compatibility with the Wang Professional Computer BASIC compiler.

A compiler instruction status symbol indicates whether each instruction can be used in a PC Compiled BASIC program. This status is useful if you intend to develop interpretive BASIC programs which you want to convert to compiled BASIC programs. There are three different status conditions.

- If the top right of an instruction page is blank, you can use the particular instruction in a compiled BASIC program.
- If the top right of an instruction page contains (NC), you cannot use the particular instruction in a compiled BASIC program.
- If the top right of an instruction page contains (C), you can use the instruction in a compiled BASIC program only under specific conditions. In cases where (C) occurs, refer to the comments section of the instruction page to find the particular restrictions.

Throughout the manual, a general format accompanies each description of a statement, command, or function. When more than one specific arrangement is permitted, separate, numbered formats are shown. Within a general format, keywords, connectives, and special characters are shown in proper sequence. Unless otherwise stated, only the shown sequence can be used.

The general formats use the following conventions.

- Each capitalized word or letter represents a required part of the instruction line. You must type in all the capitalized items that appear in the format line. However, when you type in an instruction you do not have to capitalize these required items.
- Wherever angle brackets (< >) enclose an element, you must supply a legal BASIC representative of that element.
- Square brackets ([]) indicate optional items. In general, when these items appear in a format line you have more flexibility in the design of your instruction.
- A logical Or bar (|) indicates a choice. When a logical Or bar appears in an instruction line, you can choose a parameter from either side of the bar.
- Items followed by an ellipsis (. . .) may be repeated any number of times up to the length of the line (255 characters).
- You must use all punctuation marks in an instruction in the position they are shown in the format line. However, you should never include in an instruction any of the symbols including angle brackets, square brackets, or the logical Or bar. If you do not use the indicated punctuation in an instruction line you will create a "Syntax error".
- Blank spaces between parameters in a format line show the recommended form. You must include either a blank space or a valid form of punctuation between parameters. BASIC is insensitive to the distance between an instruction's parameters. For example, each of these instructions is syntactically correct:

```
10 PRINT "That's my line": INPUT "What's yours?";Y$
10PRINT"That's my line":INPUT"What's yours?";Y$
10 PRINT      "That's my line"      : INPUT      "What's
yours?"      ;      Y$
```

CONTENTS

CHAPTER 1 OVERVIEW

1.1	Introduction	1-1
1.2	Instructions	1-1
1.3	Constants	1-2
	String Constants	1-2
	Numeric Constants	1-2
1.4	Variables	1-3
	Variable Names and Declaration Characters	1-4
	Array Variables	1-5
	Memory Space Requirements	1-5
	Variable Type Conversion	1-5
1.5	Operators and Expressions	1-7
	Arithmetic Operators	1-8
	Integer Division and Modulus Arithmetic	1-9
	Overflow and Division by Zero	1-9
	Relational Operators	1-9
	Logical Operators	1-10
	Functional Operators	1-12
	String Operations	1-13

CHAPTER 2 ENTERING AND EXITING BASIC

2.1	Introduction	2-1
2.2	Loading BASIC from the System Software Diskette	2-1
	Loading BASIC from the Program Development Menu	2-2
	Loading BASIC from the DOS Command Processor	2-3
	Loading BASIC from the "Other" Selection	2-3
2.3	Initialization	2-4
2.4	Exiting BASIC	2-5
	The SYSTEM Command	2-5
	The Restart Procedure	2-5

CHAPTER 3 ENTERING AND EDITING BASIC TEXT

3.1	Modes of Operation	3-1
	Direct Mode	3-1
	Indirect Mode	3-2
3.2	Program Lines	3-2
	Line Numbers	3-2
	Extending a Program Line to Another Physical Line	3-3
	Placing More than One Instruction on a Line	3-3
3.3	Program Development Features	3-3

CONTENTS (continued)

	Clearing Memory: NEW Command	3-3
	Automatic Line Numbering: AUTO Command	3-4
	Renumbering Program Lines: RENUM Command	3-4
	Entry of BASIC Keywords	3-4
3.4	Entering Program Lines	3-5
	Adding a New Line to a Program	3-5
	Replacing an Existing Line	3-5
	Deleting an Existing Line	3-5
3.5	BASIC Editing Features	3-5
	List Program Lines: LIST Command	3-6
	Recalling a Single Program Line from Memory:	
	EDIT Command	3-6
	Deleting a Line from Memory: DELETE Command	3-6
3.6	Altering Program Text with the Editor	3-6
	Cursor Positioning and Tabbing	3-8
	Deleting Characters and Words	3-8
	Erasing Text	3-9
	Inserting Text	3-9
3.7	BASIC Debugging Features	3-9
	Error Messages	3-9
	Listing Text on a Screen or Printer: LIST and LLIST ...	3-10
	Stopping and Resuming Program Execution	3-10
	Tracing Program Execution	3-11
3.8	Listing the Disk Directory	3-11

CHAPTER 4 PROGRAM ENTRY AND SCREEN MANIPULATION COMMANDS

4.1	Introduction	4-1
4.2	AUTO	4-1
4.3	CLS	4-3
4.4	DELETE	4-4
4.5	EDIT	4-6
4.6	FILES	4-7
4.7	KEY	4-8
4.8	LIST	4-10
4.9	LLIST	4-11
4.10	LOCATE	4-12
4.11	NEW	4-14
4.12	RENUM	4-15
4.13	RUN	4-16
4.14	SCREEN	4-17
4.15	WIDTH	4-18

CHAPTER 5 TRANSFERRING PROGRAMS TO AND FROM DISK

5.1	Introduction	5-1
	File Naming Conventions	5-1
5.2	KILL	5-3
5.3	LOAD	5-4

CONTENTS (continued)

5.4	MERGE	5-6
5.5	NAME	5-8
5.6	SAVE	5-9
5.7	RUN	5-10

CHAPTER 6 TRANSFER OF PROGRAM CONTROL

6.1	Introduction	6-1
6.2	CALL	6-1
6.3	CHAIN	6-4
6.4	END	6-6
6.6	FOR...NEXT	6-7
6.7	GOSUB...RETURN	6-10
6.8	GOTO	6-12
6.9	IF...THEN[...ELSE] and IF...GOTO	6-13
6.10	ON ERROR GOTO	6-16
6.11	ON PEN/KEY/PEN/STRIG	6-17
6.12	ON...GOSUB and ON...GOTO	6-19
6.13	RESUME	6-20
6.14	STOP	6-22
6.15	WAIT	6-24
6.16	WHILE...WEND	6-25

CHAPTER 7 DATA INPUT/OUTPUT INSTRUCTIONS

7.1	Introduction	7-1
7.2	CLEAR	7-1
7.3	COMMON	7-3
7.4	DATA	7-5
7.5	DATE\$	7-7
7.6	DEF FN	7-8
7.7	DEFINT/SNG/DBL/STR	7-10
7.8	DEF SEG	7-11
7.9	DEF USR	7-12
7.10	DIM	7-13
7.11	ERASE	7-14
7.12	ERR and ERL Variables	7-15
7.13	ERROR	7-16
7.14	INPUT	7-18
7.15	LET	7-20
7.16	LINE INPUT	7-21
7.17	LPRINT and LPRINT USING	7-23
7.18	NULL	7-24
7.19	OPTION BASE	7-25
7.20	OUT	7-26
7.21	POKE	7-27
7.22	PRINT	7-28
7.23	PRINT USING	7-33
7.24	RANDOMIZE	7-38

CONTENTS (continued)

7.24	READ	7-37
7.25	REM	7-39
7.26	RESTORE	7-41
7.27	SWAP	7-42
7.28	TIMES\$	7-43
7.29	WRITE	7-45

CHAPTER 8 DATA FILE INPUT AND OUTPUT

8.1	Disk Data Files - Sequential and Random I/O	8-1
	Sequential Files	8-1
	Random Files	8-4
	Protected Files	8-9
8.2	CLOSE	8-10
8.3	FIELD	8-11
8.4	GET	8-13
8.5	INPUT#	8-14
8.6	LINE INPUT#	8-15
8.7	LSET and RSET	8-16
8.8	OPEN (Files)	8-17
8.9	OPEN (Devices)	8-18
8.10	PRINT# and PRINT# USING	8-21
8.11	PUT	8-23
8.12	WRITE#	8-24

CHAPTER 9 FUNCTIONS

9.1	Introduction	9-1
9.2	Arithmetic Functions	9-1
	ABS	9-2
	ATN	9-3
	CDBL	9-4
	CINT(X)	9-5
	COS	9-6
	CSNG	9-7
	EXP	9-8
	FIX	9-9
	FRE	9-10
	INT	9-11
	LOG	9-12
	MKI\$, MKS\$, and MKD\$	9-13
	RND	9-14
	SGN	9-15
	SIN	9-16
	SQR	9-17
	TAN	9-18
9.3	String Functions	9-19
	ASC	9-19
	CHR\$	9-20

CONTENTS (continued)

	CVI, CVX, CVD	9-21
	HEX\$	9-22
	INKEY\$	9-23
	INPUT\$	9-24
	INSTR	9-25
	LEFT\$	9-26
	LEN	9-27
	MID\$	9-28
	OCT\$	9-29
	RIGHT\$	9-30
	SPACES\$	9-31
	STR\$	9-32
	STRING\$	9-33
	VAL	9-34
9.4	Input/Output and Special Functions.....	9-35
	EOF	9-35
	INP	9-36
	LOC	9-37
	LOF	9-38
	PEEK	9-39
	VARPTR	9-40
	VARPTR(#f)	9-42
9.5	Special Functions	9-43
	CSRLIN	9-43
	LPOS	9-44
	POINT	9-45
	POS	9-46
	SCREEN	9-47
	SPC	9-48
	TAB	9-49
	USR	9-50
CHAPTER 10	DEBUGGING	
10.1	Introduction	10-1
10.2	CONT	10-1
10.3	TROFF and TRON	10-3
CHAPTER 11	COLOR AND GRAPHICS	
11.1	Introduction	11-1
11.2	Color	11-2
	PALETTE	11-3
	COLOR	11-6
	PAINT	11-7
	POINT Function	11-8
	SCREEN Function	11-9

CONTENTS (continued)

11.3	Graphics Instructions	11-10
	CIRCLE	11-11
	LINE	11-12
	PSET	11-14
	PRESET	11-15
	DRAW	11-16
	GET and PUT	11-18
CHAPTER 12	SOUND INSTRUCTIONS	
12.1	Introduction	12-1
12.2	BEEP	12-1
12.3	SOUND	12-2
12.4	PLAY	12-3
APPENDICES		
Appendix A	ASCII Character Codes	A-1
Appendix B	Error Messages and Codes	B-1
Appendix C	Reserved Words	C-1
INDEX		INDEX-1

FIGURES

Figure 2-1	Main System Menu	2-2
Figure 2-2	Program Development Menu	2-3

TABLES

Table 1-1	Memory Requirements of Variables	1-5
Table 1-2	Arithmetic Operators	1-8
Table 1-3	Relational Operators	1-10
Table 1-4	Logical Operators	1-11
Table 2-1	Examples of BASIC Command Initialization	2-5
Table 3-1	Keys Used With GL in Entering BASIC Keywords	3-4
Table 3-2	Summary of Key-Controlled Editing Operations	3-7
Table 11-1	Video Options of the Wang PC System	11-1
Table 11-2	The Wang PC Color Palette	11-2
Table 11-3	Combination of Attributes in the Creation of Colors	11-5
Table A-1	Character Set	A-1
Table C-1	Results of Graphics Values	C-2

CHAPTER 1 OVERVIEW

1.1 INTRODUCTION

Wang Professional Computer (PC) Interpretive BASIC is an enhanced version of the BASIC language. Wang PC Interpretive BASIC includes instructions that facilitate such operations as numeric and string manipulation, output formatting, error trapping, and debugging. It also features assembly language subroutine calls, color and graphics instructions, sound and music generation capabilities, and the ability to read and write to any memory location. PC Interpretive BASIC supports variable length random and sequential files with a complete set of file manipulation statements. In addition, a full screen editor simplifies program development: you can perform most editing operations by simply moving the cursor on the screen and entering a few keystrokes.

1.2 INSTRUCTIONS

The BASIC language consists of a large group of instructions of various types, including statements and commands. Statements are programmable instructions used to write BASIC programs. In general, commands are instructions that work on programs. However, the general notion of the difference between commands and statements is largely undercut by PC Interpretive BASIC. In this implementation, you can enter and execute most commands and statements either outside or within a program. For example, you can use the LOAD, MERGE, and KILL commands within a program. On the other hand, you can use statements such as PRINT combined with functions like COS to perform quick calculations outside of a program.

Other BASIC instructions include operators and system-defined functions. Operators perform mathematical or logical operations on values. Functions perform specific mathematical, logical, or string operations. For example, the ABS function calculates the absolute value of either a value or an expression.

This chapter first discusses the values upon which BASIC operates, then discusses the BASIC arithmetic and logical operators.

1.3 CONSTANTS

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

1.3.1 String Constants

A string constant is a sequence of up to 255 alphanumeric characters enclosed in quotation marks. The following are examples of string constants:

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

1.3.2 Numeric Constants

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

- Integer constants are whole numbers between -32768 and +32767. Integer constants do not have decimal points.
- Fixed point constants are positive or negative real numbers, containing decimal points (e.g., 34.7, 508.923).
- Floating point constants are 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). Double-precision floating point constants used the letter D instead of E. The allowable range for floating point constants is 10^{-38} to 10^{+38} .

Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
```

- Hex constants are hexadecimal numbers prefixed by &H.

Examples:

```
&H76
&H32F
```

- Octal constants are octal numbers prefixed by &O or &.

Examples:

```
&O347
&I234
```

Single and Double-Precision Numeric Constants

Numeric constants can be either single-precision or double-precision numbers. Single-precision numeric constants are stored with 7 digits of precision, and printed with up to 6 digits. 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 any of the following qualities:

- seven or fewer digits
- exponential form using E
- a trailing exclamation point (!)

A double-precision constant is any numeric constant that has any of the following qualities:

- eight or more digits
- exponential form using D
- a trailing number sign (#)

Examples:

Single-Precision Constants

```
46.8
-1.09E-06
3489.0
22.5!
```

Double-Precision Constants

```
345692811
-1.09432D-06
3489.0#
7654321.1234
```

1.4 VARIABLES

Variables are names that represent values used in a BASIC program. Variables can represent either a numeric value or a string expression. You can assign the value of a value explicitly or it can be assigned as the result of calculations in a program. Before a string variable is assigned a value, its value is assumed to be all spaces. Similarly the value of a numeric variable is zero until another value is assigned to it.

1.4.1 Variable Names And Declaration Characters

PC Interpretive BASIC variable names can be up to 40 characters long. The first character must be a letter, but the remainder of the name can include letters, numbers, and decimal points.

If a variable begins with FN, BASIC considers it to be a call to a user-defined function. A BASIC reserved word cannot be a variable name, although it can be part of a variable name. For example, while the reserved word NAME would not be a legal variable name, BASIC would consider LASTNAME as a legal variable name even though a reserved word is embedded in it. variable name cannot be a reserved word. Reserved words include all BASIC commands, statements, function names, and operator names. (Refer to Appendix C for a list of BASIC reserved words.)

You can use type declaration characters at the end of a variable name to indicate whether the variable type is string, integer, single-precision, or double-precision.

String variable names are written with a dollar sign (\$) as the last character. Numeric variable names use the following type declaration characters:

%	Integer variable
!	Single-precision variable
#	Double-precision variable

If no type declaration character is used in a variable name, BASIC considers the variable to be single-precision numeric.

Examples:

<u>Variable Name</u>	<u>Action</u>
PI#	declares a double-precision value
MINIMUM!	declares a single-precision value
LIMIT%	declares an integer value
N\$	declares a string value
ABC	represents a single-precision value

The BASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL also declare the types for specific variable names. These statements are described in detail in Section 7.7.

1.4.2 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 an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

1.4.3 Memory Space Requirements

Table 1-1 lists the amount of memory used by each variable type.

Table 1-1. Memory Requirements of Variables

Variable		Amount of Memory
Scalar	String	3 bytes overhead plus 1 byte per character in the string
	Integer	2 bytes
	Single-Precision	4 bytes
	Double-Precision	8 bytes
Array	Strings	Per element: 3 bytes overhead plus 1 byte per character in the string
	Integer	2 bytes per element
	Single-Precision	4 bytes per element
	Double-Precision	8 bytes per element

1.4.4 Variable Type Conversion

Whenever necessary, BASIC converts a numeric constant from one type to another. Keep in mind the following rules and examples.

If a numeric constant is assigned to a numeric variable of another type, the value is converted to and stored as the type of the numeric variable.)

Example:

```

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

```

During expression evaluation, the result of an arithmetic or relational operation, as well as all operands, are converted to the degree of precision of the most precise operand.

Example 1:

```

Ok
10 D# = 6#/7
20 PRINT D#
RUN
.8571428571428571
Ok

```

Result:

The arithmetic was performed in double-precision and the result was returned in D# as a double-precision value.

Example 2:

```

Ok
10 D = 6#/7
20 PRINT D
RUN
.857143
Ok

```

Result:

The arithmetic was performed in double-precision and the result was returned to D (single-precision variable), rounded, and printed as a single-precision value.

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

Example:

```

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

```

If a single-precision value is assigned to double-precision variable, 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:

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

If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.

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. Refer to Subsection 1.5.5 for a discussion of logical operators.

1.5 OPERATORS AND EXPRESSIONS

An expression can be a single constant or variable. But more commonly, and expression combines constants and variables with operators. The result, in either case, is a single value.

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

- Arithmetic
- Relational
- Logical
- Functional

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 .

Examples:

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

1.5.1 Arithmetic Operators

Table 1-2 lists the BASIC arithmetic operators. When the operators appear in an arithmetic expression, BASIC performs the operations from left to right in the following order: exponentiation, multiplication or division, and addition or subtraction. To change the order in which the operations are performed, use parentheses. Two consecutive operators must be separated by parentheses. Operations within parentheses are performed first. However, inside the parentheses, the usual order of operations is maintained.

Table 1-2. Arithmetic Operators

Operator	Operation	Sample Expression
$\uparrow$	Exponentiation	$X \uparrow Y$
-	Negation	$-X$
*, /	Multiplication, Floating Point Division	$X * Y$ X / Y
+, -	Addition, Subtraction	$X + 1$ $X - Y$

The following are some sample arithmetic expressions and their BASIC counterparts.

<u>Algebraic Expression</u>	<u>BASIC Expression</u>
$X + 2Y$	$X + Y * 2$
$X - Y / Z$	$X - Y / Z$
$\frac{XY}{Z}$	$X * Y / Z$
$\frac{X + Y}{Z}$	$(X + Y) / Z$
$(X^2)^Y$	$(X \uparrow 2) \uparrow Y$
X^{YZ}	$X \uparrow (Y * Z)$
$X(-Y)$	$X * (-Y)$

1.5.2 Integer Division And Modulus Arithmetic

Two additional operators are available in PC Interpretive BASIC: Integer division and modulus arithmetic. Integer division is denoted by the backslash (\). Press the 2nd, ~~SHIFT~~ and / (Slash) keys to display the backslash on your screen. 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.

Examples:

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

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

Examples:

```
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.

1.5.3 Overflow and Division by Zero

If an expression attempts to divide by zero, the system displays a "Division by zero" error message, assigns machine infinity with the sign of the numerator to the result of the division, and continues execution.

If an expression attempts to raise zero to a negative power, the system displays a "Division by zero" error message, assigns positive machine infinity to the result, and continues execution.

If an evaluation of an expression results in overflow, the system displays an "Overflow" error message, assigns machine infinity with the algebraically correct sign to the result, and continues execution.

1.5.4 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 be used to make a decision regarding program flow. (Refer to the discussion of the IF statement in Section 6.8 for additional information.)

The set of relational operators are listed in Table 1-3. Note that the equal sign is also used to assign a value to a variable (refer to the discussion of the LET statement in Section 7.15).

Table 1-3. Relational Operators

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

1.5.5 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 Table 1-4. The logical operators are listed in order of precedence.

Table 1-4. Logical Operators

Operator	Examples of Logical Operation		
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. (Refer to the discussion of the IF statement in Section 6.9 for additional information.)

Example:

```

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

```

Logical operators convert 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).

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

<u>Example</u>	<u>Explanation</u>
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.

1.5.6 Functional Operators

An expression uses a function to call a predetermined operation to be performed on an operand. PC Interpretive BASIC contains intrinsic system functions, such as SQR (square root) or SIN (sine). Chapter 9 describes all BASIC system-defined functions. BASIC also allows the programmer to define functions. Refer to DEF FN, Section 7.6.

1.5.7 String Operations

You can compare strings 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#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"      where B$ = "8/12/78"
```

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.

The plus sign (+) concatenates strings.

Example:

```
Ok
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW" + A$+B$
RUN
FILENAME
NEW FILENAME
Ok
```


CHAPTER 2

ENTERING AND EXITING BASIC

2.1 INTRODUCTION

This chapter discusses the ways by which you can enter and exit PC Interpretive BASIC. It also describes how you can initialize the language to suit the disk and memory requirements of a particular program.

2.2 LOADING BASIC FROM THE SYSTEM SOFTWARE DISKETTE

BASIC is a file on the System Software diskette. You load BASIC from the System Software diskette by using any of the following Main System Menu selections.

- Program Development Menu
- DOS Command Processor
- Other

The particular method you use to load BASIC has no effect on its functionality. In fact, the result of each procedure is the same. These various methods for loading BASIC are provided so that users with various levels of experience with computers can easily load the language.

Use the following procedure to bring up the Main System Menu screen (refer to Figure 2-1).

1. Start your system with the System Software diskette correctly inserted in Drive A.
2. After the power-on diagnostic indicator lights are extinguished and the start-up screen indicates that the system has been brought up from Drive A, the Date and Time screen appears on the monitor. Either set the date and time or press the Execute key to proceed to the Main System Menu.

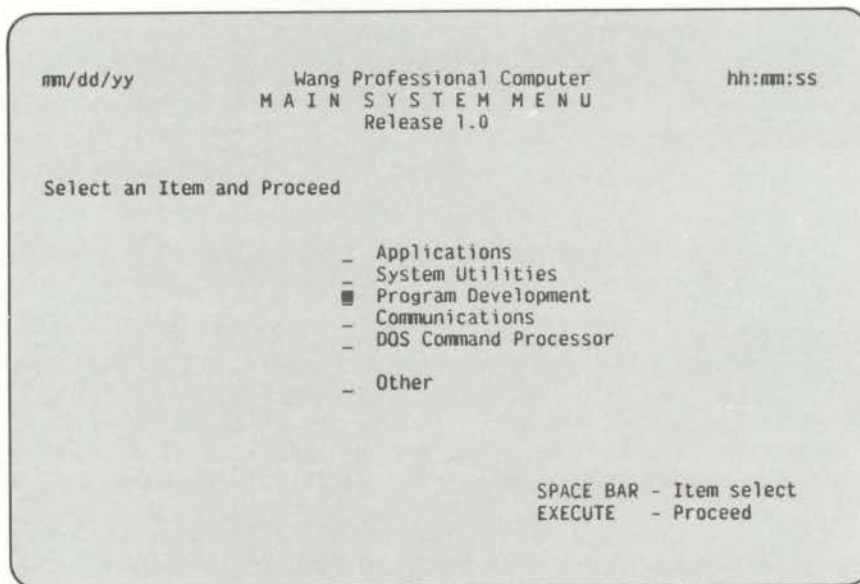


Figure 2-1 Main System Menu

2.2.1 Loading BASIC from the Program Development Menu

Use the following steps to load BASIC from the Program Development menu.

1. Move the acceptance block next to the Program Development menu selection on the Main System Menu and press the Execute key. The Program Development menu appears on the screen.
2. On the Program Development menu, move the acceptance block next to the BASIC selection, and press the Execute key (refer to Figure 2-2).

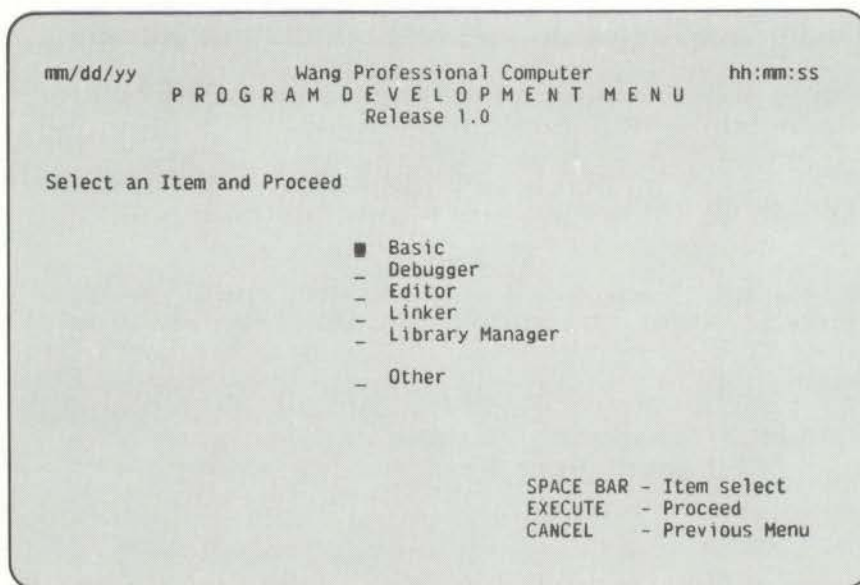


Figure 2-2 Program Development Menu

The BASIC display appears on the screen whenever you load BASIC. You can now begin to use the BASIC language instructions.

2.2.2 Loading BASIC from the DOS Command Processor

Use the following sequence of steps to load BASIC through the DOS Command Processor. This procedure allows you to load BASIC directly through the operating system.

1. Move the acceptance block next to the DOS Command Processor menu selection on the Main System Menu, and press the Execute key.
2. The DOS prompt and cursor are displayed under the operating system information. You are now in the operating system. In order to load BASIC, simply type the word BASIC after the operating system prompt.

The BASIC display appears on the screen whenever you load BASIC. You can now begin to use the BASIC language instructions.

2.2.3 Loading BASIC from the "Other" Selection

Use the following sequence of steps to load BASIC by using the "Other" method.

1. Move the acceptance block next to the "Other" entry on the Main System Menu, and press the Execute key.

2. The "Other" prompt and a cursor are displayed. In order to load BASIC, simply type the word BASIC on the line provided.

The BASIC display appears on the screen when you load BASIC. You can now begin to use the BASIC language instructions.

2.3 INITIALIZATION

You can use the BASIC command to instruct the DOS command processor how you want to initialize BASIC. The following is the format of the BASIC command.

Format:

```
A>BASIC [<filespec>] [/F:<number of files>]
      [/M:<highest memory location>] [/S:<maximum record size>]
```

where:

<number of files>, <highest memory location>, and <maximum record size> are numbers that can be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

If you include a <filespec>, BASIC proceeds as if a RUN <filespec> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied. This allows BASIC programs to be executed in batch mode using the BATCH facility of DOS. Such programs should include a SYSTEM statement to return to DOS when they have finished, allowing the next program in the batch stream to execute.

If you specify the /F:<number of files> parameter, it sets the number of disk data files that can be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is omitted, the number of files defaults to 3.

If you use the /M:<highest memory location> option it sets the highest memory location that will be used by BASIC.

If you add the /S:<maximum record size> at the end of the command line it sets the maximum record size you can use with random files. The default record size is 128 bytes.

Table 2-1. Examples of BASIC Command Initialization

Example	Action
A>BASIC PAYROLL.BAS	Use all memory and 3 files, load and execute PAYROLL.BAS.
A>BASIC INVENT/F:6	Use all memory and 6 files, load and execute INVENT.BAS.
A>BASIC /M:32768	Use first 32K of memory and 3 files.
A>BASIC DATAACK/F:2/M:&H9000	Use first 36K of memory, 2 files, and execute DATAACK.BAS.

2.4 EXITING BASIC

There are several methods you can use to exit BASIC. In general, it's safe to say that no matter which method you use to exit BASIC you will probably return to the level from which you loaded the language. So, if you loaded the language by using the Program Development menu, when you exit BASIC you should return to that menu. Similarly, if you loaded BASIC from the DOS command processor, when you exit the language you should return to the DOS command processor. Here again, the system is designed so that you can use a method that matches your experience level.

You can use any of the following methods to exit BASIC.

2.4.1 The SYSTEM Command

You should use the SYSTEM command to exit BASIC and return to the screen from whence you entered the language. Simply type SYSTEM at the cursor located under the BASIC prompt and press the Return key to execute this command.

CAUTION:

Be sure to save any valuable programs or files on disk before executing a SYSTEM command.

2.4.2 The Restart Procedure

There are several types of restart procedures. You use a restart procedure to start your system when the system won't let you proceed with your work.

You can restart your system by turning the power switch on the back panel of your electronics unit off and then on again. This type of restart is known as a cold restart. When you use the cold restart procedure, the system goes through the power-on diagnostics and displays the start-up screen. You then reset the date and time and reload BASIC. The cold restart procedure takes approximately 30 seconds.

To restart your system from the keyboard, press the 2nd + Command keys, and then press the Cancel key. This procedure is known as a warm restart and takes approximately 20 seconds. When you use the warm restart procedure, your system skips the power-on diagnostics. However, you must still reset the date and time, and reload BASIC.

CHAPTER 3

ENTERING AND EDITING BASIC TEXT

3.1 MODES OF OPERATION

After BASIC is loaded, the prompt "Ok" appears on the screen. "Ok" means BASIC is at command level. It is ready to accept commands. At this point, you can use BASIC in either of two modes: Direct mode or Indirect mode.

3.1.1 Direct Mode

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

Example:

```
Ok
PRINT 35/7
5
Ok
```

Any line of text typed while BASIC is in Direct mode will be processed by the full screen editor. BASIC is always in Direct mode after the prompt "Ok" and until a RUN command is given.

3.1.2 Indirect Mode

Indirect mode is the mode used for entering programs. Program lines are preceded by line numbers that are stored in main memory. The program stored in memory is executed when you enter a command such as the RUN command. For example:

```
Ok
10 PRINT 35/7
20 PRINT "FIVE"
RUN
5
FIVE
Ok
```

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

At any point while in Indirect mode, you can press the Shift and Cancel keys (or Ctl-C) to return to Direct mode, without saving any changes that were made to the current line.

3.2 PROGRAM LINES

Program lines in a BASIC program have the following format:

```
nnnnn BASIC instruction [:BASIC instruction...]
```

nnnn	line number
BASIC instruction	any valid BASIC statement

Example:

```
10 PRINT"475/(25/5)-95":INPUT "ANSWER";N$
```

A BASIC program line always begins with a line number and ends with a Return. An instruction line can contain a maximum of 255 characters, and can extend to more than three physical screen lines. In addition, a program line can contain more than one BASIC instruction.

3.2.1 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. They are also used as references when branching and editing. Line numbers must be in the range between 0 and 65529. A period (.) can be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

Examples:

```
LIST . (Lists the current line)
DELETE . (Deletes the current line)
```

3.2.2 Extending a Program Line to Another Physical Line

You can extend a logical line over more than one physical full screen using the line feed key sequence (Cntrl-J). Entering a line feed causes subsequent text to start on the next line without entering a carriage return. When a carriage return is finally entered, the entire logical line is passed to BASIC for storage in the program.

3.2.3 Placing More than One Instruction on a Line

You can place more than one BASIC instruction on a line by separating each instruction from the last by a colon. Some implementations of BASIC allow statements of the following form to set B and C equal to zero:

```
10 LET B=C=0
```

MS-BASIC, however, would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. To make this instruction work in MS-BASIC convert the statement to two assignment statements:

```
10 C=0:B=0
```

3.3 PROGRAM DEVELOPMENT FEATURES

BASIC provides several features that aid in program entry. These features permit the following operations:

- Clearing Old Program Text and Variables from Memory
- Automatic Line Numbering
- Renumbering Program Lines
- Easy Entry of Basic Keywords

3.3.1 Clearing Memory: NEW Command

The NEW command prepares for the entry of a new BASIC program by deleting the program currently in main memory and clearing all variables. BASIC always returns to command level after it executes a NEW command. Refer to Section 4.11 for a complete discussion of the NEW command.

CAUTION:

Be sure to save any valuable program text on disk before executing the NEW command.

3.3.2 Automatic Line Numbering: AUTO Command

The AUTO Command automatically generates a line number each time you press the Return key. You can specify both a starting line number and an increment for subsequently numbered lines. The default for both values is 10. Refer to Section 4.2 for additional information on the AUTO command.

3.3.3 Renumbering Program Lines: RENUM Command

The RENUM command renumbers program lines in memory. RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. You can specify a range of lines to be renumbered, as well as the increment between renumbered lines. The default increment is 10. Refer to Section 4.12 for a complete discussion of the RENUM command.

3.3.4 Entry of BASIC Keywords

A BASIC keyword is entered when the GL key is held down while one of the alphabetic keys A - Z is depressed. Keywords associated with each are summarized in Table 3-1. Keys not having reserved words are noted by ****.

Table 3-1. Keys Used with GL in Entering BASIC Keywords

Key	Keyword	Key	Keyword
A	AUTO	N	NEXT
B	BSAVE	O	OPEN
C	COLOR	P	PRINT
D	DELETE	Q	****
E	ELSE	R	RUN
F	FOR	S	SCREEN
G	GOTO	T	THEN
H	HEX\$	U	USING
I	INPUT	V	VAL
J	****	W	WIDTH
K	KEY	X	XOR
L	LOCATE	Y	****
M	MOTOR	Z	****

3.4 ENTERING PROGRAM LINES

When you press the Return key to conclude the creation of a program line, the system examines the line and processes it in one of the following ways.

- Adding a new line to the program in memory
- Changing a line currently in memory
- Deleting a line from the program in memory

3.4.1 Adding a New Line to a Program

A new line is added to the program in memory if all of the following conditions are met:

- The line number is between 0 and 65529.
- The line number does not match that of a line currently in memory.
- At least one non-blank character follows the line number.

If program memory is exhausted, and a line is added to the program, the error "Out of Memory" is displayed and the line is not added.

3.4.2 Replacing an Existing Line

An existing line is modified if its line number matches the line number of the currently entered line. The text of the newly entered line replaces that of the line previously existing in memory.

3.4.3 Deleting an Existing Line

An existing line is deleted if the line number of the currently entered line matches that of the existing line. The entered line must contain only a line number. If an attempt is made to delete a non-existent line, an "Undefined line number" error message is displayed.

3.5 BASIC EDITING FEATURES

BASIC provides several features that aid in program editing. These features permit the following operations:

- Recalling Program Text from Memory
- Recalling a Single Program Line from Memory
- Deleting a Line or Range of Lines from Memory

3.5.1 List Program Lines: LIST Command

The LIST Command lists all or part of the program currently in memory on the screen. The text can then be edited by means of the full screen editor. BASIC always returns to command level after it executes a LIST command. Refer to Section 4.8 for more information on the LIST command.

You can suspend program listing on a screen or printer by using Cntrl-S. Program listing is suspended at the currently displayed line and can be continued by Cntrl-Q.

3.5.2 Recalling a Single Program Line from Memory: EDIT Command

The EDIT statement displays a specified line and positions the cursor under the first digit of the line number. The line can then be edited by means of Edit mode subcommands. Refer to Section 4.5 for a complete discussion of the EDIT command.

3.5.3 Deleting a Line from Memory: DELETE Command

The DELETE Command deletes a specified line from memory. Refer to Section 4.4 for more information on the DELETE command.

3.6 ALTERING PROGRAM TEXT WITH THE EDITOR

You can modify any line existing on the screen, whether it is being typed in, or recalled to the screen by using the LIST command. A program line is not actually modified within the BASIC program until you press the Return key. Therefore, when several lines need alteration, it is sometimes easier to move around the screen making corrections to several lines at once, and then go back to the first line changed and press the Return key. It is not necessary to move the cursor to the end of the logical line before typing the carriage return. The Full Screen Editor remembers where each logical line ends and transfers the whole line even if the carriage return is typed at the beginning of the line.

You can editing any line that currently appears on the screen by moving the cursor on the screen to the place requiring change and then editing the text. You can alter text by simply typing over what appears on the screen. More advanced editing operations, as well as several control functions, can be performed by pressing the key or key combinations indicated in Table 3-2. A discussion of these editing operations follows.

Table 3-2. Summary of Key-Controlled Editing Operations

Editing Function	Key(s)	Control Code
Edit line where error occurred	Go To	Cntrl-A
Previous word	Prev	Cntrl-B
Stop program and return to BASIC command level	Shift + Cancel	Cntrl-C
Erase from Cursor to End of Logical Line	Erase	Cntrl-E
Next word	Next	Cntrl-F
Sound alarm		Cntrl-G
Destructive backspace	Bkspc	Cntrl-H
Move cursor to next tab stop	Tab	Cntrl-I
Line feed		Cntrl-J
Home cursor	Home	Cntrl-K
Clear screen and home cursor	PF 15	Cntrl-L
Pass current logical line to BASIC interpreter	Return	Cntrl-M
Append to end of line	Shift + Insert	Cntrl-N
Resume program listing on screen or printer		Cntrl-Q
Toggle insert mode	Insert	Cntrl-R
Suspend program listing on screen or printer		Cntrl-S
Activate and update function key display	Subscr	Cntrl-T
Clear logical line	Cancel	Cntrl-U
Delete word	Shift + Delete	Cntrl-W

Table 3-2. Summary of Key-Controlled Editing Operations (continued)

Editing Function	Key(s)	Control Code
Clear from cursor to end of screen	Shift + Erase	Cntrl-Z
Cursor right	East →	Cntrl-[
Cursor left	West ←	Cntrl-]
Cursor up	North ↑	Cntrl-6
Cursor down	South ↓	Cntrl- <u> </u>
Delete character at cursor location	Delete	Cntrl-?

3.6.1 Cursor Positioning and Tabbing

You can move the cursor to any position on the screen by using the → (Cntrl-[]), ← (Cntrl-[]), ↑ (Cntrl-6), ↓ (Cntrl-), and Home (Cntrl-K) keys. The Tab key (Cntrl-I) moves the cursor to the next tab stop. Tabs occur every eight columns.

Two other keys move the cursor but perform other functions as well. PF Key 15 (Cntrl-L) moves the cursor to the Home position (Row 1, Column 1) but also clears the screen. The Bkspc key (Cntrl-H) moves the cursor to the left, but deletes any character previously occupying that position.

You can also position the cursor at the beginning of the previous or next word by using the Prev (Cntrl-B) and Next (Cntrl-F) keys, respectively.

3.6.2 Deleting Characters and Words

To delete the character to the right of the cursor, press the Delete key (or Cntrl-?). All characters to the right of the deleted character are then moved one position to the left. If a logical line extends beyond one physical line, characters on subsequent lines are moved left one position, and the character in the first column of each subsequent line is moved up to the end of the preceding line.

To delete the character to the left of the cursor position, use the Bkspc key (or Cntrl-H). All characters to the right of the cursor are moved one position to the left. Subsequent characters and lines within the logical line are moved as described above.

To delete the word at the cursor position, press Cntrl-W (or the Shift and Delete keys).

3.6.3 Erasing Text

To erase from the cursor position to the end of the logical line, press the Erase key (or Cntrl-E). To clear the entire logical line, use the Cancel key (Cntrl-U).

To erase from the cursor position to the end of the screen, press the Shift and Erase keys (or Cntrl-Z). To clear the entire screen and home the cursor, press PF key '15 (or Cntrl-L).

3.6.4 Inserting Text

The Insert key (or Cntrl-R) switches between Text Entry mode and Insert mode. To insert characters at the cursor position, press the Insert key (or Cntrl-R) once. All subsequently typed characters are placed at the cursor position, and characters following the cursor are pushed to the right. As characters advance off the right side of the screen, they are inserted to the left of the subsequent line. When Insert (or Cntrl-R) is subsequently pressed, the editor returns to Text Entry mode.

You can append characters to the end of the line at which the cursor is positioned by using Cntrl-N (or Shift and Insert).

3.7 BASIC DEBUGGING FEATURES

BASIC provides several features that aid in program debugging. These features permit the following operations:

- Error Messages
- Listing Program Text on a Printer or Screen
- Stopping and Resuming Program Execution
- Tracing Program Execution

3.7.1 Error Messages

If BASIC detects an error that causes program execution to terminate, an error message is displayed. For a complete list of BASIC error codes and error messages, see Appendix B. A complete list of DOS error messages can be found in Appendix B of the Wang Professional Computer Introductory Guide.

When a Syntax Error is encountered during program execution, BASIC automatically enters Edit mode at the line that caused the error.

Example:

```
10 A = 2$12
RUN
?Syntax Error in 10
10 A = 2$12
```

The Full Screen Editor has displayed the line in error and positioned the cursor under the digit 1. The user moves the cursor right to the dollar sign (\$) and changes it to an up-arrow (↑), followed by a carriage return. The corrected line is now stored back in the program.

In this example, storing the line back in the program causes all variables to be lost. Had the programmer wanted to examine the contents of some variable before making the change, Cntrl-C would be typed to return to Direct Mode. The variables would be preserved since no program line was changed, and after the programmer was satisfied, the line could be edited and the program rerun.

3.7.2 Listing Program Text on a Screen or Printer: LIST and LLIST

The LIST command lists all or part of the program currently in memory at the terminal. The LLIST command lists all or part of the program currently in main memory at the printer, assuming a 132-character wide printer. BASIC always returns to command level after it executes a LIST or LLIST command. For complete information, refer to the discussions of the LIST and LLIST commands in Sections 4.8 and 4.9, respectively.

3.7.3 Stopping and Resuming Program Execution

At any point in program execution, you can terminate execution from the keyboard by pressing the Shift and Cancel keys (or Cntrl-C). This method suspends execution at the currently executing line. Similarly, you can use a STOP statement anywhere in a program to terminate execution (refer to the discussion of the STOP statement in Section 6.14).

When execution is stopped with either method, BASIC returns to command level. Intermediate values can be examined and changed using Direct mode statements. Then you may resume execution with CONT or a direct mode GOTO, which resumes execution at a specified line number. Refer to the discussions of CONT and GOTO in Sections 10.2 and 6.8, respectively.

3.7.4 Tracing Program Execution

The TRON statement provides a trace of the execution of program statements. 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). Refer to the discussion of TRON and TROFF in Section 10.2 for more detailed information.

3.8 LISTING THE DISK DIRECTORY

Within BASIC you can use the FILES command to list the directory of the currently selected drive. To display this directory, enter the FILES command when your system is at BASIC command level. Refer to Section 4.6 for a discussion of the FILES command.

If you want to work with the files on a disk you can move to the System Utilities menu and use the appropriate utilities. There are several system utilities that work with the disk directories. These utilities are described in Chapter 9 of the Wang Professional Computer Introductory Guide.

CHAPTER 4

PROGRAM ENTRY AND SCREEN MANIPULATION COMMANDS

4.1 INTRODUCTION

This chapter discusses the BASIC instructions that edit program text, list the contents of a program or a disk, designate and enable function keys, and set screen mode and width.

4.2 AUTO

(NC)

Function

To automatically generate a line number each time you press the Return key.

Format

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

line number	is the number at which the command begins to number. The legal range for line numbers is between 0 and 65529.
increment	is the quantity by which the subsequent line numbers increase in value.

Examples

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

Comments

1. The AUTO command begins numbering at line number and increments each subsequent line number by increment. The default for both values is 10. If you follow line number with a comma but do not specify an increment, then the increment you specified in the last AUTO command is used.

2. If AUTO generates a line number that is already in use, an asterisk is printed after the number to warn you that new input replaces the existing line. However, you can save the line and generate the next line number by pressing the Return key immediately after the asterisk.

3. You can terminate the AUTO command by typing Cntrl-C (or pressing the Shift and Cancel keys). BASIC returns to command level, but does not save the line at which you terminate the AUTO command.

Program Segment

```
NEW
Ok
AUTO 100, 10
100 INPUT "HOURS WORKED": HOURS
110 INPUT "HOURLY RATE": HRATE
120 GROSS = HOURS * HRATE
130 PRINT "GROSS PAY = $": GROSS
140 END
Press Cntrl-C
Ok
```

4.3 CLS

Function

To clear the screen to the background color.

Format

CLS

Comment

1. CLS also moves the cursor to the topmost logical left-hand position on the display.

4.4 DELETEFunction

To delete program lines.

Format

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

line number is any legal line number between 0 and 65529.

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)

Comments

1. BASIC always returns to command level after it executes a DELETE command.
2. If line number does not exist, an "Illegal function call" error occurs.

Program Segment

```
Ok
LIST
10 PRINT "*****"
20 PRINT "*"
30 PRINT "*"
40 PRINT "*"
50 PRINT "*"
60 PRINT "*"
70 PRINT "*"
80 PRINT "*"
90 PRINT "*"
100 PRINT "*****"
Ok
```

To delete lines 20 through 60 type:

DELETE 20-60

Ok

LIST

```
10 PRINT "*****"  
70 PRINT "*"      *"  
80 PRINT "*"      *"  
90 PRINT "*"      *"  
100 PRINT "*****"
```

Ok

4.5 EDIT

(NC)

Function

To enter Edit mode at the specified line.

Format

EDIT <line number>

line number is a legal line number within the program you intend to edit.

Comments

1. In Edit mode, you can edit portions of a line without retyping the entire line. BASIC displays the line number of the line to be edited, then waits for an Edit mode subcommand.

4.6 FILES

Function

To print the names of files residing on the current disk.

Format

FILES [<filespec>]

Examples

```
FILES
FILES "*.BAS"
FILES "B:*.*)"
FILES "TEST?.BAS"
```

Comments

1. If filespec is omitted, all the files on the currently selected drive will be listed. Filespec is a string that may contain question marks (?) to match any character in a filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

4.7 KEYFunction

To allow the Special Function keys to be designated "soft keys". Any one or all of twelve Special Function keys may be assigned a 15 byte string which, when the key is pressed, will be input to BASIC.

Initially, the soft keys are assigned the following values:

```
F1 - LIST
F2 - RUN
F3 - LOAD "
F4 - SAVE "
F5 - CONT
F6 - , " PRN
F7 - TRON
F8 - TROFF
F9 - KEY
F10 - SCREEN 0,0,0
F11 - WIDTH
F12 - CLS
```

Format

```
KEY <key number>,<string expression>
KEY LIST
KEY ON
KEY OFF
```

key number is an expression returning an unsigned integer in the range 1 to 10.

string expression is the key assignment text, any valid string expression.

Comments

1. The following is a description of the action of each of the formats:

KEY ON causes the key values to be displayed on the 25th Line. This is the initial setting. When the width is 40, 5 of the 12 soft keys are displayed. When the width is 80, all 12 are displayed. In either width, only the first 7 characters of each value are displayed.

KEY OFF erases the soft key display from the 25th line.

KEY LIST lists all 10 soft key values on the screen. All 15 characters of each value are displayed.

KEY <key number>,<string expression>

assigns the string expression to the soft key specified (1 to 10).

2. If the value returned for key number is not in the range 1 to 10, the system returns an "Illegal Function Call" error. The previous key string assignment is retained.
3. The key assignment string may be 1 to 15 characters in length. If the string is longer than 15 characters, the first 15 characters are assigned.
4. Assigning a null string (string of length 0) to a soft key disables the function key as a soft key.
5. When a soft key is assigned, the INKEY\$ function returns one character or the soft key string per invocation. If the soft key is disabled, INKEY\$ returns a string of length 2. The first characters is binary zero, the second is the Key Scan Code.
6. The SUBSCR key (Ctrl-T) also activates the function key display.

Examples

50 KEY ON (Display the soft keys on the 25th Line)

200 KEY OFF (Erase soft key display)

10 KEY 1,"MENU"+CHR\$(13) (Assigns the string 'MENU'<carriage return> to soft key 1. Such assignments might be used for rapid data entry. This example might be used in a program to select a menu display when entered by the user.)

20 KEY 1 "" (Would erase soft key 1)

Program Segment

The following routine initializes the first five soft keys:

```
Ok
1  KEY OFF : REM Turn off key display during init.
10 DATA key1,KEY2,KEY3,KEY4,KEY5
20 FOR I=1 TO 5:READ SOFTKEYS$(I)
30 KEY I,SOFTKEYS$(I)
40 NEXT I
50 KEY ON: REM now display new soft keys.
```


4.8 LIST

NC

Function

To list on the screen all or part of the program currently in main memory.

Format 1

LIST [<line number>]

line number is any line number between 0 and 65529.

Format 2

LIST [<line number>[-(<line number>)]]

line number is any line number between 0 and 65529.

Examples

Format 1:

LIST (Lists the program currently in memory)
LIST 500 (Lists Line 500)

Format 2:

LIST 150 (Lists all lines from 150 to the end)
LIST -1000 (Lists all lines from the lower number through 1000)
LIST 150-1000 (Lists Lines 150 through 1000, inclusive)

Comments

1. BASIC always returns to command level after it executes a LIST command.
2. In Format 1; if you omit line number, BASIC lists the program beginning at the lowest line number. Listing terminates when BASIC reaches the end of the program or when you type Cntrl-C. If you include line number, only the specified line is listed.
3. Format 2 has the following options:
 - a. If you specify only the first line number, BASIC lists the program from that line to the end of the listing.
 - b. If you specify only the second line number, BASIC lists all lines from the beginning of the program through the specified line number.
 - c. If you specify both numbers, the entire range is listed.

4.9 LLISTFunction

To list all or part of the program currently in main memory at the printer.

Format

LLIST [<line number>[-<line number>]]

line number is a line number between 0 - 65529

Examples

LLIST 40 - (prints the listing of program currently in memory starting from Line 40)

LLIST (prints all lines)

LLIST 100 (prints Line 100)

LLIST 50-80 (prints Lines 50 through 80 inclusive)

LLIST 250 - (prints Line 250 through the highest numbered line)

LLIST -250 (prints the lowest numbered line through line 250)

Comments

1. The LLIST command assumes a 132-character wide printer.
2. BASIC always returns to command level after it executes an LLIST command. The options for LLIST are the same as for the LIST command's Format 2.

4.10 LOCATE

Function

To move the cursor a position on the screen and turns the cursor on and off.

Format

LOCATE [<row>] [, [<column>] [, [<cursor>]]]

row	is a numeric expression returning an unsigned integer in the range 1 to 24. Row 25 is reserved for soft key display and cannot be overwritten even if the soft key display is turned off.
column	is a numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending upon the width of the screen
cursor	is a Boolean value indicating whether the cursor is visible or not. A value of 0 turns on the cursor; any other value turns off the cursor.

Comments

1. Any parameter can be omitted. If a parameter is omitted, the system assumes the current values of omitted parameters. Any values entered outside of the required ranges cause an "Illegal function call" error. In this event, the old values of the parameters are retained.
2. The rate of cursor blinking is 16 times per second and is not selectable.

Examples

LOCATE 1,1 (Moves the cursor to the home position in the upper left corner of the screen)

LOCATE ,,1 (Makes the cursor visible, leaving its position unchanged)

LOCATE 5,1,1 (Move the cursor to Row 5, Column 1 and make the cursor visible)

4.11 NEW

(NC)

Function

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

Format

NEW

Comment

1. Enter the NEW command to clear memory before entering a new program.
2. BASIC always returns to command level after it executes a NEW command.

Program Segment

```
Ok
5 REM the Questionnaire
10 Print "Welcome to our Company"
20 Print "We'd like you to answer a few questions."
30 Input "May we begin?"; A$
40 IF A$ = "YES" then input "What's your name?" N$
50 IF B$ = "NO" then input "Why do you object?" O$
```

```
RUN
Welcome to our Company
We'd like you to answer a few questions.
May we begin?
```

