AA–P602B–TV

Rainbow™ 100

MBASIC™-86

Reference Manual

Developed by Microsoft Corporation

digital equipment corporation

First Printing, November 1982
Revised, June 1983

© Digital Equipment Corporation 1982, 1983. All Rights Reserved.

Portions of this document are reproduced with the permission of Microsoft Corporation.
© Microsoft Corporation 1977, 1978, 1979, 1980, 1981, 1982. All Rights Reserved.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

CP/M® is a registered trademark of Digital Research Inc. CP/M®–80 and CP/M®–86 are trademarks of Digital Research Inc.

MBASIC and Multiplan are trademarks of Microsoft Corporation.

The following are trademarks of Digital Equipment Corporation:

digital™

DEC	MASSBUS	UNIBUS
DECmate	PDP	VAX
DECsystem–10	P/OS	VMS
DECSYSTEM–20	Professional	VT
DECUS	Rainbow	Work Processor
DECwriter	RSTS	
DIBOL	RSX	

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

Printed in U.S.A.

Contents

Preface vii

Chapter 1. General Information About MBASIC-86 1

Modes of Operation 1
Line Format 2
Character Set 2
Constants 4
Variables 6
Type Conversion 8
Expressions and Operators 10
Input Editing 18
Error Messages 19

Chapter 2. MBASIC-86 Commands and Statements 21

Format Notation 21
AUTO 22
CALL 23
CHAIN 24
CLEAR 26
CLOSE 27

COMMON 28
CONT 30
DATA 31
DEF FN 32
DEFINT/SNG/DBL/STR 34
DEF SEG 35
DEF USR 36
DELETE 37
DIM 38
EDIT 39
END 44
ERASE 45
ERR and ERL Variables 46
ERROR 47
FIELD 49
FILES 50
FOR . . . NEXT 51
GET 54
GOSUB . . . RETURN 55
GOTO 56
IF . . . THEN(. . . ELSE) and IF . . . GOTO 57
INPUT 59
INPUT# 61
KILL 62
LET 63
LINE INPUT 64
LINE INPUT# 65
LIST 66
LLIST 68
LOAD 69
LPRINT and LPRINT USING 70
LSET and RSET 71
MERGE 72
MID$ 73
NAME 74
NEW 75
NULL 76
ON ERROR GOTO 77
ON . . . GOSUB and ON . . . GOTO 78
OPEN 79
OPTION BASE 81
OUT 82

POKE 83
PRINT 84
PRINT USING 87
PRINT# and PRINT# USING 92
PUT 95
RANDOMIZE 96
READ 97
REM 99
RENUM 100
RESET 102
RESTORE 103
RESUME 104
RUN 105
SAVE 106
STOP 107
SWAP 108
SYSTEM 109
TRON/TROFF 110
WAIT 111
WIDTH 112
WHILE . . . WEND 113
WRITE 114
WRITE# 115

Chapter 3. MBASIC-86 Functions 117

ABS 118
ASC 118
ATN 119
CDBL 119
CHR$ 120
CINT 120
COS 121
CSNG 121
CVI, CVS, CVD 122
EOF 123
EXP 123
FIX 124
FRE 124
HEX$ 125
INKEY$ 126
INP 126

INPUT$ 127
INSTR 128
INT 128
LEFT$ 129
LEN 129
LOC 130
LOF 130
LOG 131
LPOS 131
MID$ 132
MKI$, MKS$, MKD$ 133
OCT$ 134
PEEK 134
POS 135
RIGHT$ 135
RND 136
SGN 136
SIN 137
SPACE$ 137
SPC 138
SQR 138
STR$ 139
STRING$ 139
TAB 140
TAN 140
USR 141
VAL 141
VARPTR 142

Appendix A. Summary of Error Codes and Error Messages 145

Appendix B. Mathematical Functions 151

Appendix C. ASCII Character Codes 153

Index 157

Preface

Intended Reader

This guide is intended as a reference guide for the MBASIC-86 user.

Before reading this guide, you should read the *MBASIC-86 User's Guide.*

Guide Organization

This guide discusses each MBASIC-86 command, statement, and function. The guide is organized as follows:

Chapter 1 contains general information about MBASIC-86.

Chapter 2 contains detailed information about each MBASIC-86 command and statement, organized alphabetically.

Chapter 3 contains detailed information about each MBASIC-86 function, organized alphabetically.

Appendix A lists MBASIC-86 error messages and codes.

Appendix B lists mathematical functions and the MBASIC-86 equivalents.

Appendix C contains a chart of ASCII character codes.

Conventions Used

- In examples between you and the computer, what the computer displays on the screen is shown in black. Characters you type from the keyboard are shown in color.
- Input you enter when you run an MBASIC-86 program is shown in color.
- Be sure to type all spaces and punctuation marks exactly as they are printed.
- All command, statement, and function names are in uppercase.
- When you see <Return>, press the Return key on the keyboard.
- When you see Ctrl/C, hold the control key (Ctrl key on the keyboard) while you press the C key. Be sure to hold both keys down at the same time.
- Square brackets ([ ]) indicate optional user input.
- Braces ({ }) indicate that you have the choice between two or more entries. You must select at least one unless the entries are optional.
- Information followed by an ellipsis (...) can be repeated any number of times (up to the length of the line).

1

General Information About MBASIC-86

This chapter includes information on writing and running MBASIC-86 programs.

Modes of Operation

When you start MBASIC-86, it displays the prompt "Ok". "Ok" means MBASIC-86 is at command level; that is, it is ready to accept commands. At this point, MBASIC-86 can be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, MBASIC-86 statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations can be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using MBASIC-86 as a "calculator" for quick computations that do not require a complete program.

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

Line Format

Program lines in a MBASIC-86 program have the following format:

nnnnnMBASIC-86 statement [:MBASIC-86 statement...] <Return>

At the programmer's option, more than one MBASIC-86 statement can be placed on a line, but each statement on a line must be separated from the last by a colon.

A MBASIC-86 program line always begins with a line number, ends by pressing the Return key, and can contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by using the line feed (LF) key. Pressing the line feed (LF) key lets you continue typing a logical line on the next physical line without pressing the Return key.

Line Numbers

Every MBASIC-86 program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65535. A period (.) can be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

Character Set

The MBASIC-86 character set consists of alphabetic characters, numeric characters and special characters.

The alphabetic characters in MBASIC-86 are the uppercase and lowercase characters of the alphabet.

The numeric characters in MBASIC-86 are the digits 0 through 9.