```
NEW
RUN
Ok
```

```
LIST
Ok
```

In this program segment you write a program and test it once by using the RUN command. Next you decide to redesign this program so you use the NEW command to clear your original program out of memory. The BASIC prompt returns after you try to run the program indicating that there is currently nothing in memory. The LIST command does not produce a program listing thereby confirming the fact that there is nothing in memory.

4.12 RENUM

(NC)

Function

To renumber program lines.

Format

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

new number	is the first line number you want in the new sequence. The default is 10.
old number	is the line in the current program where renumbering begins. The default is the first line of the program.
increment	is the step to use in the new sequence. The default is 10.

Examples

RENUM (Renumbers the entire program. The first new line number is 10. Lines increment by 10.)

RENUM 300,50 (Renumbers the entire program. The first new line number is 300. Lines increment by 50.)

RENUM 1000,900,20 (Renumbers the lines from 900 up so that they start with line number 1000 and increment by 20.)

Comments

1. RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers.
2. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyy" is displayed. The RENUM command does not change the incorrect line number reference xxxxx, but it can change line number yyyy.
3. Do not use the RENUM command to change the order of program lines.
4. You can not create line numbers greater than 65529. If you exceed this limit an "Illegal function call" error message appears.

4.13 RUN

Function

To execute the program currently in main memory.

Format 1

RUN [<line number>]

line number is a line number between 0 - 65529.

Examples

RUN
RUN 250

Comments

1. If you specify a <line number>, execution begins on that line. If you do not, execution begins at the lowest line number.
2. BASIC always returns to command level after it executes a RUN command.

4.14 SCREEN

Function

To set the screen mode.

Format

SCREEN [<mode>]

mode

is a valid numeric expression returning an unsigned integer value of 0, 1, and 2. Valid modes are the following:

- 0 - 300 x 800 pixel resolution (i.e., Wang Monochrome Monitor)
- 1 - 40-column (320 x 225 pixel resolution)
- 2 - 80-column (640 x 225 pixel resolution)

Examples

```
SCREEN 1
SCREEN 2
```

Comments

1. If the mode parameter is legal, the new screen mode is stored, the screen is erased, foreground color is set to white, and background and border colors are set to black.
2. Any value outside of the legal ranges results in an "Illegal function call" error.

4.14 WIDTH

Function

To set the number of characters that the terminal will display or that the printer will print on a line.

Format

```
WIDTH <size>
WIDTH <file-number>, <size>
WIDTH <device>, <size>
```

size is a valid numeric expression returning an integer result in the range 0 to 255.

file-number is a valid numeric expression returning an integer in the range 1 to 4. This is the number of a file opened to the printer (LPT1:).

Comments

1. The WIDTH statement establishes a limit for the number of characters that can appear in a printed line. You can use the WIDTH statement to establish the line width of the screen (SCRN:) or the printer (LPT1:). The WIDTH statement has no effect on the keyboard.
2. If you specify only the size parameter, the system assumes that the screen is the device. The device and file number parameters are optional and specify the printer as the device.
3. Any value entered for size outside of the required range results in an "Illegal function call" error. When this occurs, the current width is unchanged.

Screen

Valid screen (SCRN:) width can be either 40 and 80. Changing the screen width also clears the screen. In the event that an escape code sequence has changed the default screen to a screen that has only 40 columns, the width is automatically changed. The SCREEN statement also changes screen line width (refer to Section 4.14).

Examples:

```
WIDTH 80
WIDTH 40
```

Line Printer

Valid line size for printer (LPT1:) output is within the range 1 to 255. Specifying 255 as the size of a line disables line folding. Subsequently printed lines are infinitely wide. Maximum Epson printer line size is 80. However, the WIDTH statement accepts values between 80 and 255.

When "LPT1:" is specified as the device as in the statement, WIDTH "LPT1:", size, the specified size is stored as the width assignment for the printer. This form of the WIDTH statement stores the new width value without actually changing the current width setting. A subsequent OPEN "LPT1:" FOR OUTPUT AS file-number statement uses this stored value for line width while the file is open.

If a specified file-number is currently open to "LPT1:", the statement WIDTH file-number, size immediately changes the line width to the specified size. This allows the width to be changed at will while the file is open.

Example:

```
10 WIDTH "LPT1:", 75
20 OPEN "LPT:" FOR OUTPUT AS #1
6020 WIDTH #1, 40
```

Result:

Line 10 stores a printer width of 75 characters per line. Line 20 opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1 statements. Line 6020 changes the current printer width to 40 characters per line.

CHAPTER 5

TRANSFERRING PROGRAMS TO AND FROM DISK

5.1 INTRODUCTION

This chapter describes the commands you can use to manipulate programs. Since most of these commands involve transferring programs from one memory location to another, you need to understand the file naming conventions used by the language before you use these commands. BASIC's naming conventions are described in the following section.

5.1.1 File Naming Conventions

A BASIC file is composed of a three-part name called a file specification. A full file specification contains a drive designation, a filename, and a filename extension.

The format of a BASIC file specification is the following:

```
[<d>:]<filename>[.<ext>]
```

Each part of this file specification is described below.

d: is the drive designation. You can use this optional part of the file specification to specify which disk drive you want to use with a particular file. Diskette Drive A is the default drive. Drive B is the 2nd diskette drive, Drive C is the drive designation for the Winchester drive.

<filename> is a name that is 1 to 8 characters long. When you create filenames with less than 8 characters the unused positions in the filename are filled with spaces. The system performs this padding automatically.

You can use any of the following characters in any position in a filename.

A-Z	0-9	\$	&	#	@	!
%	'	(	)	-	<	>
		{	}			

Note that the system converts each lowercase character that you type to uppercase. Thus, if you type the filename "FiLe.ExT", the disk directory will show the filename as "FILE.EXT."

<ext>

is the optional filename extension; it can be up to three characters long. In all other respects, a filename extension follows the same conventions as a filename. You can specify a filename without an extension if you type a period after the filename. Otherwise, the operating system appends the default extension, .BAS to the filenames used in the SAVE, RUN, MERGE, or LOAD commands.

Device Filenames

Certain 3-letter filenames are reserved for the names of devices. These names are listed below:

<u>Name</u>	<u>Function</u>
AUX	Use when referring to input from or output to an auxiliary device.
CON	Use when referring to either keyboard input, or to output to the screen.
LST or PRN	Use when referring to the printer.
NUL	Use when you do not want to create a particular file, but the syntax of a command requires an input or output filename.

Even if given device designations or extensions, these filenames remain associated with the devices listed above. Thus, A:CON.LST still refers to the console, it is not the name of a disk file. This device naming scheme permits you to treat devices as if they were files.

5.2 KILL

Function

To delete a file from disk.

Format

KILL <"filespec">

filespec is any legal file specification as described in Section 5.1.

Examples

```
KILL "DATA1"
KILL "DATA1"
10 KILL "PAYROLL.BAS"
10 KILL "B:INFO"
```

Comments

1. The filespec must include the extension if one exists.
2. If you issue a KILL command for a file that is currently open, a "File already open" error occurs.
3. You can use the KILL command with each type of BASIC disk file: program files, random data files, and sequential data files.

Program Segment

The program in memory creates a data file, "TEMP". Prior to loading, "EXTEND", the program deletes "TEMP".

```
.
.
.
100 OPEN "TEMP" FOR INPUT AS #1
.
.
.
200 CLOSE #1
210 KILL "TEMP"
220 LOAD "EXTEND", R
```

5.3 LOAD

Function

To load a file from disk into main memory.

Format

LOAD <"filespec"> [,R]

filespec is the specification that was used when the file was saved. The operating system appends a default filename extension if one was not supplied in the SAVE command.

R indicates the Run option. Use the Run option with the LOAD command if you want to run a program after it's loaded. All open data files remain open when you use this command option to load a program.

Example

```
LOAD "INVEN",R
LOAD "MODIFY"
LOAD "B:CREATE.BAS"
```

Comments

1. Before it loads the designated program the LOAD command closes all open files and deletes all variables and program lines currently residing in memory.
2. You can chain several programs using the LOAD command with the Run option. Information can be passed between the programs using their disk data files.

Program Segment

The menu program below requests the user to choose an option number. It then branches to the appropriate LOAD statement.

Ok

```
10 PRINT "SELECT AN OPTION NUMBER"
20 PRINT TAB(10); "1 - CREATE FILE"
30 PRINT TAB(10); "2 - UPDATE FILE"
40 PRINT TAB(10); "3 - PRINT FILE"
50 PRINT TAB(10); "4 - DELETE FILE"
60 INPUT "OPTION:",NUMBER
70 ON NUMBER GOTO 100, 200, 300, 400
80 GOTO 60
100 LOAD "CREATE.BAS", R
200 LOAD "UPDATE.BAS", R
300 LOAD "PRINT.BAS", R
400 LOAD "DELETE.BAS", R
```

5.4 MERGE

(NC)

Function

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

Format

MERGE <"filespec">

filespec is the specification that was used when the file was saved.

Example

MERGE "NUMBERS"

Comments

1. The operating system appends the default filename extension, .BAS, if one was not supplied in the SAVE command.
2. The file merged into the program in memory must be in ASCII format. If it isn't, a "Bad file mode" error occurs.
3. Merging inserts the program lines on disk into the program in memory. If any lines in the disk file use the same line numbers as the program in memory, the lines from the file on disk will replace the corresponding lines in memory.
4. BASIC always returns to command level after executing a MERGE command.

Program Segment

The current file in memory is:

```
10 DIM NME$(20)
20 FOR K = 1 TO 20
30 INPUT "NAME:", NME(K)
40 NEXT K
400 PRINT "WHAT WAS YOUR NAME AGAIN?"
```

The file REPEAT was saved in ASCII format on the diskette mounted in Drive B.

```
400 PRINT "THIS IS VERY REPETITIOUS"
500 FOR C = 1 TO 20
510 PRINT "REPETITION"
520 NEXT C
530 END
```

The command to insert each instruction from the file REPEAT into the current file in memory is:

```
MERGE "B:REPEAT"
```

After executing the MERGE command, the current file in memory is:

```
10 DIM NME$(20)
20 FOR K = 1 TO 20
30 INPUT "NAME:", NME$(K)
40 NEXT K
400 PRINT "THIS IS VERY REPETITIOUS"
500 FOR C = 1 TO 20
520 NEXT C
530 END
```

Notice that Line 400 in the program in memory was replaced by line 400 in the REPEAT file by the MERGE command.

5.5 NAMEFunction

To change the name of a disk file.

Format

```
NAME <"old filespec"> AS <"new filespec">
```

old filespec	is file specification that exists on the current drive.
new filespec	is a file specification which doesn't exist on either drive.

Example

```
NAME "ACCTS" AS "LEDGER"  
NAME "LIST.BAS" AS "CALLS.BAS"  
NAME "SCHED." AS "TIMER.BAS"
```

Comments

1. After BASIC executes a NAME command, the old file exists on the specified disk with the new specification.
2. You must provide the exact file specification that you want the new file specification to have. In this case, BASIC does not assume the default file extension.

5.6 SAVEFunction

To save a program file on disk.

Format

SAVE <"filespec"> [,A | ,P])

filespec	is a specification that conforms to the operating system's requirements for file specifications. The operating system appends the default filename extension, .BAS if you don't supply one.
A	indicates the ASCII option. The ASCII option saves the file in ASCII format. Otherwise, BASIC saves the file in a compressed, binary format. ASCII format uses more space on the disk than the compressed, binary format.
P	indicates the Protect option. The Protect option protects the file by saving it in an encoded, binary format.

Examples

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

Comments

1. If you specify a file specification that already exists, the previous version of the file is written over.
2. Some disk access requires that files be in ASCII format. For instance, the MERGE command requires that the disk file be in ASCII.
3. You can't list or edit a protected file. In addition, protected files are not included on a disk's directory.

5.7 RUN

(C)

Function

To load a program from a disk and run it.

Format 2

RUN <"filespec"> [,R]

filespec is the file specification used when the file was saved. The operating system appends the default extension, .BAS if one was not supplied in the SAVE command.

R indicates the Run option. The Run option leaves all data files open and then runs the program.

Examples

RUN "NEWFILE",R
RUN "charges"

Comments

1. 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.
2. The BASIC Compiler supports the RUN and RUN line number forms of the RUN command.
3. The BASIC Compiler does not support the Run option with the RUN command. If you want to use this feature, then use the CHAIN statement. Refer to Section 6.3.

CHAPTER 6

TRANSFER OF PROGRAM CONTROL

6.1 INTRODUCTION

This chapter discusses those instructions that cause a transfer of control in a program. BASIC instructions allow the following transfers of control:

- Transfer of control to a routine or another program
- Termination or interruption of program execution
- Resuming program execution
- Repeated execution of a series of program lines
- Branching to and returning from a subroutine
- Unconditional branching
- Branching depending upon the results of a condition or a trap

6.2 CALL

Ⓒ

Function

To call an assembly language subroutine.

Format

CALL <variable name> [(<argument list>)]

variable name	contains the address that is the starting point in memory of the subroutine being called. Variable name may not be an array variable name.
argument list	contains the variables that are passed to the external subroutine. Items in the list are separated by commas. The argument list can contain only variables.

Example

CALL MYROUT (I,J,K)

Comments

1. You can transfer program flow to an external subroutine by using the CALL statement. (See also the USR function, Subsection 9.5.8.)
2. The CALL statement generates the same calling sequence used by the FORTRAN, COBOL, and BASIC compilers.
3. The CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. You should not use the user call, x=USR (n).
4. When a CALL statement is executed, control transfers to your routine via the segment address given in the last DEF SEG statement and the offset given in the variable name. Values are returned to BASIC by including the variable name that will receive the result in the argument list.
5. The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. BASIC follows the rules described for the MEDIUM case.
6. Invoking the CALL statement causes the following to occur:
 - a. For each parameter in the argument list, the 2-byte offset of the parameter's location within the Data Segment [DS] is pushed onto the stack.
 - b. BASIC's return address Code segment [CS] and offset [IP] are pushed onto the stack.
 - c. Control transfers to your routine via an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in <variable name>.

Program Segment

Example 1:

```

110 MYROUT=&H0000120
CALL MYROUT (I,J,K)
.
.
.
```

Do not use Line 110 within a BASIC Compiler program because the linking loader assigns the address of MYROUT at load time.

Example 2:

```
Ok
100 DEF SEG=&H8000
110 FOO=0
120 CALL FOO (A,B$,C0
    .
    .
    .
```

Line 100 sets the segment to 8000 Hex. FOO is set to zero so that the call to FOO executes the subroutine at location 80000H.

6.3 CHAIN

Function

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

Format

CHAIN [MERGE] <"filespec"> [, [<line number exp>] [, ALL] [, DELETE <range>]]

MERGE	is the MERGE command. Refer to Section 5.4 for a complete description.
filespec	is the file specification of the program that is called.
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 you omit ?line number exp?, program execution begins at the first line. A line number expression is not affected by a RENUM command.
ALL	signifies the ALL option. With the ALL option, every variable in the current program is passed to the called program.
DELETE	signifies the DELETE command. Refer to Section 4.4 for a full description of this command.
range	the range of line numbers you intend to delete. Any legal line numbers between 0 and 65529 are acceptable.

Example

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

Comments

1. If you omit the ALL option, the current program must contain a COMMON statement to list the variables that are passed. Refer to Section 7.3.

Example:

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

2. If you include the MERGE command option, then a subroutine can be brought into the BASIC program as an overlay. The MERGE command option 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:

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

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

Example:

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

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

5. The CHAIN statement with the MERGE command option leaves the files open and preserves the current OPTION BASE statement setting.

6. If you omit the MERGE command option, CHAIN does not preserve variable types or user-defined functions for use by the chained program. Any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN BASIC compiler statements containing shared variables must be restated in the chained program.

7. The BASIC compiler does not support the ALL, MERGE command, DELETE command, and, <line number exp> options to the CHAIN statement. Thus, the statement format is CHAIN <filename>. If you wish to maintain compatibility with the BASIC compiler, you should use the COMMON statement to pass variables; do not use overlays.

8. The CHAIN statement leaves the files open during chaining.

9. When you use the MERGE command option, place user-defined functions before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is completed.

Program Segment

This program calls NEWPGM and passes all of its variables to it. Execution begins at line 300 of NEWPGM.

```
.  
.
.
```

```
200 CHAIN "NEWPGM", 300, ALL
```

6.4 ENDFunction

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

Format

[END]

Example

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

Comments

1. You can place END statements anywhere in a program to terminate execution.
2. Unlike the STOP statement, END does not cause a break error message to be printed.
3. An END statement at the end of a program is optional.
4. BASIC always returns to command level after it executes an END statement.

6.5 FOR...NEXT

Function

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

Format

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

x, y, z are numeric expressions. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. Z is the increment used in each step.

variable is any legal variable name. The name can be up to 40 characters long.

Example

```
FOR X = 1 TO 10
.
.
.
NEXT X
```

Comments

1. The program lines following the FOR statement are executed until the NEXT statement is encountered.
2. 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 repeats. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.
3. 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.

4. The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

5. FOR...NEXT loops can be nested, that is, you can place a FOR...NEXT loop 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 can be used for all of them.

6. The variable(s) in the NEXT statement can be omitted, in which case the NEXT statement matches 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.

Program Segment 1

```
Ok
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
```

Program Segment 2

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

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

Program Segment 3:

```
Ok
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 for the loop variable is always set before the initial value is set.

6.6 GOSUB ... RETURN

Function

To branch to and return from a subroutine.

Format

GOSUB <line number>

.

.

.

RETURN

line number is the first line of a subroutine. Legal line numbers fall in the range between 0 and 65529.

Example

10 ON J GOSUB 200, 300, 300, 400

Comments

1. You can call a subroutine any number of times in a program. You can call a subroutine from within another subroutine. Such nesting of subroutines is limited only by your system's available memory.
2. A RETURN statement in a subroutine causes BASIC to branch back to the statement following the most recent GOSUB statement.
3. A subroutine can contain more than one RETURN statement, should logic dictate a return at different points in the subroutine.
4. You can use subroutines anywhere in a program, but the subroutine should be readily distinguishable from the main program.
5. You can precede a subroutine with a STOP, END, or GOTO statement that directs program control around the subroutine.

Program Segment 1

```
Ok
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
```

Program Segment 2

```
40 INPUT "ENTER CODE:", CODE
.
.
.
50 ON CODE GOTO 100, 200, 300
60 PRINT "PLEASE VERIFY CODE AND RE-ENTER": GOTO 40
.
.
.
100 REM lines to process when CODE = 1
.
.
200 REM lines to process when CODE = 2
.
.
300 REM lines to process when CODE = 3
.
.
.
```

6.7 GOTOFunction

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

Format

GOTO <line number>

line number is any legal line number between 0 and 65529.

Example

40 GOTO 500

Comment

1. If the line number is an executable statement, this statement and those that follow are executed.
2. If the line number is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Program Segment

```

Ok
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

```


6.8 IF ... THEN[... ELSE] and IF ... GOTOFunction

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

Format 1

```
IF <expression> [,] THEN <statement(s)> | <line number>
    [ELSE <statement(s)> | <line number>]
```

Format 2

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

expression	is any legal BASIC expression.
statement	is any BASIC statement.
line number	is any legal line number between 0 and 65529.

Examples

```
10 IF NUM RANGE THEN PRINT "OUT OF RANGE" ELSE PRINT "Acceptable"
10 IF .5 # NUVALUE = 10 GOTO 270
10 IF SALES = 1.0E+6 THEN 400 ELSE 600
```

Comments

1. If the expression is true (its value is not zero), the THEN or GOTO clause is executed.
 - a. THEN may be followed by either a line number for branching or one or more statements to be executed.
 - b. GOTO is always followed by a line number.
2. If the expression false (its result is zero), the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.
3. IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example the following is a legal statement:

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

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

Example:

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

Result:

"A<>C" is not printed when A<>B.

5. 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.

CAUTION:

When using the IF statement 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.

Program Segment 1

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

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

Program Segment 2

```
100 IF(I<20)*(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.

Program Segment 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 printer; otherwise, output goes to the screen.

6.9 ON ERROR GOTO

Function

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

Format

ON ERROR GOTO <line number>

line number is any legal line number between 0 and 65529.

Example

10 ON ERROR GOTO 1000

Comments

1. Once error trapping has been enabled, all errors detected, including direct mode errors (e.g., syntax errors), cause a jump to the specified error handling subroutine.
2. If line number does not exist, an "Undefined line" error results.
3. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution.
4. 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.
5. All error trapping subroutines should execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.
6. 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.

6.10 ON PEN/KEY/PEN/STRIGFunction

To enable event trapping and specify the first line of the event handling subroutine.

Format

ON <event specifier> GOSUB <line number>

event specifier can be any of the following:

COM (n) where n is the number of the COM channel (1 or 2).

KEY (n) where n is a Special Function key number (1-14). 1 through 10 are the soft keys SF 1 through SF 10. 11 through 14 are the cursor direction keys as follows: 11-Up, 12-Left, 13-Right, 14-Down. A KEY(0) ON, OFF, or STOP enables, disables, or stops all 14 key events.

PEN The Wang PC implementation supports PEN syntax.

STRIG(n) The Wang PC implementation supports STRIG syntax.

line number is any legal line number between 0 and 65529. Line number sets up the event trap line number for the specified event. A line number of 0 disables trapping for this event.

Comments

1. The following statements control the activation/deactivation of event trapping.

```
<event specifier> ON
<event specifier> OFF
<event specifier> STOP
```

2. If an event is ON, then every time BASIC starts a new statement, it checks to see if the specified event has occurred. If the event has occurred, it branches to the specified line-number, as long as the line number is not zero.

2. When an event is "stopped" no trapping can take place, but if the event happens this is remembered so an immediate trap will take place when an <event specifier> ON is executed.
3. When a trap is made for a particular event, the trap automatically causes a "stop" on that event so recursive traps can never take place. The "return" from the trap routine automatically does an ON unless an explicit OFF has been performed inside the trap routine.
4. When an error trap takes place this automatically disables all trapping.
5. Trapping will never take place when BASIC is not executing a program.
6. The following optional form of RETURN is primarily intended for use with event trapping. The event trap routine may want to go back into the BASIC program at a fixed line number while still eliminating the GOSUB entry that the trap created.

RETURN <line number>

Use of the non-local RETURN must be done with care. Any other GOSUB, WHILE, or FOR that was active at the time of the trap will remain active. If the trap comes out of a subroutine, any attempt to continue loops outside the subroutine will result in a "NEXT without FOR" error.

COM Trapping

Typically the COM trap routine will read an entire message from the COM port before returning back. It is not recommended to use the COM trap for single character messages since at high baud rates the overhead of trapping and reading for each individual character may allow the interrupt buffer for COM to overflow.

KEY and STRIG Trapping

The Wang PC supports the syntax of KEY and STRIG trapping for industry compatibility purposes.

6.11 ON...GOSUB and ON...GOTOFunction

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

Format

ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

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 is the destination of the branch. (If the value is a noninteger, the fractional portion is rounded.)
------------	---

list of line numbers	is any list of legal line numbers separated by commas. Line numbers range from 0 to 65529.
----------------------	--

Example

```
100 ON L-1 GOTO 150,300,320,390
```

Comments

1. In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.
2. 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.
3. If the value of expression is negative or greater than 255, an "Illegal function call" error occurs.

6.12 RESUMEFunction

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

Format 1

RESUME 0

Format 2

RESUME NEXT

Format 3

RESUME <line number>

line number any legal line number between 0 and 65529

Examples

RESUME or
RESUME 0

Execution resumes at the statement that caused the error.

RESUME NEXT

Execution resumes at the statement immediately following the one that caused the error.

RESUME <line number>

Execution resumes at <line number>.

Comment

1. A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Program Segment

Ok

10 ON ERROR GOTO 900

.

.

.

900 IF (ERR=230)AND(ERL=90)THEN PRINT "TRY AGAIN":RESUME 80

.

.

.

6.13 STOPFunction

To terminate program execution and return to command level.

Format

STOP

Example

5000 STOP

Comments

1. You use STOP statements anywhere in a program to terminate execution. When a STOP statement is encountered, the following message is printed:

Break in line nnnnn

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

3. BASIC always returns to command level after executing a STOP statement. Execution is resumed by issuing a CONT command (Refer to Section 10.1).

Program Segment

```
Ok
10 INPUT A,B,C
20 K=A↑2*5.3:L=B↑3/.26
30 STOP
40 M=C*K+100:PRINT M
```

Result:

? 1,2,3

```
Ok
BREAK IN 30
PRINT L
30.7692
```


To Resume Program Execution:

Ok
CONT
115.9
Ok

6.14 WAITFunction

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

Format

WAIT <port number>, I[,J]

I and J are integer expressions.

Example

100 WAIT 32,2

Comments

1. 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 you omit J, it is assumed to be zero.

CAUTION:

It is possible to enter an infinite loop with the WAIT statement, in which case you should restart the machine.

6.15 WHILE...WENDFunction

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

Format

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

expression is any valid BASIC expression.

Comments

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

2. WHILE/WEND loops can be nested to any level. Each WEND matches 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.

Program Segment

```
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
.
.
.
```


CHAPTER 7

DATA INPUT/OUTPUT INSTRUCTIONS

7.1 INTRODUCTION

BASIC provides a wide assortment of instructions that define and assign values to variables. This chapter discusses the means by which data is input to variables and the ways in which printed output is formatted.

7.2 CLEAR

(C)

Function

To set all numeric variables to zero, to set all string variables to null, to close all open files, and optionally, to set the end of memory and the amount of stack space.

Format

```
CLEAR [,]<expression1>][,<expression2>]]
```

expression1	is a memory location which, if specified, sets the highest location available for BASIC to use.
expression2	sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

Examples

```
10 CLEAR
10 CLEAR ,32768
10 CLEAR ,,2200
10 CLEAR ,32768,2000
```

Comments

1. BASIC allocates string space dynamically. An "Out of string space error" occurs only if there is no free memory left for BASIC to use.

2. The BASIC compiler supports the CLEAR statement with the restriction that <expression1> and <expression2> must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 256 bytes, and the default top of memory is the current top of memory. The CLEAR statement performs the following actions:

- Closes all files
- Clears all of the COMMON statement and user variables
- Resets the stack and string space
- Releases all disk buffers

7.3 COMMON

(C)

Function

To pass variables to a chained program.

Format

```
COMMON <variable> [, <variable>, ...]
```

variable is any legal variable name. Names can be up to 40 characters long.

Examples

```
10 COMMON A, B, R, TYPE$
10 COMMON NUM1, NUM12, account$, D()
```

Comments

1. Use the COMMON statement in conjunction with the CHAIN statement.
2. COMMON statements can appear anywhere in a program, but the preferred location is at the beginning.
3. The same variable cannot appear in more than one COMMON statement.
4. Specify array variables by appending "()" to the variable name.
5. If you are passing all of the variables, use the CHAIN statement with the All option and omit the COMMON statement.
6. The BASIC compiler supports a modified version of the COMMON statement. The COMMON statement must appear in a program before any executable statements. The nonexecutable statements are:

```
COMMON
DEFDBL, DEFINT, DEFSNG, DEFSTR
DIM
OPTION BASE
REM
%INCLUDE
```

7. Arrays in the COMMON statement must be declared in preceding DIM statements.

8. The standard form of the COMMON statement is referred to as blank COMMON. FORTRAN style named COMMON areas are also supported; however, the variables are not preserved across chains. The syntax for named COMMON is as follows:

```
COMMON /<name>/ <variable> [,<variable>,...]
```

<name> = 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with FORTRAN and assembly language routines without having to explicitly pass parameters in a CALL statement.

9. The blank COMMON size and order of variables must be the same in the chaining and chained-to programs. With the MS-BASIC compiler, the best way to ensure this is to place all blank declarations in a single include file and use the %INCLUDE statement in each program.

Example:

```
MENU.BAS
10 %INCLUDE COMDEF
.
.
.
1000 CHAIN "PROG1"

PROG1.BAS
10 %INCLUDE COMDEF
.
.
2000 CHAIN "MENU"

COMDEF.BAS
100 DIM A (100), B$(200)
110 COMMON I,J,K,A,()
120 COMMON A$,B$,(),X,Y,Z
```

Program Segment

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

7.4 DATA

Function

To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section 7.24.)

Format

DATA <list of constants>

list of constants	can contain numeric constants in any format, i.e., fixed point, floating point, or integer. (No numeric expressions are allowed in the list.)
-------------------	---

Examples

```
80 DATA 495, Route, 128, commuters, 119
80 DATA 6.55, 40, "Stevens, Arrow"
```

Comments

1. 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.
2. DATA statements are nonexecutable. You can place DATA statements anywhere in the program.
3. A DATA statement can contain as many constants as can fit on a line (separated by commas). Any number of DATA statements can be used in a program.
4. The READ statements access the DATA statements in order (by line number). The data contained therein can 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.
5. The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.
6. DATA statements can be reread from the beginning by using the RESTORE statement (refer to Section 7.26).

Program Segment

```
.  
. .  
100 READ N  
110 FOR K = 1 TO N  
120 READ HRATE(K), NME$(K)  
130 NEXT K  
140 DATA 4  
159 DATA 6.55, 40, "STEVENS, ARROW  
160 DATA 4.25, 35, "BLACK, ANNETTE"  
170 DATA 3.75, 45, "MACDONALD, CHRIS"  
. .  
. .  
. .
```


7.5 DATE\$

Function

The DATE\$ statement can be used to set or retrieve the current date.

Format 1 (To set the current date):

```
DATE$ = <"string expr">
```

Format 2 (To get the current date):

```
<"string expr"> = DATE$
```

"string expr" is a valid string literal or variable.

Example

```
10 DATE$ = "01-01-81"
```

Comments

1. The current date is fetched and assigned to the string variable if DATE\$ is the expression in a LET or PRINT statement.
2. The current date is stored if DATE\$ is the target of a string assignment.
3. If <string expr> is not a valid string, a "Type mismatch" Error will result. Previous values are retained.
4. For <string var>+DATE\$, DATE\$ returns a 10 character string in the form "mm-dd-yyyy" where mm is the month (01 to 12), dd is the day (01 to 31) and yy is the year (1980 to 2099).
5. For DATE\$ = <string expr>, <string expr> must be one of the following forms:
 - o "mm-dd-yy"
 - o "mm/dd/yy"
 - o "mm-dd-yyyy"
 - o "mm/dd/yyyy"

If any of the values are out of range or missing, an "Illegal function call" error is issued. Any previous date is retained.

7.6 DEF FNFunction

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

Format

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

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 separated by commas.
function definition	is an expression that performs the operation of the function. It is limited to one line.

Example

```
40 DEF FNAB (X,Y)= X↑3 / Y↑2
```

Comments

1. 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.
2. 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.
3. 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.
4. 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.
5. DEF FN is illegal in the direct mode.

Program Segment

```
Ok
10 DEF FNA (X,Y) = (X+Y) / 2
20 C = 100
30 D = 200
40 Z = FNA(C,D)
50 PRINT Z
RUN
150
Ok
```

7.7 DEFINT/SNG/DBL/STRFunction

To declare variable types as integer, single-precision, double-precision, or string.

Format

DEF<type> <range(s) of letters>

type is INT, SNG, DBL, or STR

range of letters is any letter or group of letters from A - Z.

Examples

10 DEFDBL L-P (All variables beginning with the letters L, M, N, O, and P are double-precision variables.)

10 DEFSTR A (All variables beginning with the letter A are string variables.)

10 DEFINT I-N,W-Z (All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z are integer variables.)

Comments

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

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

Program Segment

```
Ok
10 DEFDBL A-C
20 DEFINT E-G
30 DEFSTR D, H Z
40 ESTIMATE = 581/6
50 AVERAGE = 581#/6
60 NME = "WANG"
70 PRINT ESTIMATE, AVERAGE, NME
RUN
97 96.83333333333334 WANG
Ok
```

7.8 DEF SEG

Function

To assign the current value to be used by a subsequent CALL or user defined function call.

Format

DEF SEG [= <address>]

address is a valid numeric expression returning an unsigned integer in the range 0 to 65535.

Examples

```
100 DEF SEG=&HB800 (Set segment to Screen buffer)
20  DEF SEG (Restores segment to BASIC's data segment)
```

Comments

1. DEF and SEG must be separated by a space. Otherwise, BASIC would interpret the statement DEFSEG=100 to mean, "assign the value 100 to the variable DEFSEG."
2. The address specified is saved for use as the segment required by the PEEK, POKE, and CALL statements.
3. If you enter a value outside of the address range, an "Illegal Function Call" error is displayed. If this happens then the previous address is retained.
4. If you omit the address option, BASIC's Data Segment (DS), the initial default value, is used.
5. If you include an address option, it should be a value based upon a 16-byte boundary. For the PEEK, POKE, or CALL statements, the value is shifted left 4 bits to form the Code Segment address for the subsequent CALL instruction. BASIC does not perform additional checking to assure that the resulting segment and offset value are valid.

7.9 DEF USRFunction

To specify the starting address of an assembly language subroutine.

Format

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

digit	is any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address you specify. If <digit> is omitted, DEF USR0 is assumed.
-------	---

integer expression	is the starting address of the USR routine. Refer to Subsection 9.5.8.
--------------------	--

Example

```
300 DEF USR0=25000
```

Comment

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

Program Segment

```

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

7.10 DIM

Function

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

Format

DIM <variable(subscript)> [, <variable(subscript)>, ...]

variable is any legal variable name. Names can be up to 40 characters long.

subscript sets the maximum value of the array variable.

Examples

```
10 DIM A(15)
10 DIM SCORE(75), NME$(75), CODE(75)
```

Comments

1. 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 otherwise specified with the OPTION BASE statement (see Section 7.18).
2. The DIM statement sets all of the elements of the specified arrays to an initial value of zero.

Program Segment

```
Ok
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
```

```
·
·
·
```

7.11 ERASE

NC

Function

To eliminate arrays from a program.

Format

ERASE <arrayvariable> [, <arrayvariable>, ...]

arrayvariable is any legal array variable name.

Examples

```
40 ERASE A, B, C, T$  
40 ERASE NUM, TYPE$
```

Comments

1. After an array is erased you can either redimension the array or use the previously allotted array space for other purposes. If you attempt to redimension an array without erasing it first, a "Redimensioned array" error occurs.

Program Segment

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

7.12 ERR and ERL Variables

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

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 mode statement, use IF 65535 = ERL THEN...

Otherwise, use the following:

```
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. The BASIC error codes are listed in Appendix B.

7.13 ERRORFunction

To simulate the occurrence of a BASIC error; or to allow error codes to be defined by the user.

Format

ERROR <integer expression>

integer expression must be greater than 0 and less than 255.

Example

50 ERROR 35

Comment

1. If the value of integer expression equals an error code already in use by BASIC (Refer to Appendix B), the ERROR statement simulates the occurrence of that error, and the corresponding error message is printed. Refer to Program Segment 1 below.

Program Segment 1

```
Ok
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
String too long
Ok
```


Program Segment 2

```
.  
. 110 ON ERROR GOTO 400  
120 INPUT "WHAT IS YOU 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  
.  
.  
.
```

7.14 INPUT

Function

To allow input from the terminal during program execution.

Format

INPUT[;][<prompt string>;]<list of variables>

prompt string is a question or prompt.

list of variables consists of any legal variable type or types.
Separate variables with commas.

Examples

```
10 INPUT AGE
10 INPUT "WHAT IS YOUR AGE"; AGE
10 INPUT: "NAME:", NME$
```

Comments

1. The data that is entered is assigned to the variable(s) given in <list of variables>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.
2. The variable names in the list can 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.
3. When the program encounters an INPUT statement, program execution pauses and a question mark is displayed to indicate that the program is waiting for data. If prompt string is included, the string is displayed before the question mark. A user can then enter the required data at the terminal.
4. You can use a comma instead of a semicolon after the prompt string to suppress the question mark. For example, this statement displays the prompt INPUT "ENTER BIRTHDATE",B\$ with no question mark.
5. If INPUT is immediately followed by a semicolon, then the carriage return that a user types to input data does not echo a carriage return/line feed sequence.
6. If a user responds to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) it causes the message, "?Redo from start" to be printed. No assignment of input values is made until an acceptable set of responses is given.

Program Segment 1

```
Ok
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
```

Program Segment 2

```
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?
```

7.15 LETFunction

To assign the value of an expression to a variable.

Format

[LET] <variable>=<expression>

variable is any legal variable name. Names can be up to 40 characters long.

expression is any legal BASIC expression.

Examples

```
30 LET A = 17
30 B = 12
30 LET NUM = 256
30 LET NME$ = "ANNETTE"
30 AVG = (NUM1 + NUM2 + NUM3)/3
```

Comment

1. The word LET is optional. The equal sign is sufficient when you assign an expression to a variable name.

Program Segment

```
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
```

.

.

.

7.16 LINE INPUT

Function

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

Format

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

prompt string is any input string up to 254 characters. The prompt string is a string literal that is displayed on the screen before input is accepted.

string variable is any legal string variable name. Names can be up to 40 characters long.

Example

```
10 LINE INPUT "ENTER LAST NAME, FIRST NAME": NME$
```

Comments

1. 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>.

However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; the carriage return is ignored, the line feed is put into <string variable>, and data input continues.

2. 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 screen.

3. Type Cntrl-C to escape from a LINE INPUT prompt. BASIC returns to command level and displays the prompt "Ok". Entering CONT resumes execution at the LINE INPUT.

Program Segment

```
Ok
10 DIM NME$(5)
20 FOR C = 1 TO 5
30 LINE INPUT "ENTER LAST NAME, FIRST NAME"; NME$(C)
40 NEXT C
50 FOR J = 1 TO 5
60 PRINT "NAME"; J; NME$(J)
70 NEXT J
80 END
Ok
```

7.17 LPRINT and LPRINT USING

Function

To print data at the printer.

Format

LPRINT [<list of expressions>]

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

list of expressions is any legal expression.

string exp is any legal string expression.

Examples

```
100 LPRINT "ACCOUNT NUMBER"; ACCT
100 LPRINT USING "##.##"; 35.9, 18.165, 5.2
```

Comments

1. These statements perform the same functions as the PRINT and PRINT USING statements except that the output goes to the printer. See Section 7.21 and Section 7.22.
2. LPRINT assumes a 132-character-wide printer.

Program Segment

```
Ok
10 FOR C = 1 TO 10
20 LPRINT C
30 NEXT C
RUN
Ok
```

This program prints the numbers 1 to 10 on separate lines at the printer.

7.18 NULLFunction

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

Format

NULL <integer expression>

Comment

1. For 10-character-per-second tape punches, <integer expression> should be greater than or equal to 3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletypes and Teletype-compatible terminal screens. For 30 cps hard copy printers <integer expression> should be set at 2 or 3. The default value is 0.

Program Segment

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

Two null characters are printed at the end of each line.

7.19 OPTION BASE

Function

To declare the minimum value for array subscripts.

Format

OPTION BASE <n>

n is 1 or 0.

Example

10 OPTION BASE 1

Comment

1. The default base is 0. If the statement is executed, the lowest value an array subscript can have is one.

7.20 OUTFunction

To send a byte to a machine output port.

Format

OUT I,J

where:

I is the port number. I must be an integer expression in the range from 0 to 255.

J is the data to be transmitted. J must be an integer expression in the range 0 to 255.

Example

100 OUT 32,100

7.21 POKEFunction

To write a byte into a memory location.

Format

POKE I,J

I is the address of the memory location to be poked.
I must be in the range 0 - 65536.

J is the data to be poked. J must be in the range of
0 to 255.

Examples

```
10 POKE 104.0
10 POKE &H5A00,&HFF
```

Comments

1. The complementary function to POKE is PEEK. The argument to the PEEK statement is an address from which a byte will be read. See Subsection 9.4.5.
2. The POKE and PEEK statements are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

CAUTION:

The address you poke is not checked by BASIC. Do not poke addresses in your current program, or in a BASIC variable, or stack.

7.22 PRINT

Function

To display data on the screen.

Format

PRINT [<list of expressions>]

list of expressions includes any legal BASIC expression.
Expressions can be up to 255 characters long.

Examples

```
40 PRINT A
40 PRINT M$
40 PRINT "TRAVEL SCHEDULE 1982"
```

Comments

1. If you omit <list of expressions>, a blank line is displayed. If you include <list of expressions>, the values of the expressions are displayed on the screen.
2. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)
3. The position of each displayed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into display zones of 14 spaces each.
 - a. A comma causes the next value to be displayed at the beginning of the next zone.

Example:

```
Ok
10 PRINT "GOOD", "MORNING"
RUN
GOOD          MORNING
Ok
```

b. A semicolon causes the next value to be displayed immediately after the last value.

Example:

```
Ok
10 PRINT "GOOD"; "MORNING"
RUN
GOODMORNING
Ok
```

c. Typing one or more spaces between expressions has the same effect as typing a semicolon.

Example:

```
Ok
10 PRINT "GOOD" "MORNING"
RUN
GOODMORNING
Ok
```

4. If a comma or a semicolon terminates the list of expressions, the next PRINT statement appears on the same line, spaced accordingly.

Example:

```
10 PRINT "GOOD",
20 PRINT "MORNING"
RUN
GOOD          MORNING
Ok
```

5. If the list of expressions terminates without a comma or a semicolon, a carriage return is displayed at the end of the line.

Example:

```
10 PRINT "GOOD"
20 PRINT "MORNING"
RUN
GOOD
MORNING
OK
```

6. If the displayed line is longer than the terminal width, BASIC goes to the next physical line and continues the display.

Example:

```
10 FOR COUNT = 1 TO 10
20 PRINT COUNT,
30 NEXT COUNT
RUN
  1          2          3          4          5
  6          7          8          9         10
```

7. Displayed numbers are always followed by a space.

- a. Positive numbers are preceded by a space.

Example:

```
10 PRINT "EXPERIMENT"
20 PRINT 18; 19; "TWENTY"
RUN
EXPERIMENT
  18   19 TWENTY
Ok
```

- b. Negative numbers are preceded by a minus sign.

Example:

```
10 NUM = -10
20 PRINT "TESTING"
30 PRINT NUM; "OVER"
RUN
TESTING
-10 OVER
Ok
```

- c. Single-precision numbers that can be accurately represented with 6 or fewer digits in the unscaled or scaled format are output using the unscaled format.

Example:

```
10 PRINT 1E-7
20 PRINT 1E-8
RUN
.0000001
1E-08
Ok
```

d. Double-precision numbers that can be accurately represented with 16 or fewer digits in the unscaled or scaled format are output using the unscaled format. For example, 1D-15 is output as .0000000000000001 and 1D-16 is output as 1D-16.

Example:

```
10 PRINT 1D-16
20 PRINT 1D-17
RUN
.0000000000000001
1D-17
Ok
```

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

Example:

```
10 ? "HELLO"
RUN
HELLO
Ok
LIST
10 PRINT "HELLO"
Ok
```

Program Segment 1

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

In this program segment, the commas in the PRINT statement cause each value to be displayed at the beginning of the next display zone.

Program Segment 2

```

Ok
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 program statement, the semicolon at the end of line 20 causes both PRINT statements to be displayed on the same line. Line 40 causes a blank line to be displayed before the next prompt.

Program Segment 3

```

Ok
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 program statement, the semicolons in the PRINT statement cause each value to be displayed immediately after the preceding value. (Remember that a number is always followed by a space. Positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

7.23 PRINT USING

Function

To print strings or numbers using a specified format.

Format

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

string expressions	is a string literal (or variable) comprised of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers.
--------------------	---

list of expressions	is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons.
---------------------	--

Examples

PRINT USING " ":A\$,C\$

Comments

1. When PRINT USING is used to print strings, one of three formatting characters can be used to format the string expression.

<u>Character</u>	<u>Function</u>
"!"	Specifies that only the first character in the given string should be printed.
"\n spaces\"	Specifies that 2+n characters from the string should be printed. If the backslashes are typed with no spaces, two characters are printed; with one space, three characters are printed, and so on. If the string is longer than the expression, the extra characters are ignored. If the expression is longer than the string, the string is left-justified in the expression and padded with spaces on the right.
"&"	Specifies a variable length string expression. When the field is specified with "&", the string is printed as it is entered.

Example:

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

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

<u>Character</u>	<u>Function</u>
#	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 the positions specified, the number is right-justified (preceded by spaces) in the expression.

Examples:

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

```
10 PRINT USING "###.##" ;987.654
987.65
```

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

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

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

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

Example:

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

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

- A minus sign at the end of the expression causes negative numbers to be printed with a trailing minus sign.

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

Examples:

```
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.

Example:

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

****\$** The ****\$** at the beginning of an expression combines the effects of the above two symbols. Leading spaces are asterisk-filled and a dollar sign is printed before the number. ****\$** specifies three more digit positions, one of which is the dollar sign.

Example:

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

, A comma to the left of the decimal point in an expression 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 expression is printed as part of the expression. A comma specifies another digit position. The comma has no effect if used with the exponential (↑↑↑↑)

Examples:

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

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

Four up-arrows (↑↑↑↑) can 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 is used to the left of the decimal point to print a space or a minus sign.

Examples:

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

```
PRINT USING ".#####↑↑↑↑-";888888
.889E+06
```

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

An underscore in the expression causes the next character to be printed as a literal character.

Example:

```
PRINT USING "_!##.##_";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the expression.

%

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

Examples:

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

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

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

Program Segment

```
Ok
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  !!
Ok
```

7.24 RANDOMIZEFunction

To reseed the random number generator.

Format

RANDOMIZE [<expression>]

expression is any legal BASIC expression.

Examples

```
20 RANDOMIZE 197
20 RANDOMIZE
20 RANDOMIZE SEEDVALUE
```

Comments

1. If you omit <expression>, BASIC suspends program execution and asks you for a value by printing this message:

Random Number Seed (-32768 to 32767)?

2. If you do not reseed the random number generator, the RND function returns the same sequence of random numbers each time you run the program.

3. If you place a RANDOMIZE statement at the beginning of a program then you can change the argument every time you use the program to change the sequence of random numbers.

Program Segment

```
Ok
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)? 3 (user types 4 for new sequence)
.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
```

Data Input/Output Instructions

7.25 READ

Function

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

Format

```
READ <variable> [, <variable>, ...]
```

variable is any legal variable name. Names can be up to 40 characters long.

Examples:

```
60 READ A, B, ADDRESS  
60 READ NUMBER  
60 READ J(c), L(c), mt(c)
```

Comments

1. Always use a READ statement in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables can be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" result. (See DATA, Section 7.4.)
2. A single READ statement may access one or more DATA statements (they are accessed in order), or several READ statements may access the same DATA statement.
3. If the number of variables in a list of variables exceeds the number of elements in the DATA statement(s), an "Out of Data" message is printed.
4. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.
5. To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Section 7.26).

Program Segment 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) is 3.08, and so on.

Program Segment 2:

```

Ok
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.

7.26 REM

Function

To allow explanatory remarks to be inserted in a program.

Format 1

REM <remark>

Format 2

'<remark>

remark is any combination of up to 256 characters.

Examples

```
5 REM This program calculates the interest on loan
5 GOTO 400 'interest rate projections
```

Comments

1. REM statements are not executed but are displayed exactly as entered when the program is listed.
2. REM statements can be branched into from a GOTO or GOSUB statement. Program execution continues with the first executable statement after the REM statement.
3. Comments can be added to the end of a line by preceding the comment with a single quotation mark instead of :REM.

CAUTION:

Do not use the REM statement in a DATA statement since it would be considered legal data.

Program Segment 1

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

Program Segment 2

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

7.27 RESTORE

Function

To allow DATA statements to be reread from a specified line.

Format

RESTORE [<line number>]

line number is any legal line number between 0 and 65529.

Examples

```
130 RESTORE 100
130 RESTORE
```

Comment

1. 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.

Program Segment

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

7.28 SWAPFunction

To exchange the values of two variables.

Format

SWAP <variable>, <variable>

variable is any legal variable name. Names can be up to 40 characters long.

Examples

```
50 SWAP OLD$, NEW$
50 SWAP CUST$, TEMP$
```

Comment

1. You can swap any type of variable (integer, single-precision, double-precision, or string), but the two variables must be of the same type or a "Type mismatch" error results.

Program Segment

```
Ok
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
```

7.29 TIME\$Function

To set or retrieve the current time.

Format 1 (To set the current time):

TIME\$ = <string expr>

Format 2 (To get the current time):

<string expr> = TIME\$

string expr is a valid string literal or variable.

Comments

1. If <string expr> is not a valid string, a "Type mismatch" Error will result.
2. For <string var>" TIME\$, TIME\$ returns an 8 character string in the form "hh:mm:ss" where hh is the hour (00 to 23), mm is the minutes (00 to 59), and ss is the seconds (00 to 59).
3. For TIME\$ = <string expr>, <string expr> can be one of the following forms:
 - a. "hh" Set the hour. Minutes and seconds default to 00.
 - b. "hh:mm" Set the hour and minutes. Seconds default to 00.
 - c. "hh:mm:ss" Sets the hour, minutes, and seconds.

If any of the values are out of range, an "Illegal Function Call" error is issued. The previous time is retained.

Example:

```
10 TIME$ = "08:00"
```

4. The current time is fetched and assigned to the string variable if TIME\$ is the expression in a LET or PRINT statement.
5. The current time is stored if TIME\$ is the target of a string assignment.

Program Segment

The following program displays the current date and time on the 25th line of the screen and will "chime" on the hour.

Ok

```
10 KEY OFF:SCREEN 0"WIDTH 40: CLS
20 LOCATE 25,5
30 PRINT DATE$, ,TIME$
40 SEC = VAL(MID$(TIME$,7,2))
50 IF SEC = SSEC THEN 20 ELSE SSEC = SEC
60 IF SEC = 30 THEN 1020
70 IF SEC = 30 THEN 1020
80 IF SEC < 57 THEN 20
```

```
1000 SOUND 1000,2:GOTO 20
1010 SOUND 2000,8:GOTO 20
1020 SOUND 400,4 :GOTO 20
```


7.30 WRITE

Function

To display data on the screen.

Format

WRITE[<list of expressions>]

where:

list of expressions	contains numeric and/or string expressions; you must separate them with commas.
---------------------	---

Example

```
30 WRITE C, O, TO$
```

Comments

1. If you omit <list of expressions>, a blank line is displayed. If you include <list of expressions>, the values of the expressions are displayed on the screen.
2. When the displayed items appear, each item is separated from the last by a comma. Displayed strings are delimited by quotation marks. After the last item in the list is displayed, BASIC inserts a carriage return/line feed.
3. WRITE outputs numeric values using the same format as the PRINT statement, refer to Section 7.21.

Program Segment

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


CHAPTER 8

DATA FILE INPUT AND OUTPUT

8.1 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are three types of disk data files that you can create and access by a BASIC program.

- sequential files
- random access files
- protected files

8.1.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when you want to access data. The data that is written to a sequential file is a series of ASCII characters stored, in the order they are sent, one item after another. This data is read back from a sequential file in the same order in which it was stored.

Use the following statements and functions with sequential files:

OPEN	PRINT#	INPUT#	WRITE#
	PRINT# USING	LINE INPUT#	
CLOSE	EOF	LOC	

Program Steps

Follow the program steps listed below to create a sequential file and to access the data in the file:

- | | | |
|----|--|--------------------------------|
| 1. | OPEN the file in "O" mode | OPEN "O", #1, "DATA" |
| 2. | Write data to the file
using the PRINT# statement.
(You can also use WRITE#.) | PRINT#1,A\$;B\$;C\$ |
| 3. | To access the data in the
file, you must close the file
and reopen it in "I" mode. | CLOSE #1
OPEN "I",#1,"DATA" |

4. Use the INPUT# statement to read data from the sequential file into the program. INPUT#1,X\$,Y\$,Z\$

Program 1 - Creating a Sequential Data File

Program 1 is a short program that creates a sequential file, "DATA", from information that you enter at the keyboard.

```
Ok
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;"",D$;"",H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

Program 2 - Accessing a Sequential File

Now look at Program 2 (below). It accesses the file "DATA" that you created in Program 1 and displays the name of everyone hired in 1978.

```
Ok
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

Program 2 sequentially reads every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15, which uses the EOF function to test for end-of-file.

```
15 IF EOF(1) THEN END
```

In addition, change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the program can use the statement

```
PRINT#1,USING#####.##,";A,B,C,D
```

to write numeric data to disk without specifying explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that the program has written to or read from the file since it was opened. A sector is a 128-byte block of data.