MBASIC-86 recognizes the following special characters and terminal keys:

Character	*Name*
	Blank
=	Equal sign or assignment symbol
+	Plus sign
–	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
‘	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
BS	Deletes last character typed
ESC	Escapes Edit Mode subcommands
Tab	Moves print position to next tab stop Tab stops are every eight columns
LF	Moves to next physical line
Return	Terminates input of a line

Control Characters

The following control characters are in MBASIC-86:

Ctrl/A	Enters Edit Mode on the line being typed.
Ctrl/C	Interrupts program execution and returns to MBASIC-86 command level.
Ctrl/G	Sounds a beep at the terminal.
Ctrl/H	Backspace. Deletes the last character typed.
Ctrl/I	Tab. Tab stops are every eight columns.
Ctrl/O	Halts program output while execution continues. A second Ctrl/O restarts output.
Ctrl/R	Retypes the line that is currently being typed.
Ctrl/S	Suspends program execution.
Ctrl/Q	Resumes program execution after a Ctrl/S.
Ctrl/U	Deletes the line that is currently being typed.

Constants

Constants are the actual values MBASIC-86 uses during execution. There are two types of constants: string and numeric.

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

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

Numeric constants are positive or negative numbers. Numeric constants in MBASIC-86 cannot contain commas. There are five types of numeric constants.

1. Integer constants	Whole numbers between −32768 and +32767. Integer constants do not have decimal points.
2. Fixed Point	Positive or negative real numbers, constants, that is, numbers that contain decimal points.
3. Floating Point	Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10−38 to 10+38. Examples: 235.988E−7 = .0000235988 2359E6 = 2359000000 Double precision floating point constants use the letter D instead of E.
4. Hex constants	Hexadecimal numbers with the prefix &H. Examples: &H76 &H32F
5. Octal constants	Octal numbers with the prefix &0 or &. Examples: &0347 &1234

Single And Double Precision Form For Numeric Constants

Numeric constants can be either single precision or double precision numbers. Single precision numeric constants are stored with seven digits of precision, and printed with up to six 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:

- Seven or fewer digits
- Exponential form using E
- A trailing exclamation point (!)

Examples:

Single Precision Constants	*Double Precision Constants*
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Variables

Variables are names used to represent values that are used in a MBASIC-86 program. The value of a variable can be assigned explicitly by the programmer, or it can be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

Variable Names And Declaration Characters

MBASIC-86 variable names can be any length; up to 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point. The first character must be a letter.

A variable name cannot be a reserved word, but embedded reserved words are allowed. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all MBASIC-86 commands, statements, function names and operator names.

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

Numeric variable names can declare integer, single or double precision values. The type declaration characters for these variable names are as follows:

% Integer variable

! Single precision variable

Double precision variable

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

Examples of MBASIC-86 variable names follow.

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.

There is a second method by which variable types may be declared. The MBASIC-86 statements DEFINT, DEFSTR, DEFSNG and DEFDBL can be included in a program to declare the types for certain variable names. Refer to Chapter 2 for a more detailed description of these statements.

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) references a value in a one-dimension array, T(1,4) references 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.

Space Requirements

The space requirements for variables, arrays, and strings are:

VARIABLES:

Integer	2 bytes
Single Precision	4 bytes
Double Precision	8 bytes

ARRAYS:

Integer	2 bytes per element
Single Precision	4 bytes per element
Double Precision	8 bytes per element

STRINGS:	3 bytes overhead plus the present contents of the string

Type Conversion

When necessary, MBASIC-86 converts a numeric constant from one type to another. Note the following rules and examples.

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

Example:

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

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

   Examples:

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

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

```
10 D = 6#/7
20 PRINT D
RUN
.8571429
```

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

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

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

   Example:

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

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

   Example:

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

Expressions and Operators

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

Operators perform mathematical or logical operations on values. The operators provided by MBASIC-86 can be divided into four categories:

- Arithmetic
- Relational
- Logical
- Functional

Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	*Operation*	*Sample Expression*
^	Exponentiation	X^Y
–	Negation	–X

*,/	Multiplication, Floating Point Division	X*Y X/Y
+,−	Addition, Subtraction	X+Y X−Y

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

Here are some sample algebraic expressions and their MBASIC-86 counterparts.

Algebraic Expression	*BASIC Expression*	
$X+2Y$	X+Y*2	
$X-\frac{Y}{Z}$	X−Y/Z	
$\frac{XY}{Z}$	X*Y/Z	
$\frac{X+Y}{Z}$	(X+Y)/Z	
(X^2Y)	(X^2) *Y	
X^{Y^Z}	X^(Y^Z)	
$X(-Y)$	X*(−Y)	Two consecutive operators must be separated by parentheses.

Integer Division And Modulus Arithmetic. Two additional operators are available in MBASIC-86: Integer division and modulus arithmetic.

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

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

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

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

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

The precedence of modulus arithmetic is just after integer division.

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

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

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (−1) or "false" (0). This result can then be used to make a decision regarding program flow. See the section discussing IF. The following is a list of MBASIC-86 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

The equal sign is also used to assign a value to a variable. See the section discussing LET.

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. Here are additional examples:

```
IF SIN(X) <0 GOTO 1000
IF I MOD J <> 4 THEN K=K+1
```

Logical Operators

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

NOT

X	NOT X
1	0
0	1

AND

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

OR

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

XOR

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

IMP

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

EQV

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

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

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

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range −32768 to +32767. If the operands are not in this range, an error results. If both operands are supplied as 0 or −1, logical operators return 0 or −1. The given operation is performed on these integers in bitwise fashion, that is, 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, the AND operator can be used to "make" all but one of the bits of a status byte at a machine I/O port. The OR operators can be used to "merge" two bytes to create a particular binary value. The following examples demonstrate how the logical operators work:

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

MBASIC-86 allows you to substitute two arithmetic operators for their corresponding logical operators.

Arithmetic Operator	*Logical Operator*
*	AND
/	OR

For example:

```
10 IF A>3 AND B<17 THEN 400
```

is the same as:

```
10 IF A>3 * B<17 THEN 400
```

```
20 IF X=4 OR Y>18 THEN PRINT X,Y
```

is the same as:

```
20 IF X=4/Y>18 THEN PRINT X,Y
```

Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. MBASIC-86 has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). Chapter 3 describes all of MBASIC-86's intrinsic functions.

MBASIC-86 also allows "user defined" functions that are written by the programmer. See the section discussing DEF FN.

String Operations

Strings can be concatenated using +. For example:

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

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

= <> < > ≤ ≥

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

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

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

Input Editing

If an incorrect character is entered as a line is being typed, it can be deleted with the backspace (BS) key or with Ctrl/H. Ctrl/H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line.

To delete a line that is in the process of being typed, type Ctrl/U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. MBASIC-86 automatically replaces the old line with the new line.