Adding Data To A Sequential File

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES".

1. Open "NAMES" in "I" mode.
2. Open a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. Close "NAMES" and kill it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and close the file.

Now there is a file on disk called "NAMES" that includes all the original data plus the new data you just added.

Program 3 - Adding Data to a Sequential File

Program 3 below illustrates this technique. You can use it to create or add onto a file called NAMES. This program also illustrates the use of a LINE INPUT# statement to read strings with embedded commas from the disk file. Remember, LINE INPUT# reads in characters from the disk until it sees a carriage return or until it has read 255 characters (it does not stop at quotes or commas).


```

Ok
10      ON ERROR GOTO 2000
20      OPEN "I",#1,"NAMES"
30      REM IF FILE EXISTS, WRITE IT TO "COPY"
40      OPEN "O",#2,"COPY"
50      IF EOF(1) THEN 90
60      LINE INPUT#1,A$
70      PRINT#2,A$
80      GOTO 50
90      CLOSE #1
100     KILL "NAMES"
110     REM ADD NEW ENTRIES TO FILE
120     INPUT "NAME";N$
130     IF N$="" THEN 200 'CARRIAGE RETURN EXISTS INPUT LOOP
140     LINE INPUT "ADDRESS";A$
150     LINE INPUT "BIRTHDAY";B$
160     PRINT#2,N$
170     PRINT#2,A$
180     PRINT#2,B$
190     PRINT:GOTO 120
200     CLOSE

205     REM CHANGE FILENAME BACK TO "NAMES"
210     NAME "COPY" AS "NAMES"
2000    IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010    ON ERROR GOTO 0

```

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and the program creates "COPY" as though it was a new file.

8.1.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using them. One advantage is that random files require less room on the disk, because BASIC stores them in a packed, binary format. A sequential file is stored as a series of ASCII characters.

The biggest advantage in using random files is that you can access data randomly, anywhere on the disk. Unlike sequential files, you don't have to read through all the information in order to find a particular piece of data. This is possible because the information in a random file is stored and accessed in distinct units called records. Each record is numbered.

The statements and functions that you can use with random files are:

OPEN	FIELD	LSET/RSET	GET
PUT	CLOSE	LOC	
MKI\$	CVI		
MKS\$	CVS		
MKD\$	CVD		

Program Steps

The following program steps are required to create a random file.

1. Open the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.
OPEN "R",#1,"FILE",32
2. Use the FIELD statement to allocate space in the random buffer for the variables that you want to write to the random file.
FIELD #1, 20 AS N\$,
4 AS A\$, 8 AS P\$
3. Use LSET to move the data into the random buffer. You must make numeric values into strings when you place them in the buffer. Use the make functions: MKI\$ to make an integer value into a string, MKS\$ for a single-precision value, and MKD\$ for a double-precision value.
LSET N\$=X\$
LSET A\$=MKS\$ (AMT)
LSET P\$=TEL\$
4. Write the data from the buffer to the disk using the PUT statement.
PUT #1,CODE%

Program 4 - Create a Random File

Look at Program 4 below. It takes information that is entered at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is typed in line 30 becomes the record number.

NOTE:

Do not use a fielded string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of into the random file buffer.

```

Ok
10  OPEN "R",#1,"FILE",32
20  FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
25  IF CODE%=9999 THEN END
30  INPUT "2-DIGIT CODE";CODE%
40  INPUT "NAME";X$
50  INPUT "AMOUNT";AMT
60  INPUT "PHONE";TEL$:PRINT
70  LSET N$=X$
80  LSET A$=MK$ (AMT)
90  LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30

```

Accessing A Random File

The following program steps are required to access a random file:

- | | | |
|----|---|--|
| 1. | Open the file in "R" mode. | OPEN "R",#1,"FILE",32 |
| 2. | Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file. | FIELD #1 20 AS N\$
4 AS A\$, 8 AS P\$ |

NOTE:

In a program that performs both input and output on the same random file, you can often use a single OPEN statement and a single FIELD statement.

- | | | |
|----|--|------------------------------|
| 3. | Use the GET statement to move the desired record into the random buffer. | GET #1,CODE% |
| 4. | The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the convert functions: CVI for integers, CVS for single-precision values, and CVD for double-precision values. | PRINT N\$
PRINT CVS (A\$) |

Program 5 - Accessing a Random File

Program 5 below accesses the random file "FILE" that was created in Program 4. By typing the three-digit code at the keyboard, the information associated with that code is read from the file and displayed.

```
Ok
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

The LOC function, when used with random files, returns the current record number. The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1) 50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program 6 - Inventory

Program 6 below is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed that the inventory contains no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR\$ (255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
Ok
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$, 2 AS R$, 4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILES"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION 1)OR(FUNCTION 6) THEN PRINT
      "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
```



```

250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$) = 255 THEN INPUT "OVERWRITE"; A$:
      IF A$ = "Y" THEN RETURN
280 LSET F$ = CHR$(0)
290 INPUT "DESCRIPTION"; DESC$
300 LSET D$ = DESC$
310 INPUT "QUANTITY IN STOCK"; Q%
320 LSET Q$ = MKI$(Q%)
330 INPUT ":REORDER LEVEL"; R%
340 LSET R$ = MKI$(R%)
350 INPUT "UNIT PRICE"; P
360 LSET P$ = MKS$(P)
370 PUT#1, PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY": RETURN
420 PRINT USING "PART NUMBER ###"; PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####"; CVI(Q$)
450 PRINT USING "REORDER LEVEL #####"; CVI(R$)
460 PRINT USING "UNIT PRICE $$$#.##"; CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY": RETURN
510 PRINT D$: INPUT "QUANTITY TO ADD "; A%
520 Q% = CVI(Q$) + A%
530 LSET Q$ = MKI$(Q%)
540 PUT#1, PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY": RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT"; S%
610 Q% = CVI(Q$)
620 IF (Q% - S%) < 0 THEN PRINT "ONLY"; Q%; " IN STOCK": GOTO 600
630 Q% = Q% - S%
640 IF Q% = CVI(R$) THEN PRINT "QUANTITY NOW"; Q%;
      " REORDER LEVEL"; CVI(R$)
650 LSET Q$ = MKI$(Q%)
660 PUT#1, PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I = 1 TO 100
710 GET#1, I
720 IF CVI(Q$) < CVI(R$) THEN PRINT D$; " QUANTITY";
      CVI(Q$) TAB(50) "REORDER LEVEL"; CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER"; PART%
850 IF (PART% < 1) OR (PART% > 100) THEN PRINT "BAD PART NUMBER":
      GOTO 840 ELSE GET#1, PART%: RETURN

```



```
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$ > "Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

8.1.3 Protected Files

If you wish to save a file or program in encoded binary format, use the Protect option of the SAVE command.

Example:

```
SAVE "MYPROG",P
```

A file saved in this way cannot be listed or edited. Protected files are not displayed when you list the disk directory either with the BASIC FILES command or with the operating system's DIR utility.

8.2 CLOSEFunction

To conclude I/O to a disk file.

Format

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

file number is the number under which the file was opened. A CLOSE statement with no arguments closes all open files.

Examples

100 CLOSE
100 CLOSE #1, #4, #6

Comments

1. The association between a particular file and file number terminates upon execution of a CLOSE statement. You can then reopen the file using the same or a different file number. Until you close a file the ?file number? may not be reused to open any other file.
2. In a sequential output file the CLOSE statement writes the final buffer of output.
3. The END statement and the NEW command always close all disk files automatically. STOP does not close disk files.

8.3 FIELD

Function

To allocate space for variables in a random file buffer.

Format

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

file number is the number you used to open the file.

field width is the number of characters you want to allocate to
 <string variable>.

string variable is any legal string variable name.

Example

FIELD #1, 20 AS N\$, 10 AS ID\$, 40 AS ADD\$

This statement 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\$. (Refer to the LSET/RSET and GET statements.)

Comments

1. You must execute a FIELD statement to get data out of a random buffer after a GET or to enter data before a PUT.
2. 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 128.
3. 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.

CAUTION:

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.

Program Segment

```
10 OPEN "R", #1, "DATA", 30
20 FIELD #1, 15 AS LAST$, 15 AS FIRST$
30 FOR J = 1 TO 10
40 INPUT "LAST NAME"; L$
50 INPUT "FIRST NAME"; F$
60 LSET LAST$ = L$
70 LSET FIRST$ = F$
80 PUT #1, J
90 NEXT J
100 CLOSE : END
```

8.4 GET

Function

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

Format

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

file number is the number under which the file was opened.

record number specifies the number of the record to be gotten. If
 <record number> is omitted, the next record (after
 the last GET) is read into the buffer. The largest
 possible record number is 32767.

Examples

```
50 GET E3, I
50 GET #2
```

Comment

1. After a GET statement, you can use the INPUT# and LINE INPUT# statements to read characters from the random file buffer.

Program Segment

```
Ok
10 OPEN "R", #1, "DATA", 30
20 FIELD #1, 15 AS LAST$, 15 AS FIRST$
30 FOR J = 1 TO 10
40 GET #1
50 PRINT FIRST$; " "; LAST$
60 NEXT J
70 END
```


8.5 INPUT#

Function

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

Format

INPUT#<file number>,<variable list>

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.

Comments

1. The variable type must match the type specified by the variable name. With the INPUT# statement, no question mark is printed as a user prompt.
2. 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.
3. 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.
 - a. If the first character is a quotation mark, the string item consists of all characters read between the first quotation mark and the second. Thus, a quoted string cannot contain a quotation mark as a character.
 - b. If the first character of the string is not a quotation mark, the string is an unquoted string and terminates with 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.

8.6 LINE INPUT#

Function

To read an entire line (up to 255 characters), without delimiters, from a sequential disk data file to a string variable.

Format

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

file number is the number under which the file was opened.

string variable is the variable name to which the line is assigned.

Examples

```
50 LINE INPUT# 1, STATES
50 LINE INPUT# 3, CUSTOMER$
```

Comments

1. 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.
2. LINE INPUT# is especially useful if 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.

Program Segment

```
Ok
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
```

8.7 LSET and RSET

Function

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

Formats

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

Examples

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

Comments

1. 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. Refer to the MKI\$, MKS\$, MKD\$ functions, in Subsection 9.2.12.
2. LSET or RSET can also be used with a non-fielded string variable to left-justify or right-justify a string in a given field.

Program Segment

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

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

8.8 OPEN (Files)

Function

To allow I/O to a disk file.

Format

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

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 1 and 15. The file number is then associated with the file for as long as it is open. It is used to refer other disk I/O statements to the file.

filename is a string expression containing a name that conforms to the operating system's rules for disk filenames.

reclen is an integer expression that, if included, sets the record length for random files. The default record length is 128 bytes.

Example

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

This opens a file named INVEN in the sequential input mode. The file number is 2. The default record length of 128 bytes is used.

Comments

1. A disk file must be opened before any disk I/O operation can be performed on it.
2. The OPEN statement allocates a buffer for I/O to the file and determines the mode of access to use with a buffer.

CAUTION:

A file can be opened for output on only one file number at a time. A file can be opened for sequential input or random access on more than one file number at a time.

8.9 OPEN (Devices)Function

To establish addressability between a physical device and an I/O buffer in the data pool.

Format

```
OPEN [<dev>] <filename> [FOR <mod>] AS [#]<file number> [LEN=<recl>]
```

dev is optionally part of the filename string and can be one of the following:

```
A-D:    Disk
KYBD:   Keyboard (Input Only)
LCRN:   Screen (Output Only)
LPT1:   Line Printer 1
LPT2:   Line Printer 2
LPT3:   Line Printer 3
CAS1:   Cassette
COM1:   RS232 Communications 1
COM2:   RS232 Communications 2
```

filename a valid string literal or variable optionally containing a <dev>. If <dev> is omitted, disk A: is assumed. Refer to "DISK FILES" for naming conventions.

mode determines the initial positioning within the file and the action to be taken if the file does not exist. The valid modes and actions taken are:

```
INPUT    positions to the beginning of an
          existing file. A "File not found"
          error is given if the file does not
          exist.
```

```
OUTPUT   positions to the beginning of the
          file. If the file does not exist,
          one is created.
```

```
APPEND   positions to the end of the file.
          If the file does not exist, one is
          created.
```

If the FOR <mode> clause is omitted, the initial position is at the beginning of the file. If the file is not found, one is created. This is the Random I/O mode. That is, records can be read or written at will at any position within the file.

file number is an integer expression returning a number in the range 1 thru 15. The number is used to associate an I/O buffer with a disk file or device. This association exists until a CLOSE or CLOSE <file number> statement is executed.

lrecl is an integer expression in the range 2 to 32768. This value sets the record length to be used for random files (Refer to the FIELD statement). If omitted, the record length defaults to 128 byte records.

Examples

```
10 OPEN "CAS1:MYDATA" FOR OUTPUT AS #1
10 OPEN "PARTS.DAT" AS #1 'for random I/O on Disk A:
10 OPEN "KYBD:" FOR INPUT AS #2
10 OPEN "B:INVENT.DAT: FOR APPEND AS #1
```

Comments

1. Any values entered outside of the ranges given will result in an "Illegal Function Call" Error. The file is not opened.
 2. If the file is opened as INPUT, attempts to write to the file will result in a "Bad File Mode" error.
 3. If the file is opened as OUTPUT, attempts to read the file will result in a "Bad File Mode" error.
 4. At any one time, it is possible to have a particular disk filename OPEN under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, to use different file numbers for different modes of access. Each file number has a different buffer, so several records from the same file can be kept in memory for quick access.
- A file cannot be opened for output, however, on more than one file number at a time.
5. If the LEN=<lrecl> option is used, lrecl cannot exceed the value set by the /S:<lrecl> switch option to the command line.

6. For each device, the following OPEN modes are allowed:

KYBD:	INPUT only.
SCRN:	OUTPUT only.
CAS1:	INPUT or OUTPUT only.
LPT1:	OUTPUT only.
LPT2:	OUTPUT only.
LPT3:	OUTPUT only.
COM1:	INPUT,OUTPUT or random only.
COM2:	INPUT, OUTPUT or random only.

Disk files allow all modes.

7. When a disk file is opened FOR APPEND, the position is initially at the end of the file and the record number is set to the last record of the file (LOF(x)/128). PRINT, WRITE or PUT will then extend the file. The program may position elsewhere in the file with a GET statement. If this is done, the mode is changed to random and the position moves to the record indicated. Once the position is moved from the end of the file, additional records may be appended to the file by executing a GET #x,LOF(x)/<lrecl> statement. This positions the file pointer at the end of the file in preparation for appending.

Program Segment

```

Ok
10 OPEN "0", #1, "INFO"
20 INPUT "ENTER NAME"; NME$
30 IF NME$ = "STOP" THEN END
40 INPUT "ENTER STREET"; STREET$
50 INPUT "ENTER CITY"; CITY$
60 INPUT "ENTER STATE"; STATE$
70 INPUT "ENTER ZIPCODE"; ZIP$
80 PRINT #1, NME$, " "; STREET$, " ", CITY$, " "; STATE$, " "; ZIP$

```

8.10 PRINT# AND PRINT # USING

Function

To write data to a sequential disk file.

Format

PRINT # <filenumber>, [USING <string exp>]<list of exps>

file number	is the number used when the file was opened for output.
string exp	is comprised of formatting characters as described in Subsection 7.2.2, PRINT USING.
list of exps	includes numeric and/or string expressions written to the file.

Comments

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

Example:

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

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

2. The PRINT# statement does not compress data on the disk. An image of the data is written to the disk, just as it is displayed on the screen with a PRINT statement. For this reason, the data on the disk should be delimited carefully, so that it will be input correctly from the disk.

3. String expressions must be separated by semicolons in the list. Use explicit delimiters in the list of expressions to format the string expressions correctly on the disk.

Example:

```
LET A$="CAMERA" and B$="93604-1".
```

The statement: PRINT#1,A\$;B\$ writes 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$;" ";B$
```

As a result, the image written to disk is CAMERA,93604-1. This can be read back into two string variables.

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

Example:

```
10 LET A$="CAMERA, AUTOMATIC"
20 LET B$=" 93604-1"
```

The statement PRINT#1,A\$;B\$ would write the following image to disk:

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

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) ;A\$;CHR\$(34) ;CHR\$(34) ;B\$;CHR\$(34) writes the following image to disk:

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

The statement INPUT#1,A\$,B\$ inputs "CAMERA, AUTOMATIC" to A\$ and " 93604-1" to B\$.

You can also use the PRINT# statement with the USING option to control the format of the disk file.

Example:

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

Refer also to WRITE#, Section 8.12.

8.11 PUT

Function

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

Format

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

file number is the number under which the file was opened.

record number specifies the record to be put. The largest possible record number is 32767. The smallest record number is 1. If <record number> is omitted, the record will have the next available record number (after the last PUT).

Examples

```
80 PUT 7, 110
80 PUT#5, RECNUM%
```

Comments

1. PRINT#, PRINT# USING, and WRITE# can be used to put characters in the random file buffer before a PUT statement.
2. In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

Program Segment

```
Ok
10 OPEN "R", #3, "MAILING.DAT", 62
20 FIELD #3, 15 AS LAST$, 15 AS FIRST$, 15 AS STREET$, 10 AS CITY$, 2 AS
STATES$, 5 AS ZIP$
30 INPUT "ENTER RECORD NUMBER:", RECNUM%
40 INPUT "ENTER LAST NAME:", L$ : LSET LAST$ = L$
50 INPUT "ENTER FIRST NAME:", F$ : LSET FIRST$ = F$
60 INPUT "ENTER STREET:", $$ " LSET STREET$ = S$
70 INPUT "ENTER CITY:", C$ " LSET CITY$ = C$
80 INPUT "ENTER STATE:", ST$ " LSET STATES$ = ST$
90 INPUT "ENTER ZIPCODE:", Z$ " LSET ZIP$ = Z$
100 PUT #3, RECNUM%
110 GOTO 30
```


8.12 WRITE#Function

To write data to a sequential file.

Format

WRITE#<file number>, <list of expressions>

file number	is the number under which the file was opened in "O" mode.
-------------	--

list of expressions	includes either string or numeric expressions, separated by commas.
---------------------	---

Example

```
70 WRITE#1, ITEM$, STOCKNUMBERS$
```

Comment

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

Program Segment

```
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\$.

CHAPTER 9 FUNCTIONS

9.1 INTRODUCTION

BASIC's intrinsic functions are explained in this chapter. You can use these functions within any BASIC program.

You must always enclose an argument to a function in parentheses. In the format lines used to explain the functions in this chapter, the arguments are abbreviated as follows:

X and Y	represent numeric expressions
I and J	represent integer expressions
S\$ and T\$	represent string expressions

If you supply a floating point value in an expression where an integer is required, BASIC rounds the fractional portion and uses the resulting integer.

With the BASIC interpreter, only integer and single-precision results are returned by functions. Double-precision numbers are supported only by the BASIC compiler.

9.2 ARITHMETIC FUNCTIONS

The BASIC arithmetic functions return numeric values. Among the tasks they accomplish are the following:

- Conversion of a string value to a numeric value
- Return of an algebraic value
- Conversion to an integer, single-precision, or double-precision value
- Rounding
- Return of an absolute value
- Return of an exponent
- Return of the free space in memory
- Return of a random value
- Return of the sign of a value

9.2.1 ABSFunction

Returns the absolute value of the expression X.

Format

ABS(X)

Examples

```
10 ABS (35-50)
10 DEV=ABS(X1-X2)
```

Program Segment 1

```
Ok
10 A = 30
20 B = 90
30 PRINT ABS (A/(A+B))
RUN
.25
Ok
```

Program Segment 2

```
Ok
10 FOR I = 1 TO 3
20 READ J
30 DATA 200, 0, -314
40 PRINT J, ABS(J)
50 NEXT I
60 END
RUN
200          200
0            0
-314         314
Ok
```

9.2.2 ATN

Function

Returns the arctangent of X in radians.

Format

ATN(X)

Examples

```
10 ATN (4)
10 MEASURE = ATN(Y/X)
```

Comment

1. The result is in the range $-\pi/2$ to $\pi/2$. The expression X can be any numeric type, but the evaluation of ATN is always performed in single-precision.

Program Segment

```
Ok
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
  1.24905
Ok
```

9.2.3 CDBLFunction

Converts X to a double-precision number.

Format

CDBL(X)

Examples

```
10 PRINT CDBL (421.87)
30 PRINT 1/7, CDBL(1/7)
```

Program Segment

```
Ok
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
454.67 454.6700134277344
Ok
```


9.2.4 CINT(X)

Function

Converts X to an integer by rounding the fractional portion.

Format

CINT(X)

Examples

```
PRINT CINT(45.67)
10 ROUND = CINT(97.125)
```

Comments

1. If X is not in the range -32768 to 32767, an "Overflow" error occurs.
2. Refer to the CDBL and CSNG functions (Sections 9.2.3 and 9.2.6) for converting numbers to the double-precision and single-precision data types. Also, refer to the FIX and INT functions (Sections 9.2.8 and 9.2.10), both of which return integers.

Program Segment 1

```
10 A = 32.74
20 PRINT CINT(A)
RUN
33
Ok
```

Program Segment 2

```
Ok
10 FOR K = 1 TO 7
20 INPUT "ENTER TEST GRADE:", GRADE
30 TOTAL = TOTAL + GRADE
40 NEXT K
50 AVG = TOTAL/7
60 PRINT "THE AVERAGE, ROUNDED TO THE NEAREST INTEGER IS:; CINT(AVG)"
70 END
```

9.2.5 COSFunction

Returns the cosine of X. X is in radians. The calculation of COS(X) is performed in single-precision.

Format

COS(X)

Examples

```
20 A = 4*COS(.7)
20 VALUE = COS(DEGREE*3.141593/180)
```

Program Segment 1

```
10 X = 2*COS(.4)
20 PRINT X
RUN
1.84212
Ok
```

Program Segment 2

```
Ok
10 PI = 3.141593
20 INPUT "ENTER MEASURE OF ANGLE IN DEGREES:", DEGREES
30 RADIANS = DEGREES * (PI/180)
40 PRINT "COS("; DEGREES; ") ="); COS(RADIANS)
50 END
RUN
ENTER MEASURE OF ANGLE IN DEGREES: 45
COS(45) = .7071068
Ok
```

9.2.6 CSNG

Function

Converts X to a single-precision number.

Format

CSNG(X)

Examples

```
10 PRINT CSNG(100/30)
```

Comment

1. Refer to the CINT and CDBL functions for converting numbers to the integer and double-precision data types.

Program Segment

```
OK
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
  975.3421   975.3421
Ok
```

9.2.7 EXPFunction

Returns e to the power of X . X must be ≤ 87.3365 .

Format

EXP(X)

Examples

```
50 PRINT EXP (7)
50 CONST = EXP (.X* .5)/4
```

Comments

1. If the number calculated by the EXP function causes an "Overflow" error then, machine infinity, with the appropriate sign, is supplied as the result, and execution continues.

Program Segment

```
10 X = 5
20 PRINT EXP (X-1)
RUN
54.5982
Ok
```

9.2.8 FIXFunction

Returns the truncated integer part of X.

Format

FIX(X)

Examples

```
100 PRINT FIX(37.42)
100 TRUNCX = FIX(X)
```

Comments

1. FIX(X) is equivalent to $\text{SGN}(X) * \text{INT}(\text{ABS}(X))$.
2. The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Program Segment

```
Ok
10 PRINT FIX(58.75)
RUN
58
Ok
10 PRINT FIX(-58.75)
RUN
-58
Ok
```


9.2.9 FREFunction

FRE returns the number of bytes in memory that BASIC is not using.

Format 1

FRE(X)

Format 2

FRE(S\$)

Example

```
10 PRINT FRE(0)
```

Comments

1. FRE("") forces a garbage collection before returning the number of free bytes. During a garbage collection BASIC collects any useful data and releases any unused areas of memory that were once used for strings. BE PATIENT: garbage collection may take a while. BASIC will not initiate a garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

2. The arguments you supply to the FRE function are dummy arguments.

9.2.10 INTFunction

Returns the largest integer $\leq X$.

Format

INT(X)

Example

```
10 PRINT INT(53.15)
```

Comment

1. Refer to the FIX and CINT functions which also return integer values.

Program Segment

```
Ok  
10 PRINT INT(99.89)  
RUN  
99
```

```
Ok  
10 PRINT INT(-12.11)  
RUN  
-13  
Ok
```

9.2.11 LOGFunction

Returns the natural logarithm of X. X must be greater than zero.

Format

LOG(X)

Examples

```
110 LOG(15)
110 NUVAL = LOG(SQR(ABS(NUM)))
```

Program Segment

```
Ok
10 PRINT LOG(45/7)
RUN
1.860752
Ok
```

9.2.12 MKI\$, MKS\$, and MKD\$Function

Convert numeric values to string values.

Format

```
MKI$(<integer expression>)
MKS$(<single-precision expression>)
MKD$(<double-precision expression>)
```

Examples

```
80 CODE$ = MKI$(I)
80 GROSS$ = MKS$(F)
80 SALES$ = MKP$(P)
```

Comments

1. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI\$ converts an integer to a 2-byte string. MKS\$ converts a single-precision number to a 4-byte string. MKD\$ converts a double-precision number to an 8-byte string.

2. Refer also to CVI, CVS, CVD, Subsection 9.3.3.

Program Segment

```
90  AMT=(K+T)
100  FIELD #1, 8 AS D$, 20 AS N$
110  LSET D$ = MKS$(AMT)
120  LSET N$ = A$
130  PUT #1
.
.
```

9.2.13 RNDFunction

Returns a random number between 0 and 1.

Format

RND[(X)]

Examples

```
60 PRINT RND
70 IF RND< = .5 THEN PRINT "HEADS" ELSE PRINT "TAILS"
```

Comments

1. X omitted generates the next random number in the sequence. X=0 repeats the last number generated.
2. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (refer to RANDOMIZE, Section 7.23).
3. X<0 always restarts the same sequence for any given X.

Program Segment