MBASIC-86 provides more sophisticated editing capabilities. See the section discussing EDIT.

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

Error Messages

If MBASIC-86 detects an error that causes program execution to terminate, an error message is printed. For a complete list of MBASIC-86 error codes and error messages, see Appendix A.

2

MBASIC-86 Commands and Statements

Format Notation

This chapter describes all of the MBASIC-86 commands and statements listed alphabetically. Each description is formatted as follows:

Format: Shows the correct format for the instruction.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

AUTO

Format: AUTO[line number[,increment]]

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

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

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input replaces the existing line. However, typing a Return immediately after the asterisk saves the line and generates the next line number.

AUTO is terminated by typing Ctrl/C. The line in which Ctrl/C is typed is not saved. After Ctrl/C is typed, MBASIC-86 returns to command level.

Example:

```
AUTO 100,50
```

Generates line numbers 100, 150, 200 . . .

```
AUTO
```

Generates line numbers 10, 20, 30, 40 . . .

CALL

Format: CALL variable name[(argument list)]

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is the recommended way of interfacing machine language programs with MBASIC-86. It is further suggested that the old style user call (x = USR(n)) not be used. Refer to the section of Chapter 4, "Assembly Language Subroutines" of the *MBASIC-86 User's Guide* which discusses using the CALL statement, for a complete description of how to use the CALL statement for assembly language subroutines.

When a CALL statement is executed, control is transferred to the user's routine via the segment address given in the last DEF SEG statement, and offset given in the variable name. Values are returned to MBASIC-86 by including the variable name which will receive the result in the argument list.

Example:

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

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

CHAIN

Format: CHAIN [MERGE] "filename"[,[line number exp] [,ALL][,DELETErange]]

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

Remarks: "Filename" is the name of the program that is called. Example:

```
CHAIN "PROG1"
```

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

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

Line number exp is not affected by a RENUM command.

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

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

If the MERGE option is included, it allows a subroutine to be brought into the MBASIC-86 program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE "OVRLAY",1000
```

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:

```
CHAIN MERGE "OVRLAY2",1000,DELETE 1000-5000
```

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

Notes:

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

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

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions are undefined after the merge is complete.

CLEAR

Format: CLEAR[, [expression1][, expression2]]

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

Remarks: Expression1 is a memory location which, if specified, sets the highest location available for use by MBASIC-86.

Expression2 sets aside stack space for MBASIC-86. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

Notes: MBASIC-86 allocates string space dynamically. An "Out of string space error" occurs only if there is no free memory left for MBASIC-86 to use.

MBASIC-86 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 COMMON and user variables
Resets the stack and string space
Releases all disk buffers

Examples:

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```

CLOSE

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

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

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

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

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

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

Example: See Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

COMMON

Format: COMMON list of variables

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements can appear anywhere in a program, though you should place them at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "( )" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

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

Note:
MBASIC-86 supports a modified version of the COMMON statement. COMMON statements can appear anywhere in a program, though you should place them before any executable statements. The current non-executable statements are:

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

Arrays in COMMON must be declared in preceding DIM statements.

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/ list of variables

where name is one to six alphanumeric characters starting with a letter. This is useful for communicating with FORTRAN and assembly language routines without having to explicitly pass parameters in the CALL statement.

The blank COMMON size and order of variables must be the same in the CHAINing and CHAINed-to programs. With MBASIC-86, the best way to ensure this is to place all blank COMMON declarations in a single include file and use the %INCLUDE statement in each program. For 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
```

CONT

Format: CONT

Purpose: To continue program execution after typing a Ctrl/C or after executing a STOP or END statement.

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

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

CONT is invalid if the program has been edited during the break.

Example: See the example in the section discussing STOP.

DATA

Format: DATA list of constants

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

Remarks: DATA statements are nonexecutable and can be placed anywhere in the program. A DATA statement can contain as many constants as fit on a line (separated by commas), and any number of DATA statements can be used in a program. 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.

List of constants can contain numeric constants in any format, that is, fixed point, floating point or integer. No numeric expressions are allowed in the list. String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type, numeric or string, given in the READ statement must agree with the corresponding constant in the DATA statement.

A DATA statement can be reread from the beginning by use of the RESTORE statement.

Example: See the examples in the section discussing READ.

DEF FN

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

Purpose: To define and name a function that you write.

Remarks: Name must be a legal variable name. This name, preceded by FN, becomes the name of the function. Parameter list is comprised of those variable names in the function definition that are 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. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

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

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

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

Example:

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

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

DEFINT/SNG/DBL/STR

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

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

Remarks: 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.

If no type declaration statements are encountered, MBASIC-86 assumes all variables without declaration characters are single precision variables.

Examples:

```
10 DEFDBL L-P
```

All variables beginning with the letters L, M, N, O, and P 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.

DEF SEG

Format: DEF SEG[= address]

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

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

Remarks: The address specified is saved for use as the segment required by the PEEK function and the POKE and CALL statements.

Any value entered outside of the address range results in an "Illegal Function Call" error. The previous value is retained.

If the address option is omitted, the segment to be used is set to MBASIC-86's Data Segment (DS). This is the initial default value.

If the address option is given, it should be a value based upon a 16-byte boundary. For the PEEK function or POKE or CALL statements, the value is shifted left 4 bits to form the Code Segment address for the subsequent call instruction. MBASIC-86 does not perform additional checking to assure that the resultant segment + offset value is valid.

NOTE
DEF and SEG must be separated by a space. Otherwise, MBASIC-86 would interpret the statement DEFSEG = 100 to mean, "assign the value 100 to the variable DEFSEG."

Example:

```
10 DEF SEG=&HB800 'Set segment to Screen buffer
20 DEF SEG 'Restore segment to MBASIC-86's DS
```

DEF USR

Format: DEF USR[digit] = integer expression

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

Remarks: Digit can be any digit from one to nine. The digit corresponds to the number of the USR routine whose address is being specified. If digit is omitted, DEF USR0 is assumed. The value of integer expression is the starting address of the USR routine. See Chapter 4, "Assembly Language Subroutines," of the *MBASIC-86 User's Guide*.

Any number of DEF USR statements can occur in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

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

DELETE

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

Purpose: To delete program lines.

Remarks: MBASIC-86 always returns to command level after a DELETE is executed. If line number does not exist, an "Illegal function call" error occurs.

Examples:

```
DELETE 40
```

Deletes line 40

```
DELETE 40-100
```

Deletes lines 40 through 100, inclusive

```
DELETE-40
```

Deletes all lines up to and including line 40.

DIM

Format: DIM list of subscripted variables

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

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be ten. 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 zero, unless otherwise specified with the OPTION BASE statement.

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Examples:

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

EDIT

Format: EDIT line number

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, you can edit portions of a line without retyping the entire line. When entering Edit Mode, MBASIC-86 types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

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

Edit Mode subcommands perform the following functions:

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

In the descriptions of the subcommands that follow, ch represents any character, text represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and ESC represents pressing the Escape key.

1. Moving the Cursor

Space — Use the space bar to move the cursor to the right. [i]Space moves the cursor "i" spaces to the right. Characters are printed as you space over them.

Backspace	In Edit Mode, [i]Backspace moves the cursor "i" spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I	I textESC inserts text at the current cursor position. The inserted characters print on the terminal. To terminate insertion, type ESC. If Return is pressed during an Insert command, the effect is the same as typing ESC and then pressing the Return key. During an Insert command, the backspace key on the terminal can be used to delete characters to the left of the cursor. Backspace prints out the characters as you backspace over them. If an attempt is made to insert a character that makes the line longer than 255 characters, a bell (Ctrl/G) is typed and the character is not printed.
X	The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, press the ESC or Return key.

3. Deleting Text

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

4. Finding Text

S	The subcommand [i]S ch searches for the ith occurrence of ch and positions the cursor before it. The character at the current cursor position is not included in the search. If ch is not found, the cursor stops at the end of the line. All characters passed over during the search are printed.
K	The subcommand [i]K ch is similar to [i]s ch, except all the characters passed over in the search are deleted. The cursor is positioned before ch, and the deleted characters are enclosed in backslashes.

5. Replacing Text

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

6. Ending and Restarting Edit Mode

Return	Pressing the Return key prints the remainder of the line, saves the changes you made, and exits Edit Mode.
E	The E subcommand has the same effect as pressing the Return key, except the remainder of the line is not printed.
Q	The Q subcommand returns to MBASIC-86 command level, *without* saving any of the changes that were made to the line during Edit Mode.

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

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

NOTE
If MBASIC-86 receives an unrecognizable command or illegal character while in Edit Mode, the Rainbow 100 computer beeps and the command or character is ignored.

Syntax Errors

When a syntax error is encountered during execution of a program, MBASIC-86 automatically enters Edit Mode at the line that caused the error. For example:

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

When you finish editing the line and press the Return key (or the E subcommand), MBASIC-86 reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first leave the Edit Mode by typing the "Q" subcommand and pressing the Return key. MBASIC-86 returns to command level, and all variable values are preserved.

Ctrl/A

To enter Edit Mode on the line you are currently typing, type Ctrl/A. MBASIC-86 inserts a carriage return, an exclamation point (!) and a space. The cursor is positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

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

END

Format: END

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

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

You can include more than one END statement in your MBASIC-86 program.

Examples:

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

ERASE

Format: ERASE list of array variables

Purpose: To eliminate arrays from a program.

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

Examples:

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

ERR and ERL Variables

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

If the statement that causes the error is a direct mode statement, ERL contains 65535. To test if an error occurs in a direct statement, use IF 65535 = ERL THEN . . .

Otherwise, use

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 can appear to the left of the equal sign in a LET (assignment) statement.

ERROR

Format: ERROR integer expression

Purpose: 1) To simulate the occurrence of a MBASIC-86 error.
2) To allow error codes to be defined by the user.

Remarks: The value of integer expression must be greater than 0 and less than 255. If the value of integer expression equals an error code already in use by MBASIC-86, the ERROR statement simulates the occurrence of that error, and the corresponding error message is printed. See Example 1.

To define your own error code, use a value that is greater than any used by the MBASIC-86 error codes. It is preferable to use the highest available values, so compatibility can be maintained when more error codes are added to MBASIC-86. This user-defined error code can then be conveniently handled in an error trap routine. See Example 2.

If an ERROR statement specifies a code for which no error message has been defined, MBASIC-86 responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

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

Or, in direct mode:

```
Ok
ERROR 15
String too long
Ok
```

Example 2:

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

FIELD

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

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

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

File number is the number under which the file is OPENed. Field width is the number of characters to be allocated to string variable. For example:

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

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

The total number of bytes allocated in a FIELD statement must not exceed the record length that is specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. The default record length is 128.

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

Example: See Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

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

FILES

Format: FILES["filename"]

Purpose: To display the names of files stored on the current diskette.

Remarks: If "filename" is omitted, all the files on the currently selected drive are listed. "filename" is a string formula which contains question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or type matches any file or any type.

Examples:

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

FOR . . . NEXT

Format: FOR variable = x TO y [STEP z]

.

.

.

NEXT [variable][,variable. . .]

where x, y and z are numeric expressions.

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

Remarks: Variable is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement execute until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, MBASIC-86 branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR. . .NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter decrements each time through the loop, and the loop executes until the counter is less than the final value.

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.

Nested Loops

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

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, MBASIC-86 displays a "NEXT without FOR" error message and execution is terminated.

Example 1:

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

Example 2:

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

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

Example 3:

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

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

GET

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

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

Remarks: The file number is the number under which the file was OPENed. If record number is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example: See Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

Note: After a GET statement, the FIELD statement can be used to adjust characters in a random file buffer.

GOSUB. . . RETURN

Format: GOSUB line number

.

.

.

RETURN

Purpose: To branch to, and return from, a subroutine.

Remarks: Line number is the first line of the subroutine.

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

The RETURN statement(s) in a subroutine cause MBASIC-86 to branch back to the statement following the most recent GOSUB statement. A subroutine can contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines can be inserted anywhere in the program, but the subroutines should be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it can be preceded by a STOP, END or GOTO statement that directs program control around the subroutine.

Example:

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

GOTO

Format: GOTO line number

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

Remarks: If line number is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after line number.

Example:

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

IF. . . THEN[. . . ELSE] and IF. . . GOTO

Format: IF expression THEN statement(s) or line number

[ELSE statement(s) or line number]

Format: IF expression GOTO line number

[ELSE statement(s) or line number]

Remarks: If the result of expression is not zero, the THEN or GOTO clause is executed. THEN can be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of expression is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

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

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

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

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

does not print "A<>C" when A<>B.

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

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

IF ABS (A–1.0)<1.0E–6 THEN . . .

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

Example 1:

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

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

Example 2:

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

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

Example 3:

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

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

INPUT

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

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

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

A comma can be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement:

```
INPUT "ENTER BIRTHDATE",B$
```

prints the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then pressing the Return key to input data does not echo a carriage return/line feed sequence.

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

The variable names in the list 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.