```
Ok
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT I
RUN
24 30 31 51 5
Ok
```


9.2.14 SGNFunction

IF $X > 0$, SGN(X) returns 1.
IF $X = 0$, SGN(X) returns 0.
IF $X < 0$, SGN(X) returns -1.

Format

SGN(X)

Example

110 ON SGN(X)+2 GOTO 100,200,300

Result:

The program branches to 100 if X is negative, 200 if X is 0, and 300 if X is positive.

Program Segment

```
Ok
10 INPUT NUM
20 ROUNDNUM = SGN(NUM) * INT(ABS(NUM) * 100 + .5)/100
30 PRINT NUM: "ROUNDED TO THE NEAREST HUNDREDTH IS"; ROUNDNUM
RUN
732.7868
  32.7868 ROUNDED TO THE NEAREST HUNDREDTH IS 32.79
Ok
RUN
?-12.2061
-12.2061 ROUNDED TO THE NEAREST HUNDREDTH IS -12.21
Ok
```

9.2.15 SINFunction

Returns the sine of X in radians. SIN(X) is calculated in single-precision.

Format

SIN(X)

Example

```
10 PRINT SIN(2.3)
10 VALUE = SIN(DEGREE*3.141593/180)
10 COS(X)=SIN(X+3.14159/2).
```

Program Segment 1

```
Ok
10 PRINT SIN(1.5)
RUN
.9974951
Ok
```

Program Segment 2

```
Ok
10 PI = 3.141593
20 INPUT "ENTER MEASURE OF ANGLE IN DEGREE"; DEGREES
30 RADIANS = DEGREES * (PI/180)
40 PRINT "SIN("; DEGREES; ") ="; SIN(RADIANS)
RUN
```

9.2.16 SQRFunction

Returns the square root of X. X must be ≥ 0 . If a negative value is supplied for X, an "Illegal function call" error results.

Format

SQR(X)

Examples

```
110 PRINT SQR (16)
110 ROOT = SQR(NUNUM)
```

Program Segment

```
OK
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
  10          3.162278
  15          3.872984
  20          4.472136
  25          5
Ok
```

9.2.17 TANFunction

Returns the tangent of X in radians. TAN(X) is calculated in single-precision.

Format

TAN(X)

Example

```
10 Y = Q*TAN(X)/2
```

Comment

1. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Program Segment

```
Ok
10 PI = 3.141593
20 INPUT "ENTER MEASURE OF ANGLE IN DEGREES:", DEGREES
30 RADIANS = DEGREES * (PI/180)
40 PRINT "TAN(' DEGREES; ") ="; TAN(RADIANS)
```

9.3 STRING FUNCTIONS

String functions manipulate string values. Among the tasks accomplished by string functions are the following:

- Return of the ASCII code of a string character
- Return of the ASCII character corresponding to an ASCII code
- Conversion of string values to numeric values
- Return of a string from a file or a screen
- Searching a string for the occurrence of a substring
- Return of the length of a string variable
- Returns the numeric value of a string

9.3.1 ASC

Function

Returns a numerical value that is the ASCII code of the first character of the string S\$.

Format

ASC(S\$)

Examples

```
10 PRINT ASC(S$)
10 PRINT ASC("A")
```

Comments

1. See Appendix A for ASCII codes.
2. If S\$ is null, an "Illegal function call" error is returned.

Program Segment

```
Ok
10 S$ = "TEST"
20 PRINT ASC(S$)
RUN
84
Ok
```

Refer to the Subsection 9.3.2 for ASCII-to-string conversion.

9.3.2 CHR\$Function

Returns a string whose one element has ASCII code I.

Format

CHR\$(I)

Examples

```
10 PRINT CHR$(65)
10 QUOTE$ = CHR$(34)
```

Comments

1. ASCII codes are listed in Appendix A.
2. CHR\$ is commonly used to send a special character to the screen. (ASCII codes are listed in Appendix A.) For instance, PRINT CHR\$(07) sounds the alarm, and PRINT CHR\$(12) clears the screen and returns the cursor to the home position.
3. See the ASC function (Subsection 9.3.1) for ASCII-to-numeric conversion.

Program Segment 1

```
Ok
10 PRINT CHR$(66)
RUN
B
Ok
```

Program Segment 2

```
10 QUOTE$ = CHR$(34)
20 PRINT QUOTE$; "WANG LABORATORIES, INC."; QUOTE$
RUN
"WANG LABORATORIES, INC."
Ok
```

9.3.3 CVI, CVX, CVD

Function

Convert string values to numeric values. Alpha strings that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single-precision number. CVD converts an 8-byte string to a double-precision number.

Format

```
CVI( 2-byte string )
CVX( 4-byte string )
CVD( 8-byte string )
```

Examples

```
80 I = CVI(CODE = $)
80 F = CVX(GROSS$)
80 D = CVD(SALES$)
```

Comment

1. Also refer to MKI\$, MKS\$, MKD\$, Subsection 9.2.12.

Program Segment

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
.
.
.
```

9.3.4 HEX\$Function

Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX\$(X) is evaluated.

Format

HEX\$(X)

Examples

```
70 PRINT HEX$(72)
70 HVAL$ = HEX(J)
```

Comments

1. Refer to the OCT\$ function (Subsection 9.3.11) for octal conversion.
2. The maximum value allowed for X is 65535. If a larger value is assigned to X, an "Overflow" error results.

Program Segment

```
Ok
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
32 DECIMAL IS 20 HEXADECIMAL
Ok
```

9.3.5 INKEY\$Function

Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the screen.

Format

INKEY\$

Examples

```
40 N$ = INKEY$
40 IF INKEY$ = "E" THEN END
```

Comment

1. No characters will be echoed and all characters are passed through to the program except for Control-C, which terminates the program. (With the BASIC Compiler, Control-C is also passed through to the program.)

Program Segment

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSES$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSES$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

9.3.6 INPUT\$Function

Returns a string of X characters, read from the screen or from file number Y.

Format

INPUT\$(X[, [#]Y])

Example

```
100 IF INPUT (1) = "E" THEN END
```

Comment

1. If the screen is used for input, no characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT\$ function.

Program Segment 1

```
Ok
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN HEXADECIMAL
10 OPEN "I", 1, "DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1, #1)));
40 GOTO 20
50 PRINT
60 END
```

Program Segment 2

```
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 S$=INPUT$(1)
120 IF S$="P" THEN 500
130 IF S$="S" THEN 700 ELSE 100
.
.
.
```


9.3.7 INSTR

Function

Searches for the first occurrence of string T\$ in S\$ and returns the position at which the match is found. Optional offset I sets the position for starting the search.

Format

INSTR([I,]S\$,T\$)

Example

```
60 PLACE = INSTR(SEARCH$, CHAR$)
```

Comments

1. I must be in the range 1 to 255.
2. If I > LEN(S\$) or if S\$ is null or if T\$ cannot be found, INSTR returns 0. If T\$ is null, INSTR returns I or 1. S\$ and T\$ may be string variables, string expressions or string literals.
3. If I=0 is specified, error message "ILLEGAL ARGUMENT IN line number " will be returned.

Program Segment

```
Ok
10 S$ = "ABCDEB"
20 T$ = "B"
30 PRINT INSTR(S$,T$);INSTR(4,S$,T$)
RUN
2    6
Ok
```

9.3.8 LEFT\$Function

Returns a string comprised of the leftmost I characters of S\$.

Format

LEFT\$(S\$,I)

Examples

```
30 FIRST$ = LEFT$(FULL$,NUM)
```

Comments

1. I must be in the range 0 to 255.
2. If I is greater than LEN(S\$), the entire string (S\$) will be returned. If I=0, the null string (length zero) is returned.
3. Also see the MID\$ and RIGHT\$ functions (Subsections 9.3.10 and 9.3.12, respectively).

Program Segment 1

```
Ok
10 A$ = "BASIC MANUAL"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Program Segment 2

```
Ok
10 FULL$ = "MICKEY MOUSE"
20 PLACE = INSTR(FULL$," ")
30 PRINT "FIRST NAME IS " LEFT$(FULL$, PLACE -1)
RUN
MICKEY
Ok
```

9.3.9 LENFunction

Returns the number of characters in S\$.

Format

LEN(S\$)

Examples

```
30 PRINT LEN(S$)
30 FOR COUNT = 1 TO LEN(S$)
```

Comment

1. Non-printing characters and blanks are counted.

Program Segment

```
Ok
10 S$ = "PORTLAND, OREGON"
20 PRINT LEN (S$)
RUN
16
Ok
```

9.3.10 MID\$Function

Returns a string of length J characters from S\$ beginning with the Ith character.

Format

MID\$(S\$,I[,J])

Examples

```
15 PRINT MID$(MENU$, 1, 7)
15 MID$(TEXT$,G) = "###"
```

Comments

1. I and J must be in the range 1 to 255.
2. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned.
3. If I > LEN(S\$), MID\$ returns a null string.
4. Also refer to the LEFT\$ and RIGHT\$ functions (Subsections 9.3.8 and 9.3.12, respectively).
5. If you specify I=0 then the following error message will appear: "Illegal Argument in <line number>".

Program Segment

```
Ok
LIST
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOO EVENING
Ok
```

9.3.11 OCT\$Function

Returns a string which represents the octal value of the decimal argument.

Format

OCT\$(X)

Examples

```
70 PRINT OCT$(40)
70 OCTVAL$ = OCT$(J)
```

Comments

1. X is rounded to an integer before OCT\$(X) is evaluated.
2. Refer to the HEX\$ function (Subsection 9.3.4) for hexadecimal conversion.
3. The maximum value for X is 65535.

Program Segment

```
Ok
10 PRINT OCT$(24)
RUN
30
Ok
```


9.3.12 RIGHT\$Function

Returns the rightmost I characters of string S\$.

Format

RIGHT\$(S\$,I)

Example

20 RIGHT\$(A\$,3)

Comments

1. If I=LEN(S\$), returns S\$. If I=0, the null string (length zero) is returned.
2. Also refer to the MID\$ and LEFT\$ functions (Subsections 9.3.8 and 9.3.10, respectively).

Program Segment

```
Ok
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$,5)
RUN
BASIC
Ok
```

9.3.13 SPACE\$Function

Returns a string of spaces of length X.

Format

SPACE\$(X)

Examples

```
110 SPACE$(5)
110 PRINT SPACE$(15); "HELLO"
```

Comments

1. The expression X is rounded to an integer and must be in the range 0 to 255.
2. Also see the SPC function (Subsection 9.5.6).

Program Segment

```
Ok
10 FOR I = 1 TO 5
20 S$ = SPACE$(I)
30 PRINT S$;I
40 NEXT I
RUN
  1
    2
      3
        4
          5
Ok
```

9.3.14 STR\$Function

Returns a string representation of the value of X.

Format

STR\$(X)

Example

```
130 PRINT STR$(30)
```

Comment

1. Also refer to the VAL function.

Program Segment

```
Ok
10 INPUT "ENTER STREET NAME"; N$
20 INPUT "ENTER STREET NUMBER"; N
30 B$ = STR$(N) + " " + N$
40 PRINT B$
```

9.3.15 STRING\$Function

Returns a string of length I whose characters all have ASCII character code of J or the first character of S\$.

Format

STRING\$(I,J)

Examples

```
710 S$ = STRING$(10,45)
710 PRINT STRING$(20,46)
```

Program Segment

```
LIST
10 S$ = STRING$(10,65)
20 PRINT S$
RUN
AAAAAAAAAA
```

9.3.16 VALFunction

Returns the numerical value of string S\$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string.

Format

VAL(S\$)

Example

```
150 PRINT COL1VAL ;SPC(30) COL2VAL
```

Comment

1. Refer to the STR\$ function for numeric to string conversion.

Program Segment

```
Ok
10 READ NAME$,CITY$STATE$ZIP$
20 IF VAL(ZIP$) 90000 OR VAL(ZIP$) 96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VALZIP$) =90801 OR VAL(ZIP$) =90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
.
.
.
```


9.4 INPUT/OUTPUT FUNCTIONS

The Input/Output functions assist in file and memory operations.

9.4.1 EOF

Function

Returns -1 (true) if the end of a sequential file has been reached.

Format

EOF(file number)

Example

```
30 IF EOF(7) THEN 400
```

Comment

1. Use EOF to test for end-of-file while inputting, to avoid "Input past end" errors.

Program Segment

```
Ok
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
```

.
.
.

9.4.2 INPFunction

Returns the byte read from port I.

Format

INP(I)

Example

```
100 A=INP(255)
```

Comments

1. I must be in the range 0 to 255.
2. INP is the complementary function to the OUT statement, Section 7.19.

9.4.3 LOC

Function

With random disk files, LOC returns the record number just read or written from a GET or PUT.

Format

LOC(<file number>)

Example

```
200 IF LOC(1) 50 THEN STOP
```

Comments

1. If the file was opened but no disk I/O has been performed yet, LOC returns a 0.
2. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was opened.

9.4.4 LOFFunction

The LOF function returns the number of bytes allocated to the file.

Format

LOF(<file number>)

Comment

1. <filenumber> is tied to the currently open file.

Program Segment

```
10 OPEN "FILE.BIG" AS #1
20 GET #1,LOF(1)/128
```

```
  .
  .
  .
```

The above statements will get the last record of the file FILE.BIG assuming that the file was created with a default record length of 128 bytes.

9.4.5 PEEK

Function

Returns the byte (decimal integer in the range 0 to 255) read from memory location I.

Format

PEEK(I)

Example

400 A=PEEK(&H5A00)

Comments

1. I must be in the range 0 to 65536.
2. PEEK is the complementary function to the POKE statement, Section 7.20.

9.4.6 VARPTRFunction

For files, the VARPTR function returns the address of the first byte of the File Control Block (FCB) for the opened file.

Format

VARPTR(<file number>)

file number is tied to a currently open file. Offsets to information in the FCB from the address returned by VARPTR are:

<u>OFF</u>	<u>SIZE</u>		<u>CONTENTS</u>
0	1	Mode	The mode in which the file was opened: 1 - Input Only 2 - Output Only 4 - Random I/O 16 - Append Only 32 - Internal use 64 - Future use 128 - Internal use
1	38	FCB	Disk File Control Block.
39	2	CURLOC	Number of sectors read or written for sequential access. For random access, it contains the last record number +1 read or written.
41	1	ORNOFS	Number of bytes in sector when read or written.
42	1	NMLOFS	Number of bytes left in Input buffer.
43	3	***	Reserved for future expansion.
46	1	DEVICES	Device Number: 0-9 - Disks A: thru J: 255 - KYBD: 254 - SCRNI: 253 - LPT1: 252 - CAS1: 251 - COM1: 250 - COM2: 249 - LPT2: 248 - LPT3:
47	1	WIDTH	Device Width.
48	1	POS	Position in buffer for PRINT.
49	1	Flags	Internal use during LOAD/SAVE not used for data files.
50	1	OUTPOS	Output Position used during tab expansion.
51	128	BUFFER	Physical Data buffer. Used to transfer data between DOS and BASIC. Use this offset to examine data in Sequential I/O mode.

Functions

179	2	VRECL	Variable length record size. Default is 128. Set by length option is OPEN statement.
181	2	PHYREC	Current physical record number.
183	2	LOGREC	Current logical record number.
185	1	***	Future use.
186	2	OUTPOS	Disk files only. Output position for PRINT, INPUT, and WRITE.
188	<n>	FIELD	Actual FIELD data buffer. Size is determined by /S: switch. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine File data in Random I/O mode.

Program Segment

```

Ok
10 OPEN " DATA.FIL" AS #1.
20 FCBADR = VARPTR(#1) .
30 DATADR = FCBADR+188
40 A$ = PEEK(DATADR) '
FCBADR contains start of FCB.
DATADR contains address of data buffer.
A$ contains 1st byte in data buffer.

```

9.4.7 VARPTR

Function

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 CAN be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Format 1

VARPTR(<variable name>)

Format 2

VARPTR(#{<file number>})

Example

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

Comments

1. All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.
2. Format 2 for sequential files returns the starting address of the disk I/O buffer assigned to file number. For random files, Format 2 returns the address of the FIELD buffer assigned to <file number>.

9.5 SPECIAL FUNCTIONS

BASIC includes an assortment of special functions that determine cursor and print head position and return the coordinate values or attributes of cursor coordinates. The USR function initiates the call to a user assembly language subroutine.

9.5.1 CSRLIN

Function

To return the current row position of the cursor

Format

x = CSRLIN

x is a numeric variable receiving the value returned by a function. The returned value is in the range 1 to 24.

Comments

1. The CSRLIN function returns the current row position of the cursor. The statement, x = POS(0) returns the column location of the cursor, a value in the range 1 to 40 or 1 to 80, depending upon the current line width of the screen.

Example

```
10 Y = CSRLIN 'Record current row
20 X = POS(0) 'Record current column
30 LOCATE 24, 1: PRINT "SMILE" 'Print SMILE on last line
40 LOCATE Y,X 'Restore position of cursor
```

9.5.2 LPOSFunction

Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head.

Format

LPOS(X)

X is a dummy argument.

Example

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

9.5.3 POINT

Function

Returns the attribute value of the specified pixel from the screen.

Format

POINT (<x-coordinate>, <y-coordinate>)

Comment

1. In 40-column mode, valid results are 0 through 15. In 80-column mode, valid results are 0 through 3. In 300 x 800 mode, valid results are 0 through 1. If the specified pixel is out of range, a value of -1 is returned.

9.5.4 POSFunction

Returns the current horizontal (column) cursor position. The leftmost position is 1. X is a dummy argument.

Format

POS(I)

Example

```
40 IF POS(X)>60 THEN PRINT CHR$(13)
40 ROUNDX = INT(POSX + .005) * 100/100
```

Comment

1. Refer to the LPOS function (Subsection 9.5.2).

9.5.5 SCREENFunction

Returns the ordinal or color attribute of the character at the specified row and column.

Format

`x = SCREEN (<row>, <column> [,z])`

<code>x</code>	is a numeric variable receiving the ordinal returned.
<code>row</code>	is a valid numeric expression returning an unsigned integer in the range 1 to 24.
<code>column</code>	is a valid numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending upon the current width.
<code>z</code>	is a valid numeric expression returning a Boolean result.

Comments

1. Any values entered outside the required ranges results in an "Illegal function call" error.
2. If the optional parameter "z" is given and is non-zero, the color attribute for the character is returned instead.

9.5.6 SPCFunction

Prints I blanks on the screen. SPC can only be used with the PRINT and LPRINT statements.

Format

SPC(I)

Comments

1. I must be in the range 0 to 255. A 'semicolon' is assumed to follow the SPC(I) command.
2. Refer also to the SPACE\$ function.

Program Segment

```
Ok
10 PRINT "OVER" SPC(15) "THERE"
RUN
OVER          THERE
Ok
```

9.5.7 TABFunction

Spaces to position I on the screen.

Format

TAB(I)

Example

```
10 PRINT "NAME" TAB (25) "POSITION":PRINT
```

Comments

1. I must be in the range 1 to 255.
2. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. TAB can only be used in PRINT and LPRINT statements.

Program Segment

```
Ok
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB (25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                AMOUNT

G. T. JONES          $25.00
Ok
```

9.5.8 USRFunction

Calls the user's assembly language subroutine with the argument X.

Format

USR[<digit>](X)

digit	is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine.
-------	---

Comment

1. If <digit> is omitted, USR0 is assumed.

Program Segment

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
```

```

.
.
.
```

Using USR Action CallsFormat

x = USR[<digit>] (argument)

digit	is an integer from 0 to 9. <digit> specifies which USR routine is being called. (Refer to "The DEF USR Statement"). If <digit> is omitted, USR0 is assumed.
-------	---

argument	is an numeric or string expression.
----------	-------------------------------------

x	is the variable receiving the result of the function call. Its type (numeric or string) must be consistent with the argument passed. The variable type may be set to Integer by calling MAKINT in the user's routine before returning to BASIC.
---	---

Comments

1. A DEF SEG statement must be executed prior to a USR call to assure that the Code Segment points to the subroutine being called. The segment given in the DEF SEG statement determines the starting segment of the subroutine. (Refer to Section 7.7 for a discussion of the DEF SEG statement).
2. For each USR function used, a corresponding DEF USR statement must have been executed to define the USR call offset. The address given in the DEF USR statement determines the starting address of the subroutine.
3. When the USR function call is made, register [AL] contains a value that specifies the type of argument that was given. The value in [AL] may be one of the following:
 - 2 Two-byte integer (two's complement)
 - 3 String
 - 4 Single-precision floating point number
 - 8 Double-precision floating point number
4. If the argument is a number, the [BX] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC- contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

CHAPTER 10 DEBUGGING

10.1 INTRODUCTION

BASIC provides several instructions that aid in program debugging. These features permit you to stop, resume, and trace program execution.

10.2 CONT

(NC)

Purpose

To continue program execution after a Control-C has been issued, or a STOP or END statement has been executed.

Format

CONT

Comments

1. CONT can be used to continue execution after an error. 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 (i.e., or prompt string).
2. You can use CONT in conjunction with STOP for debugging. When execution is stopped, intermediate values can be examined and changed using direct mode statements. Then you can resume execution with CONT or a direct mode GOTO, which resumes execution at a specified line number.
3. CONT is invalid if the program has been edited during the break.

Program Segment

```
.  
. 100 STOP  
110 PRINT "... and as I was saying..."  
. 110 PRINT "... and as I was saying..."  
. 110 PRINT "... and as I was saying..."  
RUN  
. 110 PRINT "... and as I was saying..."  
. 110 PRINT "... and as I was saying..."  
BREAK IN 100  
CONT  
...and as I was saying...  
. 110 PRINT "... and as I was saying..."  
. 110 PRINT "... and as I was saying..."  
. 110 PRINT "... and as I was saying..."
```

10.3 TRON and TROFF

Purpose

To trace the execution of program statements.

Format

TRON
TROFF

Comment

1. 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).
2. TROFF is the default state of the system.

Program Segment

```

OK
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;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

```


CHAPTER 11 COLOR AND GRAPHICS

11.1 INTRODUCTION

BASIC includes instructions that draw characters and graphics on the screen in a variety of colors. Instructions that locate characters on the screen move the cursor to the row and column position on the screen where the character is to appear. Similarly, graphics instructions draw points, lines, and shapes on the screen according to horizontal (x-axis) and vertical (y-axis) pixel position. A pixel is the smallest addressable unit on the screen.

Graphic instructions require that a graphics card be installed in the system. Color can only be produced on a system which includes an industry-standard Monitor/Graphics card connected to a color monitor or TV screen. Refer to Table 11-1 to determine the capabilities of your system.

Table 11-1. Video Options of the Wang PC System

Monitor/Card	Screen Size (row x column)	Graphics (Y or N)	Resolution (XxY pixels)	Colors
Wang Monochrome Monitor with Wang Monitor Card	25 x 80	N		B&W
Wang Monochrome Monitor with Wang Monitor Card and Wang Graphics Card	25 x 80	Y	800 x 300	B&W
Any other monitor/TV with industry-standard Monitor/Graphics card	25 x 80 or 25 x 40	Y	640 x 225 or 320 x 225	4 colors or 16 colors

Not only does screen size determine the size and resolution of characters on the screen, but it also sets the number of colors that can be displayed at one time. The Wang color palette contains 16 colors. In 40-column mode (320 x 225 pixels), all 16 colors can be displayed at one time; in 80-column mode (640 x 225 pixels), only the first 4 colors can be displayed. In 300 x 800 mode, only black and white are displayed. You can use either the SCREEN or WIDTH command, as discussed in Sections 4.14 and 4.15, respectively, to switch between modes.

11.2 COLOR

Color output can only be produced on a color monitor or color TV screen. The Wang PC palette contains 16 slots, each of which can contain one of 16 user-created colors. The original Wang PC palette contains the colors listed in Table 11-2. However, the PALETTE statement can create any of these colors and place them in any of the 16 slots. In fact, the same color can appear in more than one slot (refer to Subsection 11.2.1).

Special color instructions and graphic instructions use the colors entered in the palette as foreground, background, and border colors. Characters and graphic figures appear on the screen in the foreground color. The background color fills the rest of the addressable area of the screen. The border color fills the periphery of the screen. The Wang Professional Computer implementation of BASIC does not distinguish between background and border colors; the border color is always color 0.

The BASIC COLOR statement enables subsequent output to be displayed in the selected foreground, background, and border colors. In addition, each BASIC graphic instruction contains a parameter that specifies the color for the graphic figure, and the BASIC PAINT statement allows you to specify the color which will fill a shape produced on the screen.

Table 11-2. The Wang PC Color Palette

Palette Entry	Color	Palette Entry	Color
0	Black	8	Dark grey
1	Blue	9	Light blue
2	Green	10	Light green
3	White	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Yellow
7	Light Grey	15	Cyan

Placement of colors in the palette is important. A color monitor in 40-column mode can use all 16 colors of the palette. However, a color monitor in 80-column mode can use only the first four colors in the palette (0 through 3). If you specify a color outside the range of 0 to 3 in 80-column mode, the two low-order bits of the binary value of the color specification determine which color is selected.

The Wang Monochrome Monitor only uses two colors. The background color is black and is considered to be color 0 when referenced in BASIC. The foreground color is white and is considered to be color 1 when referenced in BASIC.

11.2.1 PALETTE

Function

To place colors in the color palette.

Format

PALETTE
 PALETTE <entry-number>, <value>
 PALETTE USING <array>

entry-number is an integer in the range -32768 to 32767. Only the low-order byte is used to determine the entry (0-15).

value is an integer in the range -32768 to 32767. The easiest way to specify value is as a four-digit hexadecimal value (i.e., &H0F0F). The four hexadecimal digits represent the intensity, red, green, and blue attributes, respectively. A hex value of 0 to 7 turns off the respective attribute; a hex value of 8 to F turns on the respective attribute.

array is a 16-element integer array. Each array element represents a palette entry. Each element can contain a signed integer from -32768 to 32767 that represents a color as described above for the value parameter. If you do not want a palette-entry to change, specify -1 in the respective array-element.