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

Examples:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?   5
 5 SQUARED IS 25
OK
LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
OK
RUN
WHAT IS THE RADIUS?  7.4
THE AREA OF THE CIRCLE IS 171.946
WHAT IS THE RADIUS?
```

INPUT#

Format: INPUT# file number, variable list

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

Remarks: The file number is the number used when the file was OPENed for input. Variable list contains the variable names that are assigned to the items in the file. The variable type must match the type specified by the variable name. With INPUT#, no question mark prints, as with INPUT.

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

If MBASIC-86 is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark, the string item 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. If the first character of the string is not a quotation mark, the string is an unquoted string, and terminates on a comma, Return or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example: Refer to Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

KILL

Format: KILL "filename"

Purpose: To delete a file from diskette.

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

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

Example:

```
200 KILL "DATA1"
```

Refer to Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

LET

Format: [LET] variable = expression

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

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

Example:

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

Or:

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

LINE INPUT

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

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

Remarks: The prompt string is a string literal that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to string variable. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into string variable, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return you type to end the input line does not echo a carriage return/line feed sequence on the screen.

A LINE INPUT can be escaped by typing Ctrl/C. MBASIC-86 returns to command level and displays Ok. Typing CONT resumes execution at the LINE INPUT.

Example: See the example in the section discussing LINE INPUT#.

LINE INPUT#

Format: LINE INPUT# file number, string variable

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

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

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a MBASIC-86 program saved in ASCII mode is being read as data by another program.

Example:

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

LIST

Format 1: LIST [line number]

Format 2: LIST [line number[– [line number]]]

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

Remarks: MBASIC-86 always returns to command level after a LIST is executed.

Format 1: If you omit line number, the program is listed beginning at the lowest line number. Listing is terminated either by the end of the program or by typing Ctrl/C. If line number is included, only the specified line is listed.

Format 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.

2. If only the second number is specified, all lines from the beginning of the program through that line are listed.

3. If both numbers are specified, the entire range is listed.

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 lowest number through 1000.

```
LIST 150-1000
```

Lists lines 150 through 1000, inclusive.

LLIST

Format: LLIST [line number[– [line number]]]

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

Remarks: LLIST assumes a 132-character-wide printer.

MBASIC-86 always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Format 2.

Example: See the examples for LIST, Format 2.

LOAD

Format: LOAD "filename"

Purpose: To load a file from the diskette into memory.

Remarks: The "filename" is the name that was used when the file was SAVEd.

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

Example:

```
LOAD "STRTRK",R
```

LPRINT and LPRINT USING

Format: LPRINT [list of expressions]

LPRINT USING string exp;list of expressions

Purpose: To print data on the printer.

Remarks: Same as PRINT and PRINT USING, except output is printed on the printer.

LPRINT assumes a 132-character-wide printer.

LSET and RSET

Format: LSET string variable = string expression
RSET string variable = string expression

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

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

Examples:

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

See also Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

Note: 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. For example:

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

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

MERGE

Format: MERGE "filename"

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

Remarks: "Filename" is the name used when the file was SAVEd. The file must be SAVEd in ASCII format. If not, a "Bad file mode" error occurs.

If any lines in the diskette file have the same line numbers as lines in the program in memory, the lines from the file on diskette replace the corresponding lines in memory. MERGEing can be thought of as "inserting" the program lines on diskette into the program in memory.

MBASIC-86 always returns to command level after executing a MERGE command.

Example:

```
MERGE "NUMBRS"
```

MID$

Format: MID$ (string expl,n[,m]) = string exp2

where n and m are integer expressions and string expl and string exp2 are string expressions.

Purpose: To replace a portion of one string with another string.

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

Example:

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

MID$ is also a function that returns a substring of a given string.

NAME

Format: NAME "old filename" AS "new filename"

Purpose: To change the name of a diskette file.

Remarks: The "old filename" must exist and the "new filename" must not exist; otherwise an error results. After a NAME command, the file exists on the same diskette, in the same area of disk space, with the new name.

Example:

```
OK
NAME "ACCTS" AS "LEDGER"
OK
```

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

NEW

Format: NEW

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

Remarks: NEW is entered at command level to clear memory before entering a new program. MBASIC-86 always returns to command level after a NEW is executed.

NULL

Format: NULL integer expression

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

Remarks: For 10-character-per-second tape punches, integer expression should be 13. When tapes are not being punched, integer expression should be 0 or 1 for Teletypes and Teletype-compatible terminal screens. Integer expression should be 2 or 3 for 30 cps hard copy printers. The default value is 0.

Example:

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

Two null characters are printed after each line.

ON ERROR GOTO

Format: ON ERROR GOTO line number

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

Remarks: Once error trapping has been enabled, all errors detected, including direct mode errors (for example, Syntax errors), cause a jump to the specified error handling subroutine. If line number does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes MBASIC-86 to stop and print the error message for the error that caused the trap. All error trapping subroutines should execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note: If an error occurs during execution of an error handling subroutine, the MBASIC-86 error message prints and execution terminates. Error trapping does not occur within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 1000
```

ON. . . GOSUB and ON. . . GOTO

Format: ON expression GOTO list of line numbers

ON expression GOSUB list of line numbers

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

Remarks: The value of expression determines which line number in the list is 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 non-integer, the fractional portion is rounded.

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

If the value of expression is zero or greater than the number of items in the list, but less than or equal to 255, MBASIC-86 continues with the next executable statement. If the value of expression is negative or greater than 255, an "Illegal function call" error occurs.

Example:

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

OPEN

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

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

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

Mode is a string expression whose first character is one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

File number is an integer expression whose value is between 1 and 15. The number is then associated with the file for as long as it is OPEN and is used to refer other diskette I/O statements to the file.

"Filename" is a string expression containing a name that conforms to the CP/M-86/80 operating system rules for diskette filenames, that is, "filename.ext".

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

Do not include another statement on the same line with the OPEN statement.

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

Example:

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

See also Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

OPTION BASE

Format: OPTION BASE n

where n is 1 or 0.

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript can have is one.

OUT

Format: OUT I,J

where I and J are integer expressions in the range 0 to 255. The integer expression I is the port number, and the integer expression J is the data to be transmitted.

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

Remarks: OUT is the complementary statement to the INP function.

Example:

```
100 OUT 32,100
```

POKE

Format: POKE I,J

where I and J are integer expressions.

Purpose: To write a byte into a memory location.

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

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read.

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

Example:

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

PRINT

Format: PRINT [list of expressions]

Purpose: To output data on the screen.

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

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. MBASIC-86 divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return prints at the end of the line. If the printed line is longer than the terminal width, MBASIC-86 goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E – 7 is output as .0000001 and 1E – 8 is output as 1E – 08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D – 16 is output as .0000000000000001 and 1D – 17 is output as 1D – 16.

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

Example 1:

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

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

Example 2:

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

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

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

Example 3:

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

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. Don't forget, a number is always followed by a space and positive numbers are preceded by a space.

PRINT USING

Format: PRINT USING string exp;list of expressions

Purpose: To print strings or numbers using a specified format.

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

String Fields

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

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

"\n spaces\" Specifies that 2 + n characters from the string are to be printed. If the 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 field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right. An example follows:

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

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

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

Numeric Fields

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

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

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

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

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

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

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

+ A plus sign at the beginning or end of the format string 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 format field causes negative numbers to be printed with a trailing minus sign. For example:

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

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

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

```
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. For example:

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

**$ The **$ at the beginning of a format string 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. For example:

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

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

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

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

^^^^ Four carets (or up-arrows) can be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position can 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. For example:

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

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

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

_ An underscore in the format string causes the next character to be output as a literal character. For example:

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

The literal character itself can be an underscore by placing "__" in the format string.

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

```
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.

PRINT# and PRINT# USING

Format: PRINT#filenumber, [USINGstring exp;]list of exps

Purpose: To write data to a sequential diskette file.

Remarks: File number is the number used when the file was OPENed for output. String exp is comprised of formatting characters as described in the section discussing PRINT USING. The expressions in list of expressions are the numeric and/or string expressions that are written to the file.

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

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

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

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

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

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

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

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

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

The image written to the diskette is

```
CAMERA,93604-1
```

which can be read back into two string variables.

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

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

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

writes the following image to the diskette:

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

and the statement:

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

inputs "CAMERA" to A$ and "AUTOMATIC 93604 – 1" to B$. To separate these strings properly on the diskette, write double quotes to the diskette image using CHR$(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"
```

and the statement:

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

inputs "CAMERA, AUTOMATIC" to A$ and " 93604 – 1" to B$.

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

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

For more examples using PRINT#, see Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

See also the section discussing WRITE#.

PUT

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

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

Remarks: File number is the number under which the file was OPENed. If record number is omitted, the record has the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

Example: See Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide*.

Note: The FIELD statement can be used to ready characters for input into the random file buffer before a PUT statement.

RANDOMIZE

Format: RANDOMIZE [expression]

Purpose: To reseed the random number generator.

Remarks: If expression is omitted, MBASIC-86 suspends program execution and asks for a value by printing

Random Number Seed (– 32768 to 32767)?

before executing RANDOMIZE.

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

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN

Random Number Seed (-32768 to 32767)?3
.2226007 .5941419 .2414202 .2013798 5.361748E-02
Ok
RUN
Random Number Seed (-32768 to 32767)?4
.628988 .7656049 .5551561 .775797 .7834911
Ok
RUN
Random Number Seed (-32768 to 32767)?3
.2226007 .5941419 .2414202 .2013798 5.361748E-02
Ok
```

Note the first and third sequences are the same.

READ

Format: READ list of variables

Purpose: To read values from a DATA statement and assign them to variables. See the section discussing DATA.

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

A single READ statement can access one or more DATA statements (they are accessed in order), or several READ statements can access the same DATA statement. If the number of variables in list of variables exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA 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.

To reread DATA statements from the start, use the RESTORE statement. See the section discussing RESTORE.

Example 1:

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

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

Example 2:

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

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

REM

Format: REM remark

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

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

REM statements can be branched into from a GOTO or GOSUB statement, and execution continues with the first executable statement after the REM statement.

Remarks can be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

Warning: Do not use this as a data statement as it would be considered legal data.

Example:

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

Or:

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

140 NEXT I
    .
    .
    .
```

RENUM

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

Purpose: To renumber program lines.

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

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

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

Examples:

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

RESET

Format: RESET

Purpose: To close all diskette files and write the directory information to a diskette before it is removed from a diskette drive.

Remarks: Always execute a RESET command before removing a diskette from a diskette drive. Otherwise, when the diskette is used again, there is a possibility it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every diskette with open files.

RESTORE

Format: RESTORE [line number]

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

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

Example:

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

RESUME

Formats: RESUME

RESUME 0

RESUME NEXT

RESUME line number

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

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

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

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

Example:

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

RUN

Format 1: RUN [line number]

Purpose: To execute the program currently in memory.

Remarks: If line number is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. MBASIC-86 always returns to command level after a RUN is executed.

Example:

```
RUN
```

Format 2: RUN "filename" [,R]

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

Remarks: The "filename" is the name used when the file was SAVEd.

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

Example:

```
RUN "NEWFIL",R
```

See also Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide*.

Note: MBASIC-86 supports the RUN and RUN line number forms of the RUN statement. MBASIC-86 does not support the R option with RUN. If you want this feature, use the CHAIN statement.

SAVE

Format: SAVE "filename" [,A][,P]

Purpose: To save a program file on diskette.

Remarks: The "filename" is a quoted string that conforms to the CP/M-86/80 operating system's requirements for filenames. The CP/M-86/80 operating system appends the default filename type BAS if you did not supply one with the SAVE command. If "filename" already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, MBASIC-86 saves the file in a compressed binary format. ASCII format takes more space on the diskette, but some diskette access commands require that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some commands such as LIST can require an ASCII format file.

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

You should always save MBASIC-86 programs using uppercase filenames. Certain CP/M-86/80 operating system commands do not read lowercase filenames. If you have saved a program file with a lowercase file name, change it to uppercase by LOADing the lowercase filename and then SAVEing the file with an uppercase filename.

Examples:

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

See also Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide.*

STOP

Format: STOP

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

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

```
Break in line nnnnn
```

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

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

Example:

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
OK
PRINT L
30.76923
OK
CONT
115.9
OK
```

SWAP

Format: SWAP variable, variable

Purpose: To exchange the values of two variables.

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

Example:

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

SYSTEM

Format: SYSTEM

Purpose: To leave MBASIC-86 and return control to the CP/M-86/80 operating system.

Remarks: SYSTEM closes all files and then returns to the CP/M-86/80 operating system.

TRON/TROFF

Format: TRON

TROFF

Purpose: To trace the execution of program statements.

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

Example:

```
TRON
OK
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 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
```

WAIT

Format: WAIT port number, I[,J]

where I and J are integer expressions.

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

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, MBASIC-86 loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

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

Example:

```
100 WAIT 32, 2
```

WIDTH

Format: WIDTH [LPRINT] integer expression

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

Remarks: If the LPRINT option is omitted, the line width is set at the screen. If LPRINT is included, the line width is set at the printer.

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

If integer expression is 255, the line width is "infinite," that is, MBASIC-86 never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

Example:

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

WHILE. . . WEND

Format: WHILE expression

.

.

[loop statements]

.

.