Comments

1. PALETTE sets the color palette to its initial configuration, as listed in Table 11-2.
2. The second and third forms of the PALETTE instruction combine the red, green, blue, and intensity parameters and places the resulting color in a palette slot. Table 11-3 lists the 16 possible colors and the parameters that create them. In Table 11-3, a value of 0 indicates that the attribute of the parameter is off; a value of E indicates that the attribute is on.
3. When the graphics statements are used in any screen mode, palette-entry-3 is the default foreground color and palette-entry-0 is the default background color.

Examples

PALETTE (Sets the color palette to initial configuration: black is palette-entry 0, blue is palette-entry-1, etc.

PALETTE 0, &H0FF0 (Moves brown to palette-entry-0)

PALETTE 15, &HFF00 (Moves light red to palette-entry-15)

PALETTE USING Z%(0) (Uses array Z%() to define the palette)

Table 11-3. Combination of Attributes in the Creation of Colors

Resulting Color	Intensity	Red	Green	Blue
Black	0	0	0	0
Blue	0	0	0	E
Green	0	0	E	0
White	E	E	E	E
Red	0	E	0	0
Magenta	0	E	0	E
Brown	0	E	E	0
Light grey	0	E	E	E
Dark grey	E	0	0	0
Light blue	E	0	0	E
Light green	E	0	E	0
Light cyan	E	0	E	E
Light red	E	E	0	0
Light magenta	E	E	0	E
Yellow	E	E	E	0
Cyan	0	0	E	E

11.2.2 COLORPurpose

To set the foreground, background, and border colors for subsequent output.

Format

COLOR [<foreground>] [, [<background>] [,<border>]]

foreground	is an unsigned integer in the range 0 to 15.
background	is an unsigned integer in the range 0 to 15.
border	is an unsigned integer in the range 0 to 15. The Wang PC ignores the border parameter.

Comments

1. The Wang PC color palette contains 16 slots, numbered 0 through 15, into which the programmer can place any of a set of 16 user-created colors. The same color can be placed in more than one palette slot. Colors are created and entered into the palette by means of the PALETTE instruction described in Subsection 11.2.1.
2. The border color is set to the same color as the background color and cannot be set independently.
3. When the graphics statements are used in any screen mode, palette-entry-3 is the default foreground color and palette-entry-0 is the default background color.

11.2.3 PAINT

Function

The PAINT statement fills in a graphics figure of the specified border attribute with the specified paint attribute.

Format

PAINT (<x-start>, <y-start>) [,<color-attribute> [,<border-attribute>]]

x-start and y-start represent coordinates located ⁱⁿ outside the border of the figure. If they reference a border point, the PAINT statement has no effect.

color-attribute represents the color with which the figure is to be filled. If the color-attribute parameter is not included, the color-attribute defaults to the foreground color (1 in 80-column mode, 3 in 40-column mode).

border-attribute represents the color that surrounds the figure.

Comments

1. Because there are only two attributes in 300 x 800 mode, painting essentially means whitening out an area until white is encountered or blacking out an area until black is hit.

2. The PAINT statement can fill any figure, but painting very jagged edges or very complex figures can result in an "Out of memory" error. If this happens, you must use the CLEAR statement to increase the amount of available stack space.

11.2.4 POINT Function

Function

Returns the attribute value of the specified pixel from the screen.

Format

POINT (<x-coordinate>, <y-coordinate>)

Comment

1. In 40-column mode, valid results are 0 through 15. In 80-column mode, valid results are 0 through 3. In 300 x 800 mode, valid results are 0 through 1. If the specified pixel is out of range, a value of -1 is returned.

11.2.5 SCREEN Function

Function

Returns the ordinal or color attribute of the character at the specified row and column.

Format

`x = SCREEN (<row>, <column>[,z])`

<code>x</code>	is a numeric variable receiving the ordinal returned.
<code>row</code>	is a valid numeric expression returning an unsigned integer in the range 1 to 24.
<code>column</code>	is a valid numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending upon the current width.
<code>z</code>	is a valid numeric expression returning a Boolean result.

Comments

1. Any values entered outside the required ranges results in an "Illegal function call" error.
2. If the optional parameter "z" is given and is non-zero, the color attribute for the character is returned instead.

11.3 GRAPHICS INSTRUCTIONS

The graphics statements PSET, PRESET, CIRCLE, LINE and DRAW reference x-axis and y-axis pixel coordinates when producing graphic figures. 40-column mode yields 320 x 225 pixel resolution; 80-column mode yields 640 x 225 pixel resolution. The Wang Monochrome Monitor yields 800 x 300 resolution. You can use either the SCREEN or WIDTH command, as discussed in Sections 4.13 and 4.14, respectively, to switch between modes.

Coordinates can be expressed in either of two forms: absolute and relative. The absolute form of representing coordinates is the following:

(<absolute-x>, <absolute-y>)

The relative form of representing coordinates is the following:

STEP (<x-offset>, <y-offset>)

The absolute form is more common and directly refers to a point without regard to the last point referenced. The relative form references a point relative to the most recent point referenced.

Examples:

(10,10) - absolute form
STEP (10,0) - offset 10 in x and 0 in y
(0,0) - origin

In the formats and examples, the relative coordinate form is not shown. However, this form can be used wherever a coordinate is used. Note that all graphics statements and functions update the "most recent point used". For instance, in a LINE statement, if the relative coordinate form is used on the second coordinate, it is relative to the first coordinate. The only other way the "most recent point used" is changed is that the SCREEN and CLS statements initialize it to be the point in the middle of the screen: (320, 112) for 40-column mode; (160, 112) for 80-column mode; and (150, 400) for the Wang Monochrome Monitor. In each mode, (0, 0) references the upper left hand corner of the screen.

11.3.1 CIRCLE

Function

The CIRCLE statement draws an ellipse with the specified center and radius.

Format

```
CIRCLE (<x-center>, <y-center>), <radius>
      [, <attribute> [, <start>, <end> [, <aspect>]]]
```

x-center and y-center	specify the pixel coordinates of the center of the circle. The standard relative notation can also be used to specify the center point.
radius	indicates the pixel length of the radius of the circle. If the aspect ratio is between zero and 1, the radius is given in x-pixels. If it is greater than 1, the radius is given in y-pixels.
attribute	can be 0 to 15, with the default foreground color being 3 in 40-column mode and 1 in 80-column mode. If you specify a color outside the range of 0 to 3 in 80-column mode, BASIC looks at the two low-order bits of the value to determine the color.
start and end	are radian arguments between 0 and 2π which allow you to specify the angles where the drawing of the ellipse will begin and end. If the start or end angle is negative, the ellipse is connected to the center point with a line, and the angles are treated as if they were positive. (Note that this is different than adding 2π .) The start angle can be less than the end angle.
aspect	describes the ratio of the x-radius to the y-radius. The default aspect ratio is 5/6 in medium resolution and 5/12 in high resolution, and produces a visual circle in either graphics mode assuming a standard monitor screen aspect of 4/3. In many cases, an aspect ratio of 1 will produce nicer looking circles in 40-column mode. The circle will also be drawn faster.

11.3.2 LINEFunction

The LINE statement draws a line or rectangle, using the specified set(s) of coordinates.

Format

```
LINE [(x1, y1)] -[(x2, y2)][[,<attribute>] [,b[f]]]
```

(x1, y1) are starting pixel coordinates

(x2, y2) are ending pixel coordinates

attribute is a value in the range 0 to 15

Comments

1. If only -(x2, y2) is specified, the LINE statement draws a line from the "most recent point used" to point (x2, y2). The line is drawn in the current foreground color.

Example:

```
LINE -(319, 199)
```

If both (x1, y1) and (x2, y2) are specified, the LINE statement draws a line from point (x1, y1) to point (x2, y2) in the current foreground color.

Example:

```
LINE (0, 100) - (319, 100)
```

2. The attribute parameter can be specified to produce the line in color 0 to 15. If you specify a color outside the range of 0 to 3 in 80-column mode, BASIC looks at the two low-order bits of the value to determine the color.

Example:

```
10 CLS
20 LINE (RND * 319), (RND * 199), RND * 4
```

3. The LINE statement draws a rectangle with points (x1, y1) and (x2, y2) at opposite corners when the literal "b" or "bf" is included as the final argument. This avoids the need for the four LINE statements:

```
LINE (x1,y1) - (x2,y1)
LINE (x1,y1) - (x1,y2)
LINE (x2,y1) - (x2,y2)
LINE (x1,y2) - (x2,y2)
```


The "bf" also draws the same rectangle, but fills the interior points with the selected attribute.

4. When the coordinates given to the LINE statement are out of the legal range, a coordinate out of range is given the closest legal value.

11.3.3 PSET

Function

The PSET statement draws a point on the screen at the specified pixel.

Format

PSET (<x-coordinate>, <y-coordinate>) [, <attribute>]

x-coordinate, y-coordinate are coordinates of the point to be drawn.

attribute can be specified to produce the line in color 0 to 15. If you specify a color outside the range of 0 to 3 in 80-column mode, BASIC looks at the two low-order bits of the value to determine the color.

Comments

1. Note that when BASIC scans the value of the coordinates, it allows them to be beyond the edge of the screen. However, values outside the integer range cause an overflow error.
2. The attribute argument is optional. It defaults to the current graphics foreground color.

Example

```
5 REM draw a diagonal line to (100,100)
10 FOR i = 1 TO 100
20 PSET (i,i)
30 NEXT
35 REM clear out each line by setting each pixel to 0
40 FOR i = 100 TO 0 STEP -1
50 PSET (i,i), 0
60 NEXT
```


11.3.4 PRESET

Function

Like the PSET statement, PRESET also draws a point on the screen at the specified pixel coordinates. The only difference is that if the attribute parameter is not specified in the PRESET statement, the system automatically uses an attribute of 0. In other words, the system defaults to a background color, instead of a foreground color.

Format

PRESET (<x-coordinate>, <y-coordinate>) [,<attribute>]

x-coordinate, y-coordinate are coordinates of the point to be drawn.

attribute can be specified to produce the line in color 0 to 15. If you specify a color outside the range of 0 to 3 in 80-column mode, BASIC looks at the two low-order bits of the value to determine the color.

11.3.5 DRAWFunction

To draw an object determined by the specified string.

Format

DRAW <string-expression>

Comments

1. Each of the movement commands begin movement from the "current graphics position". This is usually the coordinate of the last graphics point plotted with another graphics statement. The current position defaults to the center of the screen when a program is run.

2. The following are the movement commands that compose a string in the DRAW statement. These commands move one unit if you do not supply an n argument.

U [n]	Move up (scale factor * n) points
D [n]	Move down
L [n]	Move left
R [n]	Move right
E [n]	Move diagonally up and right
F [n]	Move diagonally up and left
G [n]	Move diagonally down and left
H [n]	Move diagonally down and right

3. The following movement command permits absolute or relative movement. Note that you cannot mix absolute and relative movement commands.

M <X,Y>	In absolute mode, a line is drawn to point X,Y from the current position. Relative movement is specified by signed X and Y values. A line connects the current position with the position determined by the offset of the current position by +X or -X and +Y or -Y.
---------	--

4. The following commands can precede any of the above movement commands:

B	Move but don't plot any points.
N	Move but return to original position when done.
A <n>	Set angle n. n can range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so that they will appear the same size as with 0 or 180 degrees on a monitor screen with the standard aspect ratio of 4/3.

- C <n> Set attribute n. n represents the color of the image. n can range from 0 to 16.
- S <n> Set scale factor. n can range from 1 to 255. The scale factor multiplied by the distances given with U,D,L,R, or relative M commands to get the actual distance traveled.
- x <string> Execute substring (not supported by BASIC compiler). This powerful command allows you to execute a second substring from a string, much like GOSUB in BASIC. You can have one string execute another, which executes a third, and so on.

5. Numeric arguments can be constants like "123" or "=variable;", where variable is the name of a variable (not supported by BASIC compiler).

11.3.6 GET and PUTFunction

The PUT and GET statements transfer graphics images to and from the screen. When used together, the PUT and GET statements make animation and high-speed object motion possible in graphics mode.

Format (GET Statement)

GET (x1,y1)-(x2,y2), <array name>

Format (PUT Statement)

PUT (x1,y1), <array> [, <action verb>]

array	holds the screen image. The array can be of any type except string, and must be dimensioned large enough to hold the entire image. The contents of the array after a GET will be meaningless when interpreted directly (unless the array is of type integer - see below).
-------	---

action verb	is used to interact the transferred image with the image already on the screen. The following are the action verbs:
-------------	---

PSET transfers the data onto the screen verbatim.

PRESET is the same as PSET except that a negative image (black or white) is produced.

AND is used when you want to transfer the image only if an image already exists under the transferred image.

OR is used to superimpose the image onto the existing image.

XOR is a special mode often used for animation. XOR causes the points on the screen to be inverted where a point exists in the array image. This behavior is exactly like the cursor on the screen. XOR has a unique property that makes it especially useful for animation: when an image is PUT against a complex background once twice, the background is restored unchanged. This allows you to move an object around the screen without obliterating the background.

The default action mode is XOR.

Comments

1. The GET statement transfers the screen image bounded by the rectangle described by the specified points into an array. The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ",b" option, that is points (x1, y1) and (x2, y2) are at opposite corners of the rectangle.
2. The PUT statement transfers the image stored in the array onto the screen. The specified point is the coordinate of the top left corner of the image. An "Illegal Function call" error will result if the image to be transferred is too large to fit on the screen.
3. It is possible to GET an image in one screen mode and PUT it in another, although the effect may be quite strange because of the way points are represented in each mode.

In 40-column mode, AND, OR and XOR have the following effects on color:

AND					OR					XOR				
array	screen	attrib			array	screen	attrib			array	screen	attrib		
attrib	0	1	2	3	attrib	0	1	2	3	attrib	0	1	2	3
0	0	0	0	0	0	0	1	2	3	0	0	1	2	3
1	0	1	0	1	1	1	1	3	3	1	1	0	3	2
2	0	0	2	2	2	2	3	2	3	2	2	3	0	1
3	0	1	2	3	3	3	3	3	3	3	3	2	1	0

4. Animation of an object is usually performed as outlined below:
 - a. PUT the object(s) on the screen.
 - b. Recalculate the new position of the object(s).
 - c. PUT the object(s) on the screen a second time at the old location(s) to remove the old image(s).
 - d. Go to step one, this time PUTting the object(s) at the new location.

This method of animation leaves the background unchanged. You can minimize flicker by shortening the time between Step d and Step a, and by making sure that there is enough time delay between Steps a and c. If more than one object is to be animated, each object should be processed simultaneously, according to the above four steps.

5. If it is not important to preserve the background, you can animate images using the PSET action verb. This method leaves a border around the image as soon as it has gotten as large as the maximum distance the image will move. Thus, when an image is moved, this border will effectively erase any points. This method may be somewhat faster than the method using XOR described above since only one PUT is required to move an object (although you must PUT a larger image).

6. The storage format in the array is as follows:

2 bytes giving the x dimension in bits
2 bytes giving the y dimension
the array data

The data for each row of pixels is left justified on a byte boundary, so if there are less than a multiple of 8 bits stored, the rest of the byte will be filled with zeros. The required array size in bytes is the following:

$$4 + \text{INT}((x * \text{bitsperpixel} + 7) / 8) * y$$

where:

bitsperpixel is 2 in 40-column mode and 1 in 80-column mode

The number of bytes per element of an array are the following:

2 for integer
4 for single-precision
8 for double-precision

Example:

Say you want to GET a 10 x 12 image into an integer array. The number of bytes required is $4 + \text{INT}((10 * 2 + 7) / 8) * 12$ or 40 bytes. Thus, you would need an integer array with at least 20 elements.

It is possible to examine the X and Y dimensions and even the data in an array if an integer array is used. The X dimension is in element 0 of the array, and the Y dimension is in element 1. It must be remembered, however, that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

CHAPTER 12 SOUND AND MUSIC

12.1 INTRODUCTION

BASIC provides a number of instructions that produce sounds from the Wang PC keyboard speaker. The flexibility of the PLAY instruction and the range of the SOUND instruction enable the production of music.

12.2 BEEP

Function

The BEEP statement sounds the speaker at 800 Hertz for 1/4 second.

Format

BEEP

Example

2430 IF X < 20 THEN BEEP 'X is out of range, complain.

12.3 SOUND

Function

The SOUND statement generates sound through the speaker.

Format

SOUND <freq>, <duration>

freq	is the desired frequency in Hertz. A valid numeric expression returning an unsigned Integer in the range 37 to 32767.
------	---

duration	is the desired duration in Clock ticks. A valid numeric expression returning an unsigned Integer in the range 0 to 65535. Current clock ticks occur 18.2 times per second.
----------	--

Rules

1. If the duration is zero, any current SOUND statement that is running is turned off. If no SOUND statement is running, SOUND x,0 has no effect.

Example

2500 SOUND RND*1000=37,2 Create random sounds.

12.4 PLAY

Function

To produce a sequence of notes determined by the specified string.

Format

PLAY <string expression>

Comments

1. The single character commands in PLAY are:

A-G [# +,-]	Play the note. A "#" or "+" afterwards means sharp, and "-" means flat.
L <n>	Length - Sets the length of each note. L4 is a quarter note, L1 is a whole note, etc. n can range from 1 to 64. The length can also follow the note when it is desired to change the length only for the note. In this case, A16 is equivalent to L16A.
MF	Music Foreground. Music (PLAY statement) and SOUND are to run in Foreground. That is, each subsequent note or sound will not start until the previous note or sound is finished. This is the initial default.
MB	Music Background. Music (PLAY statement) and SOUND are to run in Background. That is, each note or sound is placed in the buffer allowing the BASIC program to continue execution while music plays in the "background". Up to 32 notes (or rests) can be played in background at a time.
MN	Music Normal. Each note will play 7/8ths of the time determined by L (length).
ML	Music Legato. Each note will play the full period set by L (length).
MS	Music Staccato. Each note will play 3/4ths of the time determined by L (length).
N <n>	Play note n. n can range from 0 to 84. In the 7 possible octaves, there are 84 notes. N=0 means rest.
O <n>	Octave - Sets the current octave. There are 7 octaves (0..6).

P <n> Pause. P can range from 1 to 64.

T <n> Tempo - Sets the number of L4's in the minute. n
can range from 32 to 255. Default is 120.

. Dot or Period. After each note causes the note to
play $3/2$ times the period of time determined by L
(length) times T (tempo). Multiple dots can appear
after the note. The duration is scaled
accordingly. (Ex: A. $3/2$, A.. $9/4$, A... $27/8$
etc.). Dots may appear after a pause (P) and scale
the pause length as described above.

X <string> Execute substring.

2. Because of the slow clock interrupt rate, some notes will not play at high tempos; e.g., L64 at T255. Which note/tempo combinations these are must be determined through experimentation.

APPENDIX A

ASCII CHARACTER CODES

Table A-1 lists the Wang Professional Computer character set.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SPACE	0	@	P	'	p	°	Â	â	Ğ	ğ	Þ	þ	£
1			!	1	A	Q	a	q	•	À	à	U	ü	Ð	ð	ƒ
2			"	2	B	R	b	r	•	Á	á	Í	í	Ý	ý	¥
3			#	3	C	S	c	s	•	Ä	ä	İ	ı	Ş	ş	¼
4			\$	4	D	T	d	t	•	Å	å	Î	î	'	'	½
5			%	5	E	U	e	u	•	—	—	Í	í	Û	û	¾
6			&	6	F	V	f	v	•	Ä	ä	Ï	ï	Ü	ü	•
7			'	7	G	W	g	w	•	ı	†	LL	ll	Ú	ú	•
8			(	8	H	X	h	x	•	Æ	æ	Ñ	ñ	Û	ü	•
9			)	9	I	Y	i	y	•	Ç	ç	Ô	ô	•	™	•
A			*	:	J	Z	j	z	•	†	□	Ô	ô	•	□	•
B			+	;	K	[	k	{	•	•	REQ. SPACE	Ó	ó	R	←	•
C			,	<	L	\	l		•	Ê	ê	Ö	ö	•	•	•
D			-	=	M	]	m	}	•	±	È	è	Õ	õ	«	»
E			.	>	N	^	n	~	•	É	é	Æ	æ	§	ß	•
F			/	?	O	_	o	•	•	Ë	ë	Ø	ø	¶	•	•

Table A-1. Character Set

The following list summarizes Table A-1 and provides the decimal value of each character. The decimal value is used in combination with the CHR\$ function and the PRINT and LPRINT statements to produce special characters and perform special operations on the screen and printer (refer to Subsection 9.3.2 for additional information).

Decimal	Hex	Character or Explanation
000	00	Null
001	01	Edit line where error occurred
002	02	Previous word
003	03	Stop program and return to BASIC command level
004	04	
005	05	Erase from cursor to end of logical line
006	06	Next word
007	07	Sound alarm
008	08	Destructive backspace
009	09	Tab
010	0A	Line feed
011	0B	Home cursor
012	0C	Clear screen and home cursor
013	0D	Carriage return
014	0E	Append to end of line
015	0F	
016	10	
017	11	Resume program listing on screen or printer
018	12	Toggle insert mode
019	13	Suspend program listing on screen or printer
020	14	Activate and update function key display
021	15	Clear logical line
022	16	
023	17	Delete word
024	18	
025	19	
026	1A	Clear from cursor to end of screen
027	1B	Cursor right / Escape
028	1C	Cursor left
029	1D	Cursor up
030	1E	Cursor down
031	1F	Delete character at cursor location
032	20	Space
033	21	Exclamation point
034	22	Double quote "
035	23	Number sign #
036	24	Dollar sign \$
037	25	Percent sign %
038	26	Ampersand &
039	27	Single quote '
040	28	Left parenthesis (
041	29	Right parenthesis)

Decimal	Hex	Character or Explanation
042	2A	Asterisk *
043	2B	Plus sign +
044	2C	Comma ,
045	2D	Minus sign -
046	2E	Period .
047	2F	Slash /
048	30	0
049	31	1
050	32	2
051	33	3
052	34	4
053	35	5
054	36	6
055	37	7
056	38	8
057	39	9
058	3A	Colon :
059	3B	Semicolon ;
060	3C	Less than sign <
061	3D	Equal sign =
062	3E	Greater than sign >
063	3F	Question mark ?
064	40	At sign @
065	41	A
066	42	B
067	43	C
068	44	D
069	45	E
070	46	F
071	47	G
072	48	H
073	49	I
074	4A	J
075	4B	K
076	4C	L
077	4D	M
078	4E	N
079	4F	O
080	50	P
081	51	Q
082	52	R
083	53	S
084	54	T
085	55	U
086	56	V
087	57	W
088	58	X
089	59	Y
090	5A	Z

Decimal	Hex	Character or Explanation
091	5B	Left bracket [
092	5C	Backslash
093	5D	Right bracket]
094	5E	Up arrow Ü
095	5F	Underscore _
096	60	Open quote
097	61	a
098	62	b
099	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	Left brace
124	7C	Vertical bar
125	7D	Right brace
126	7E	Approximate symbol
127	7F	Cent sign
128	80	Degree sign
129	81	Center graphic
130	82	Tab graphic
131	83	Return graphic
132	84	Indent graphic
133	85	Decimal tabulation graphic
134	86	Format graphic
135	87	Stop graphic
136	88	Note graphic
137	89	Merge graphic
138	8A	Down arrow
139	8B	Up arrow

Decimal	Hex	Character or Explanation
140	8C	Left arrow
141	8D	Plus-or-minus sign
142	8E	Inverted exclamation
143	8F	Inverted question mark
144	90	A circumflex
145	91	A grave
146	92	A acute
147	93	A umlaut
148	94	A tilde
149	95	Left arrow
150	96	A angstrom
151	97	Down arrow
152	98	Ligature AE
153	99	C cedilla
154	9A	Double dagger
155	9B	Bullet
156	9C	E circumflex
157	9D	E grave
158	9E	E acute
159	9F	E umlaut
160	A0	a circumflex
161	A1	a grave
162	A2	a acute
163	A3	a umlaut
164	A4	a tilde
165	A5	Right arrow
166	A6	a angstrom
167	A7	Dagger
168	A8	Ligature ae
169	A9	c cedilla
170	AA	Ballot box
171	AB	Required space
172	AC	e circumflex
173	AD	e grave
174	AE	e acute
175	AF	e umlaut
176	B0	G hacek
177	B1	IJ ligature
178	B2	Dotted I
179	B3	I circumflex
180	B4	I grave
181	B5	I acute
182	B6	I umlaut
183	B7	Catalonian LL ligature
184	B8	N tilde
185	B9	O circumflex
186	BA	O grave
187	BB	O acute
188	BC	O umlaut

Decimal	Hex	Character or Explanation
189	BD	O tilde
190	BE	OE ligature
191	BF	slashed O
192	C0	G hacek
193	C1	ij ligature
194	C2	dotless i
195	C3	i circumflex
196	C4	i grave
197	C5	i acute
198	C6	i umlaut
199	C7	Catalonian ll ligature
200	C8	n tilde
201	C9	o circumflex
202	CA	o grave
203	CB	o acute
204	CC	o umlaut
205	CD	o tilde
206	CE	oe ligature
207	CF	slashed o
208	D0	Icelandic Thorn
209	D1	Icelandic Eth
210	D2	Y acute
210	D3	S cedilla
211	D4	Open Quote
212	D5	u circumflex
213	D6	u grave
214	D7	u acute
215	D8	u umlaut
216	D9	Copyrighted symbol
217	DA	Registered Trademark symbol
218	DB	Prescription symbol
219	DC	a superior (for Number abbrev.)
220	DD	European open quotes
221	DE	section symbol
222	DF	paragraph symbol
223	E0	Icelandic thorn
224	E1	Icelandic eth
225	E2	y acute
226	E3	s cedilla
227	E4	close quote
228	E5	u circumflex
229	E6	u grave
230	E7	u acute
231	E8	u umlaut
232	E9	Trademark symbol
233	EA	International Monetary Symbol
234	EB	double arrow
235	EC	o superior (for Number abbrev.)