WEND

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

Remarks: If expression is not zero (that is, true), loop statements are executed until the WEND statement is encountered. MBASIC-86 then returns to the WHILE statement and checks expression. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

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.

Example:

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

WRITE

Format: WRITE[list of expressions]

Purpose: To output data at the screen.

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

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, MBASIC-86 inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement.

Example:

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

WRITE#

Format: WRITE# file number, list of expressions

Purpose: To write data to a sequential file.

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

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

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

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

writes the following image to disk:

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

A subsequent INPUT# statement, such as:

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

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

3
MBASIC-86 Functions

This chapter discusses the instrinsic functions provided by MBASIC-86. The functions can be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments are abbreviated as follows:

X and Y	Represent any numeric expressions
I and J	Represent integer expressions
X$ and Y$	Represent string expressions

If a floating point value is supplied where an integer is required, MBASIC-86 rounds the fractional portion and uses the resulting integer.

> **NOTE**
> With MBASIC-86, only integer and single precision results are returned by functions.

ABS

Format: ABS(X)

Purpose: To return the absolute value of the expression X.

Example:

```
PRINT ABS(7*(-5))
 35
Ok
```

ASC

Format: ASC(X$)

Purpose: To return a numerical value that is the ASCII code of the first character of the string X$. See Appendix C for ASCII codes. If X$ is null, an "Illegal function call" error is returned.

Example:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function for ASCII-to-string conversion.

ATN

Format: ATN(X)

Purpose: To return the arctangent of X in radians. Result is in the range −pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.249046
Ok
```

CDBL

Format: CDBL(X)

Purpose: To convert X to a double precision number.

Example:

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67 454.6699829101563
Ok
```

CHR$

Format: CHR$(I)

Purpose: To return a string whose one element has ASCII code I. ASCII codes are listed in Appendix C. CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message.

Example:

```
PRINT CHR$(66)
B
Ok
```

See the ASC function for the ASCII-to-numeric conversion.

CINT

Format: CINT(X)

Purpose: To convert X to an integer by rounding the fractional portion. If X is not in the range −32768 to 32767, an "Overflow" error occurs.

Example:

```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data types. See also the FIX and INT functions, both of which return integers.

COS

Format: COS(X)

Purpose: To return the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.842122
Ok
```

CSNG

Format: CSNG(X)

Purpose: To convert X to a single precision number.

Example:

```
10 A# = 975.3421
20 PRINT A#; CSNG(A#)
RUN
 975.342041015625 975.3421
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

CVI, CVS, CVD

Format: CVI(2-byte string)
CVS(4-byte string)
CVD(8-byte string)

Purpose: To convert string values to numeric values. Numeric values that are read in from a random diskette 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.

Example:

```
 .
 .
 .
70 FIELD #1,4 AS N$, 12 AS B$, . . .
80 GET #1
90 Y=CVS(N$)
 .
 .
 .
```

See also MKI$, MKS$, MKD$, and Chapter 3, "MBASIC-86 Diskette I/O," of the *MBASIC-86 User's Guide*.

EOF

Format: EOF(file number)

Purpose: To return −1 (true) if the end of a sequential or random file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

Example:

```
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
  .
  .
  .
```

EXP

Format: EXP(X)

Purpose: To return e to the power of X. X must be ≤ 97.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.59815
Ok
```

FIX

Format: FIX(X)

Purpose: To return the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples:

```
PRINT FIX(58.75)
 58
Ok

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

FRE

Format: FRE(0)
FRE(X$)

Purpose: To return the number of bytes in memory not being used by MBASIC-86. Arguments to FRE are dummy arguments.

FRE forces a garbage collection before returning the number of free bytes. BE PATIENT: garbage collection can take 1 to 1 ½ minutes. MBASIC-86 does not initiate garbage collection until all free memory has been used up. Therefore, using FRE periodically results in shorter delays for each garbage collection.

Example:

```
PRINT FRE(0)
 14542
Ok
```

HEX$

Format: HEX$(X)

Purpose: To return a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
OK
```

See the OCT$ function for octal conversion.

INKEY$

Format: INKEY$

Purpose: To return either a one-character string containing a character read from the computer or a null string if no character is pending at the computer. No characters are echoed and all characters are passed through to the program except for Ctrl/C, which terminates the program.

Example:

```
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 RESPONSES$=RESPONSES$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

INP

Format: INP(I)

Purpose: To return the byte read from port I. I must be in the range 0 to 65535. INP is the complementary function to the OUT statement.

Example:

```
100 A=INP(255)
```

INPUT$

Format: INPUT$(X[,[#]Y])

Purpose: To return a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters are echoed and all control characters are passed through except Ctrl/C, which is used to interrupt the execution of the INPUT$ function.

Example 1:

```
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
```

Example 2:

```
    .
    .
    .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$+"P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
    .
    .
    .
```

INSTR

Format: INSTR([I,]X$, Y$)

Purpose: To search for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I>LEN(X$) or if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ can be string variables, string expressions or string literals.

Example:

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2 6
Ok
```

Note: If I = 0 is specified, the error message "Illegal function call in line number" is displayed.

INT

Format: INT(X)

Purpose: To return the largest integer ≤ X.

Examples:

```
PRINT INT(99.89)
 99
Ok

Print INT(-12.11)
-13
Ok
```

See the FIX and CINT functions which also return integer values.

LEFT$

Format: LEFT$(X$,I)

Purpose: To return a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) is returned. If I = 0, the null string (length zero) is returned.

Example:

```
10 A$ = "BASIC"
20 B$ = LEFT$(A$,5)
30 PRINT B$
RUN
BASIC
OK
```

Also see the MID$ and RIGHT$ functions.

LEN

Format: LEN(X$)

Purpose: To return the number of characters in X$. Non-printing characters and blanks are counted.