Decimal	Hex	Character or Explanation
236	ED	European close quotes
237	EE	Beta
238	EF	centered dot
239	F0	pound sterling
240	F1	florin
241	F2	yen
242	F3	1/4
243	F4	1/2
244	F5	3/4
245	F6	accent circumflex
246	F7	accent grave
247	F8	accent acute
248	F9	accent umlaut (diaeresis)
249	FA	accent tilde
250	FB	accent cedilla
251	FC	accent hacek
252	FD	accent breve
253	FE	5 unit space
254	FF	Dead key

APPENDIX B

ERROR MESSAGES AND CODES

B.1 INTRODUCTION

Error messages and error codes have equal importance in Wang PC Interpreted BASIC. The system displays an error message when it encounters an error during program execution. Two BASIC language elements directly reference error codes. The ERR variable contains the numeric code of the most recent error, and is used in conjunction with the ERR variable to set error traps. Among the functions of the ERROR statement is to simulate the occurrence of an error condition, when the statement specifies the numeric code of a particular error.

This appendix lists BASIC errors sequentially according to numeric code. Table B-1 lists the BASIC errors in alphabetical order, as well as the corresponding error codes.

Table B-1. Error Codes and Messages

Bad file mode	54	Input past end	62
Bad file name	64	Line buffer overflow	23
Bad file number	52	Missing operand	22
Bad record number	63	NEXT without FOR	01
Can't continue	17	No RESUME	19
Communication buffer overflow	69	Out of data	04
Device unavailable	68	Out of memory	07
Direct statement in file	66	Out of string space	14
Disk full	61	Overflow	06
Disk I/O error	57	RESUME without error	20
Disk media error	72	Redimensioned array	10
Disk write protect	70	RETURN without GOSUB	03
Disk not ready	71	String formula too complex	16
Division by zero	11	String too long	15
Field overflow	50	Subscript out of range	09
File already exists	58	Syntax error	02
File already open	55	Too many files	67
File not found	53	Type mismatch	13
FOR without NEXT	26	Undefined line	08
Illegal direct	12	Undefined user function	18
Illegal function call	05	Unprintable error	21
Internal error	51	WEND without WHILE	30
		WHILE without WEND	29

B.2 ERROR MESSAGES AND RECOVERY

The following are the BASIC error messages.

01 NEXT without FOR

A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.

02 Syntax error

A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).

03 RETURN without GOSUB

A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.

04 Out of data

A READ statement is executed when there are no DATA statements with unread data remaining in the program.

05 Illegal function call

A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of any of the following:

- a negative or unreasonable large subscript
- a negative or zero argument with LOG
- a negative argument to SQR
- a negative mantissa with a non-integer exponent
- a call to a USR function for which the starting address has not yet been given
- an improper argument to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, PIKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON...GOTO.

06 Overflow

The result of a calculation is too large to be represented in BASIC's number format. If underflow occurs, the result is zero and execution continues without an error.

07 Out of memory

A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

08 Undefined line

A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.

09 Subscript out of range

An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

10 Redimensioned array

Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero

A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

12 Illegal direct

A statement that is illegal in direct mode is entered as a direct mode command.

13 Type mismatch

A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

14 Out of string space

String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.

15 String too long

An attempt is made to create a string more than 255 characters long.

16 String formula too complex

A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 Can't continue

An attempt is made to continue a program that has halted due to an error, has been modified during a break in execution, or does not exist.

18 Undefined user function

A USR function is called before the function definition (DEF statement) is given.

19 No RESUME

An error trapping routine is entered but contains no RESUME statement.

20 RESUME without error

A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error

An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand

An expression contains an operator with no operand following it.

23 Line buffer overflow

An attempt is made to input a line that has too many characters.

26 FOR without NEXT

A FOR was encountered without a matching NEXT.

29 WHILE without WEND

A WHILE statement does not have a matching WEND.

30 WEND without WHILE

A WEND was encountered without a matching WHILE.

50 Field overflow

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error

An internal malfunction has occurred. Please contact your Wang Service Representative.

52 Bad file number

A statement or command references a file with a file number that was not opened or is out of the range of file numbers specified at initialization.

53 File not found

A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode

An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN statement with a file mode other than I, O, or R.

55 File already open

A sequential output mode OPEN statement is issued for a file that is already open; or a KILL is given for a file that is already open.

57 Disk I/O error

An I/O error occurred on a disk I/O operation. It is a fatal error from which the operating system cannot recover.

58 File already exists

The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 Disk full

All disk storage space is in use.

62 Input past end

An INPUT statement is executed after all data in the file has been input, or for an empty file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number

In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

64 Bad file name

An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

66 Direct statement in file

A direct statement is encountered while loading an ASCII-format file. The LOAD is terminated.

67 Too many files

An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

68 Device unavailable

An attempt was made to open a file to a non-existent device. It may be that hardware did not exist to support the device, such as LPT2: or LPT3:, or was disabled by the user. This occurs if an OPEN "COM1:..." statement is executed but the user disabled RS232 support via the /C:0 switch directive on the command line.

69 Communication buffer overflow

Occurs when a communication input statement is executed but the input queue was already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs will attempt to clear this fault unless characters continue to be received faster than the program can process them. In this case several options are available:

- Increase the size of the COM receive buffer via the /C: switch.
- Implement a "hand-shaking" protocol with the host/satellite such as XON/SOFF as demonstrated in the TTY programming example to turn transmit off long enough to catch up.
- Use a lower Baud rate for transmit and receive.

70 Disk write protect

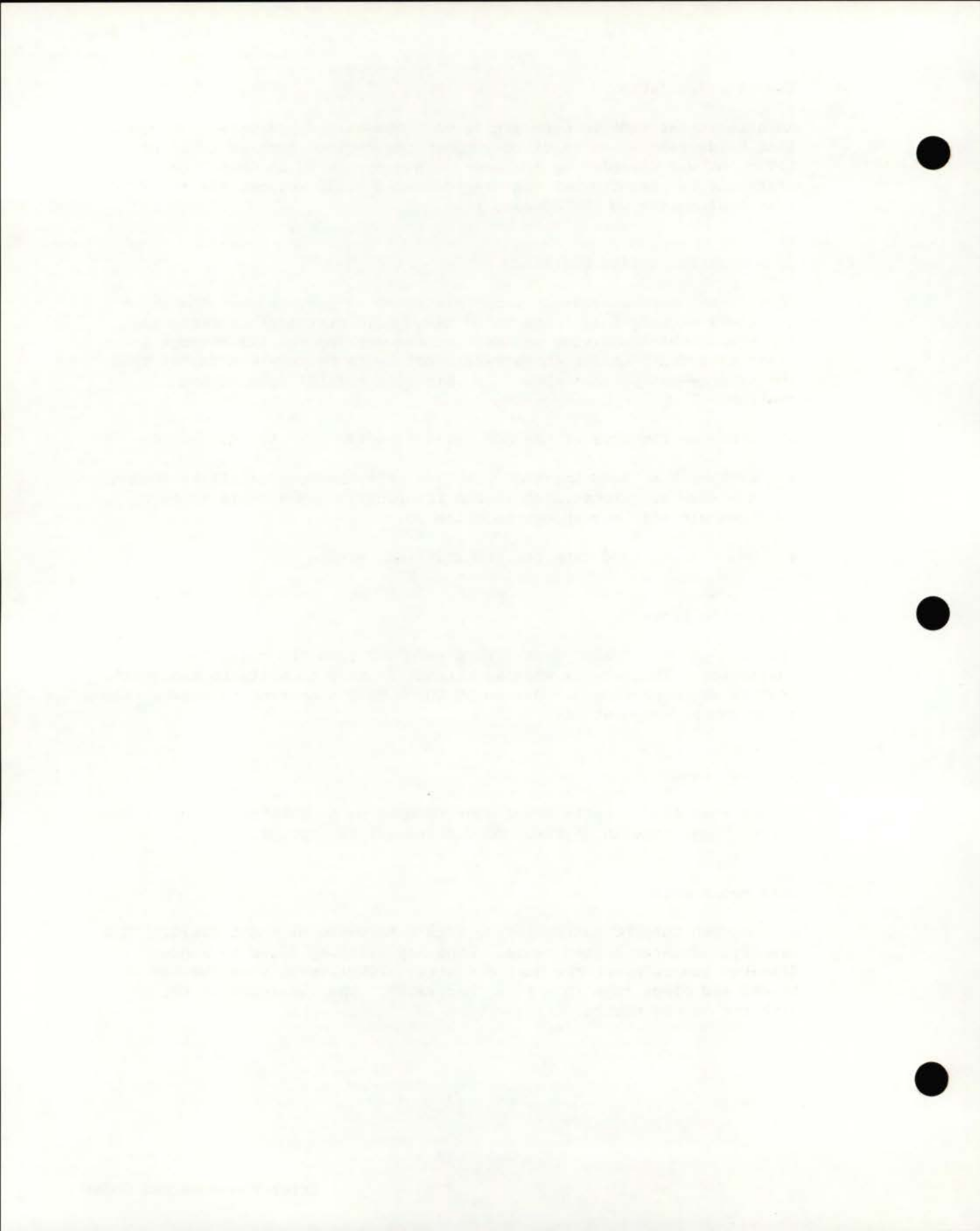
This is one of 3 "hard" disk errors returned from the diskette controller. This occurs when an attempt is made to write to a diskette that is write protected. Use an ON ERROR GOTO statement to detect this situation and request user action.

71 Disk not ready

Occurs when the diskette drive door is open or a diskette is not in the drive. Again use an ON ERROR GOTO statement to recover.

72 Disk media error

Occurs when the FDC controller detects a hardware or media fault. This usually indicates harmed media. Copy any existing files to a new diskette and reformat the hurt diskette. FORMAT will flag the bad tracks and place them in a file "badtrack". The remainder of the diskette is now usable.



APPENDIX C RESERVED WORDS

Reserved words have special meaning to BASIC, and cannot be used as variable names. Reserved words include all BASIC commands, statements, function names, and operator names. You should always separate reserved words from other parts of a BASIC statement by using spaces or other special characters as indicated in the general format.

The following is a list of BASIC reserved words.

ABS	DELETE
AND	DIM
ASC	DRAW
ATN	EDIT
AUTO	ELSE
BEEP	END
BLOAD	EOF
BSAVE	EQV
CALL	ERASE
CDBL	ERL
CHAIN	ERR
CHRS	ERROR
CINT	EXP
CIRCLE	FIELD
CLEAR	FILES
CLOSE	FIX
CLS	FNxxxxxxxx
COLOR	FOR
COM	FRE
COMMON	GET
CONT	GOSUB
COS	GOTO
CSING	HEX\$
CSRLIN	IF
CVD	IMP
CVI	INKEYS
CVS	INP
DATA	INPUT
DATES	INPUT#
DEF	INPUTS
DEFDBL	INSTR
DEFINT	INT
DEFSNG	KEY
DEFSTR	KILL

LEFTS
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID\$
MKD\$
MKI\$
MKS\$
MOD
MOTOR
NAME
NEW
NEXT
NOT
OCT\$
OFF
ON
OPEN
OPTION
OR
OUT
PALETTE
PAINT
PEEK
PEN
PLAY
POINT
POKE
POS
PRESET
PRINT
PRINT#
PSET
PUT

RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHTS
RNDPM
RSET
RUN
SAVE
SCREEN
SGN
SIN
SOUND
SPACES
SPC(
SOR
STEP
STICK
STOP
STR\$
STRIG
STRING\$
SWAP
SYSTEM
TAB(
TAN
THEN
TIMES\$
TO
TROFF
TRON
USING
USR
VAL
VARPTR
WAIT
WEND
WHILE
WIDTH
WRITE
WRITE#
XOR

INDEX

A

ABS, 9-2
Absolute value, 9-1, 9-2
Addition, 1-8
AND, 1-10 to 1-12
Arctangent, 9-3
Arithmetic functions, 9-1
Arithmetic operators, 1-8
Array variables, 1-5
Arrays, 7-3, 7-13, 7-14
ASC, 9-19
ASCII codes, A-1
Aspect, 11-11
Assembly, 6-2, 7-12, 7-27, 9-50
ATN, 9-3
AUTO, 3-3, 4-1, 4-2
Automatic line numbering, 3-3

B

Background, 11-2, 11-6
Backspace, 3-7
BASIC keywords, 3-4
BEEP, 12-1
Boolean, 1-10

C

CALL, 6-2 to 6-4
CDBL, 9-4
CHAIN, 6-4, 6-5
CHR\$, 9-20
CINT(X), 9-5
CIRCLE, 11-11
CLEAR, 7-1, 7-2
Clear screen, 3-7
CLOSE, 8-10
CLS, 4-3
Color, 11-2
Color palette, 11-2
COLOR statement, 11-6
Command initialization, 2-6
COMMON, 7-3 to 7-4
Comparisons, 1-13
Constants, 1-2, 1-3
CONT, 3-10, 10-1, 10-2

Conversion, variable type 1-5

Converting:

ASCII code to string, 9-20
character to ASCII code, 9-19
double to single-precision, 9-7
number to string, 9-13
numeric to hexadecimal, 9-22
numeric to octal, 9-29
single to double-precision, 9-4
string to numeric, 9-21

Coordinates, 11-7, 11-10

COS, 9-6

Cosine, 9-6

CSNG, 9-7

CSRLIN, 9-43

CTRL, 3-5 to 3-10

Cursor, 3-6 to 3-10

Cursor position, 3-8

CVD, 9-21

CVI, 9-21

CVS, 9-21

CVX, 9-21

D

DATA, 7-39 to 7-41, 7-43

DATE\$, 7-7

DEF FN, 6-5

DEF SEG, 7-11

DEF USR, 7-12

DEFDBL, 7-10

DEFINT, 7-10

DEFSNG, 7-10

DEFSTR, 7-10

DELETE, 3-6, 4-4, 4-5

Device filenames, 5-2

DIM, 7-13

Direct mode, 1-1, 3-1, 3-2

Disk directory, 3-11

Division, 1-8, 1-9

Division by zero, 1-9

Double-precision, 1-3

DRAW, 11-16

Drive designation, 5-1

INDEX (continued)

E

EDIT, 3-6, 4-6 to 4-8
 Edit mode, 4-6 to 4-8
 Editor, 3-6
 ELSE, 6-13 to 6-15
 END, 6-6
 End of file, 9-35
 EOF, 9-35
 ERASE, 7-14
 Erasing text, 3-9
 ERL, 7-15
 ERR, 7-15
 ERROR, 6-16, 7-16, 7-17
 Error codes, B-1
 Error trapping, 6-16
 Escape, 4-6
 Event trapping, 6-17, 6-18, 11-10
 Executing a program, 6-18
 Exiting BASIC, 2-1, 2-6, 2-7
 EXP, 9-8
 Exponentiation, 1-8, 1-9
 Expressions, 1-7, 1-8, 1-13

F

FIELD, 8-11 to 8-13
 File Control Block (FCB), 9-40
 File specification, 5-1
 Filename, 1-13, 5-1, 5-2
 Filename extension,
 1-14, 5-1, 5-2
 Files,
 1-14, 1-15, 8-1, 8-4, 8-5,
 8-7, 8-9, 8-10
 FILES (statement), 4-9
 FIX, 9-9
 FOR, 6-7 to 6-9
 FOR...NEXT, 6-7 to 6-9
 Foreground, 11-2, 11-6
 FRE, 9-10
 Functions, 1-12

G

GET, 11-17 to 11-20
 GL, 3-4
 GOSUB, 6-10, 6-11, 6-19
 GOTO, 6-12 to 6-14, 6-16, 6-19
 Graphics, 11-1, 11-10

H

HEX\$, 9-22
 Hexadecimal, 9-22

I

IF, 6-13, 6-14
 IF...THEN, 6-13, 6-14
 Indirect mode, 1-1, 3-1, 3-2
 INKEY\$, 9-23
 INP, 9-36
 INPUT, 7-18, 7-19
 INPUT#, 8-14
 INPUT\$, 9-24
 Insert mode, 3-9
 Inserting text, 3-9
 INSTR, 9-25
 INT, 9-11
 Integer division, 1-9

K

KEY, 4-10, 4-11
 KILL, 5-3

L

LEFT\$, 9-26
 LEN, 9-27
 LET, 7-20
 LINE, 11-12, 11-13
 LINE INPUT, 7-21, 7-22
 LINE INPUT#, 8-15
 Line numbers, 3-2
 Lines, 3-2 to 3-6
 LIST, 3-5, 3-10, 4-13
 LLIST, 3-10, 4-13
 LOAD, 5-4
 Loading BASIC, 2-1 to 2-5
 LOC, 9-37
 LOCATE, 4-15
 LOF, 9-38
 LOG, 9-12
 Logarithm, 9-12
 Logical operators, 1-10, 1-11
 Loops, 6-8, 6-18, 6-24, 6-25
 LPOS, 9-44
 LPRINT, 7-23
 LPRINT USING, 7-23
 LSET, 8-16

INDEX (continued)

M

Main System Menu,
 2-1, 2-2, 2-4, 2-7
 MERGE, 5-6, 5-7
 Messages, 3-9
 MID\$, 9-28
 MKD\$, 9-13
 MKI\$, 9-13
 MKS\$, 9-13
 Modulus arithmetic, 1-9
 Multiplication, 1-8

N

NAME, 5-8
 Negation, 1-8
 NEW, 3-3
 NEXT, 6-8
 NOT, 1-10 to 1-17
 NULL, 7-24
 Numeric constants, 1-2, 1-3
 Numeric expressions, 9-1
 Numeric variables, 7-1

O

OCT\$, 9-29
 Octal, 9-29
 ON ERROR, 6-16
 ON ERROR GOTO, 6-16
 ON...GOSUB, 6-19
 ON...GOTO, 6-19
 OPEN, 8-17 to 8-20
 Operators, 1-6 to 1-13
 OPTION BASE, 7-25
 OR, 1-10 to 1-12
 OUT, 7-26
 Overflow, 1-9, B-3

P

PAINT, 11-7
 Palette, see Color palette
 PALETTE (statement), 11-3, 11-4
 PEEK, 9-39
 PEN, 11-10
 PLAY, 12-3, 12-4
 POINT, 9-45, 11-8
 POKE, 7-27
 POS, 9-46

PRESET, 11-15
 PRINT, 7-28 to 7-32
 PRINT USING, 7-33 to 7-37
 PRINT#, 8-21 to 8-22
 PRINT# USING, 8-3, 8-21, 8-22
 Program Development Menu,
 2-1 to 2-3, 2-6
 Program lines, 3-2 to 3-4
 Protected files
 1-14, 1-15, 8-9
 PSET, 11-14
 PUT, 8-23

R

Random access files, 1-14, 8-1
 Random number generator,
 7-38, 9-14
 RANDOMIZE, 7-38
 READ, 7-39, 7-40
 Relational operators, 1-9, 1-10
 REM, 7-41, 7-42
 Remarks, 7-41
 RENUM, 3-4, 4-17
 Renumbering program lines, 3-3
 Reserved words, 1-4, 3-4
 RESET, 2-7
 Restart procedure, 2-7
 RESTORE, 7-43
 RESUME, 3-10, 6-20, 6-21,
 RETURN, 6-10, 6-11
 RIGHT\$, 9-30
 RND, 9-14
 Rounding, 7-36, 9-1, 9-5
 RSET, 8-16
 RUN, 5-10

S

SAVE, 5-9
 SCREEN, 4-19, 9-47, 11-9
 Sequential files,
 1-14, 8-1, 8-4
 SGN, 9-15
 SIN, 9-16
 Sine, 9-16
 Single-precision, 1-3
 Soft Keys, 4-10, 4-11
 SOUND, 12-2, 12-3
 Space bar, 4-7
 SPACE\$, 9-31

INDEX (continued)

SPC, 9-48
SQR, 9-17
STICK, 11-10
STOP, 3-10, 6-22
STR\$, 9-32
STRIG, 6-17, 6-18
String comparisons, 1-13
String constants, 1-2
String expressions, 9-1
String functions, 9-19
String space, 7-1, 7-2
String variables, 1-4
STRING\$, 9-33
Subroutines, 6-10, 6-16
Subscripts, 1-5
Subtraction, 1-8
SWAP, 7-44
Syntax errors, 6-16
SYSTEM, 2-7
System Menu, see Main System Menu

T

TAB, 9-49
Tab key, 3-8
TAN, 9-18
Tangent, 9-18
THEN, 6-13 to 6-15
TIMES\$, 7-45, 7-46
TRACE, 3-11, 10-3
Tracing program execution, 3-11
TROFF, 3-11, 10-3
TRON, 3-11, 10-3
Two's complement, 1-10, 1-12
Type declaration characters, 1-4

U

USR, 9-50, 9-51

V

VAL, 9-34
Variable names, 1-4
Variable type conversion, 1-5
Variables, 1-3 to 1-5
VARPTR, 9-40 to 9-42

W

WAIT, 1-14, 6-24
WHILE...WEND, 6-25
WIDTH, 4-20, 4-21
WRITE, 7-47
WRITE#, 8-24

NOTES

NOTES

WANG**Customer Comment Form**Publication Number **700-8019**

Title _____

THE WANG PC BASIC LANGUAGE GUIDE

Help Us Help You . . .

We've worked hard to make this document useful, readable, and technically accurate. Did we succeed? Only you can tell us! Your comments and suggestions will help us improve our technical communications. Please take a few minutes to let us know how you feel.

How did you receive this publication?

- | | |
|--|--------------------------------------|
| ☐ Support or Sales Rep | ☐ Don't know |
| ☐ Wang Supplies Division | ☐ Other _____ |
| ☐ From another user | _____ |
| ☐ Enclosed with equipment | _____ |

How did you use this Publication?

- | | |
|--|--|
| ☐ Introduction to the subject | ☐ Aid to advanced knowledge |
| ☐ Classroom text (student) | ☐ Guide to operating instructions |
| ☐ Classroom text (teacher) | ☐ As a reference manual |
| ☐ Self-study text | ☐ Other _____ |

Please rate the quality of this publication in each of the following areas.

	EXCELLENT	GOOD	FAIR	POOR	VERY POOR
Technical Accuracy — Does the system work the way the manual says it does?	☐	☐	☐	☐	☐
Readability — Is the manual easy to read and understand?	☐	☐	☐	☐	☐
Clarity — Are the instructions easy to follow?	☐	☐	☐	☐	☐
Examples — Were they helpful, realistic? Were there enough of them?	☐	☐	☐	☐	☐
Organization — Was it logical? Was it easy to find what you needed to know?	☐	☐	☐	☐	☐
Illustrations — Were they clear and useful?	☐	☐	☐	☐	☐
Physical Attractiveness — What did you think of the printing, binding, etc?	☐	☐	☐	☐	☐

Were there any terms or concepts that were not defined properly? ☐ Y ☐ N If so, what were they? _____After reading this document do you feel that you will be able to operate the equipment/software? ☐ Yes ☐ No
☐ Yes, with practice

What errors or faults did you find in the manual? (Please include page numbers) _____

Do you have any other comments or suggestions? _____

Name _____ Street _____

Title _____ City _____

Dept/Mail Stop _____ State/Country _____

Company _____ Zip Code _____ Telephone _____

Thank you for your help.

WANG

Fold



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS

PERMIT NO. 16

LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

**WANG LABORATORIES, INC.
CHARLES T. PEERS, JR., MAIL STOP 1363
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851**



Fold

Cut along dotted line.

WANG

The completed order form should be mailed to:

WANG LABORATORIES, INC.
Supplies Division
51 Middlesex St.
No. Chelmsford MA 01863

To Order by Phone, Call:

(800)225-0234

From Mass., Hawaii, and Alaska

(617)256-1400**TELEX 951-743****Order Form for Wang Manuals and Documentation**

① Customer Number (If Known) _____				
② Bill To: _____			Ship To: _____	
_____			_____	
_____			_____	
_____			_____	
③ Customer Contact: _____			④ Date _____ Purchase Order Number _____	
() () _____			_____	
Phone _____ Name _____			_____	
⑤ Taxable Yes ☐ No ☐ ⑥ Tax Exempt Number _____ ⑦ Credit This Order to A Wang Salesperson Please Complete _____				
			Salesperson's Name _____ Employee No. _____ RDB No. _____	
⑧ Document Number	Description	Quantity	⑨ Unit Price	Total Price
⑩ _____ Authorized Signature _____ Date _____ ☐ Check this box if you would like a free copy of the Corporate Publications Literature Catalog (700-5294)			Sub Total	
			Less Any Applicable Discount	
			Sub Total	
			Local/State Tax	
			Total Amount	

Ordering Instructions

1. If you have purchased supplies from Wang before, and know your Customer Number, please write it here.
2. Provide appropriate Billing Address and Shipping Address.
3. Please provide a phone number and name, should it be necessary for WANG to contact you about your order.
4. Your purchase order number and date.
5. Show whether order is taxable or not.
6. If tax exempt, please provide your exemption number.
7. If you wish credit for this order to be given to a WANG salesperson, please complete.
8. Show part numbers, description and quantity for each product ordered.
9. Pricing extensions and totaling can be completed at your option; Wang will refigure these prices and add freight on your invoice.
10. Signature of authorized buyer and date.

Wang Supplies Division Terms and Conditions

1. **TAXES** — Prices are exclusive of all sales, use, and like taxes.
2. **DELIVERY** — Delivery will be F.O.B. Wang's plant. Customer will be billed for freight charges; and unless customer specifies otherwise, all shipments will go best way surface as determined by Wang. Wang shall not assume any liability in connection with the shipment nor shall the carrier be construed to be an agent of Wang. If the customer requests that Wang arrange for insurance the customer will be billed for the insurance charges.
3. **PAYMENT** — Terms are net 30 days from date of invoice. Unless otherwise stated by customer, partial shipments will generate partial invoices.
4. **PRICES** — The prices shown are subject to change without notice. Individual document prices may be found in the Corporate Publications Literature Catalog (700-5294)
5. **LIMITATION OF LIABILITY** — In no event shall Wang be liable for loss of data or for special, incidental or consequential damages in connection with or arising out of the use of or information contained in any manuals or documentation furnished hereunder.

WANG

Fold



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 NO. CHELMSFORD, MA.

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
Supplies Division
c/o Order Entry Dept.
M/S 5511
51 Middlesex St.
No. Chelmsford, MA 01863



Fold

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL: 617/459-5000, TWX 710-343-6769, TELEX 94-7421

WANG

Printed in U.S.A.
700-8019
9-83-61M