Example:

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
RUN
 16
OK
```

LOC

Format: LOC(file number)

Purpose: With random diskette files, LOC returns the record number just read or written from a GET or PUT. If the file is opened but no diskette I/O has been performed yet, LOC returns a 0. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

Example:

```
200 IF LOC(1)>50 THEN STOP
```

LOF

Format: LOF(file number)

Purpose: To return the number of records present in the last extent read or written. If the file does not exceed one extent (128 records), then LOF returns the true length of the file.

Example:

```
110 IF NUM%>LOF(1) THENPRINT "INVALID ENTRY"
```

LOG

Format: LOG(X)

Purpose: To return the natural logarithm of X. X must be greater than zero.

Example:

```
PRINT LOG(45/7)
 1.860752
Ok
```

LPOS

Format: LPOS(X)

Purpose: To return the current position of the printer print head within the line printer buffer, but does not necessarily give the physical position of the print head. X is a dummy argument.

Example:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

MID$

Format: MID$(X$,I[,J])

Purpose: To return a string on length J characters from X$ beginning with the Ith character. I and J must be in the range 1 to 255. 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. If I>LEN(X$), MID$ returns a null string.

Example:

```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
RUN
GOOD EVENING
OK
```

Also see the LEFT$ and RIGHT$ functions.

Note: If I = 0 is specified, error message "Illegal function call in line number" is displayed.

MKI$, MKS$, MKD$

Format: MKI$(integer expression)
MKS$(single precision expression)
MKD$(double precision expression)

Purpose: To convert numeric values to string values. 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 an single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example:

```
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
    .
    .
```

See also the sections discussing CVI, CVS, CVD, Chapter 3, “MBASIC-86 Diskette I/O,” of the *MBASIC-86 User's Guide*.

OCT$

Format: OCT$(X)

Purpose: To return a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example:

```
PRINT OCT$(24)
30
OK
```

See the HEX$ function for hexadecimal conversion.

PEEK

Format: PEEK(I)

Purpose: To return the byte (decimal integer in the range 0 to 225) read from memory location I. I must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement.

Example:

```
A=PEEK(&H5A00)
```

POS

Format: POS(X)

Purpose: To return the current cursor position. The leftmost position is 1. X is a dummy argument.

Example:

```
IF POS(X)>60 THEN PRINT CHR$(13)
```

Also see the LPOS function.

RIGHT$

Format: RIGHT$(X$,I)

Purpose: To return the rightmost I characters of string X$. If I = LEN(X$), returns X$. If I = 0, the null string (length zero) is returned.

Example:

```
10 A$ = "DISK BASIC"
20 PRINT RIGHT$(A$,5)
RUN
BASIC
OK
```

Also see the MID$ and LEFT$ functions.

RND

Format: RND[(X)]

Purpose: To return a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded. See the section discussing RANDOMIZE. However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X = 0 repeats the last number generated.

Example:

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 12  65  86  72  79
Ok
```

SGN

Format: SGN(X)

Purpose: To return the sign of X.
If X>0, SGN(X) returns 1.
If X = 0, SGN(X) returns 0.
If X<0, SGN(X) returns −1.

Example:

```
ON SGN(X) GOTO 100,200,300
```

branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

SIN

Format: SIN(X)

Purpose: To return the sine of X in radians. SIN(X) is calculated in single precision. COS(X) = SIN(X + 3.14159/2).

Example:

```
PRINT SIN(1.5)
 .9974951
OK
```

SPACE$

Format: SPACE$(X)

Purpose: To return a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Example:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
 1
  2
   3
    4
     5
OK
```

Also see the SPC function.

SPC

Format: SPC(I)

Purpose: To print I blanks on the screen. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A°;' is assumed to follow the SPC(I) command.

Example:

```
PRINT "OVER" SPC(15) "THERE"
OVER                THERE
OK
```

Also see the SPACE$ function.

SQR

Format: SQR(X)

Purpose: To return the square root of X. X must be ≥ 0.

Example:

```
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
```

STR$

Format: STR$(X)

Purpose: To return a string representation of the value of X.

Example:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
    .
    .
    .
```

Also see the VAL function.

STRING$

Format: STRING$(I,J)
STRING$(I,X$)

Purpose: To return a string of length I whose characters all have ASCII code J or the first character of X$.

Example:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

TAB

Format: TAB(I)

Purpose: To space to position I on the output line. 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. I must be in the range 1 to 255. TAB can only be used in PRINT and LPRINT statements.

Comment: Execute a TAB(0) before using the TAB(XX) command, especially if XX is a variable.

Example:

```
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
```

TAN

Format: TAN(X)

Purpose: To return the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 Y = Q*TAN(X)/2
```

USR

Format: USR[digit](X)

Purpose: To call the user's assembly language subroutine with the argument X. Digit is in the range zero to nine and corresponds to the digit supplied with the DEF USR statement for that routine. If digit is omitted, USR0 is assumed. See Chapter 4, "Assembly Language Subroutines," of the *MBASIC-86 User's Guide*.

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
    .
    .
    .
```

VAL

Format: VAL(X$)

Purpose: To return the numerical value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For following example returns −3.

VAL(" −3")

Example:

```
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 VAL(ZIP$)>=90801 AND VAL(ZIP$)=90815 THEN
PRINT NAME$ TAB(25) LONG BEACH''
    .
    .
    .
```

See the STR$ function for numeric to string conversion.

VARPTR

Format 1: VARPTR(variable name)

Format 2: VARPTR(#file number)

Purpose: 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 can 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.

Note: 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.

Format 2: For sequential files, returns the starting address of the diskette I/O buffer assigned to file number. For random files, returns the address of the FIELD buffer assigned to file number.

Example:

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

Appendices

A

Summary of Error Codes and Error Messages

Code	Number	Message
NF	1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	2	Syntax error A line is encountered that contains some incorrect sequence of characters such as unmatched parenthesis, misspelled command or statement, or incorrect punctuation.
RG	3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.

Code	Number	Message
FC	4	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error can also occur as the result of: 1. A negative or unreasonably large subscript 2. A negative or zero argument with LOG 3. A negative argument to SQR 4. A negative mantissa with a non-integer exponent 5. A call to a USR function for which the starting address has not yet been given 6. An improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.
OV	6	Overflow The result of a calculation is too large to be represented in MBASIC-86's number format. If underflow occurs, the result is zero and execution continues without an error.
OM	7	Out of memory A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
UL	8	Undefined line number A line reference in a GOTO, GOSUB, IF. . . THEN. . . ELSE or DELETE is to a nonexistent line.

Code	Number	Message
BS	9	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.
DD	10	Duplicate definition 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.
/0	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.
ID	12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
TM	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.
OS	14	Out of string space String variables have caused MBASIC-86 to exceed the amount of free memory remaining. MBASIC-86 allocates string space dynamically, until it runs out of memory.
LS	15	String too long An attempt is made to create a string more than 255 characters long.

Code	Number	Message
ST	16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.
CN	17	Can't continue An attempt is made to continue a program that: 1. has halted due to an error, 2. has been modified during a break in execution, or 3. does not exist.
UF	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.

Code	Number	Message
	26	FOR without NEXT A FOR is encountered without a matching NEXT.
	29	WHILE without WEND A WHILE statement does not have a matching WEND.
	30	WEND without WHILE A WEND is encountered without a matching WHILE.

Diskette Errors

Code	Number	Message
	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 in MBASIC-86. Report to Digital Equipment Corporation the conditions under which the message appeared. Call the service number listed in the *MBASIC-86 User's Guide.*
	52	Bad file number A statement of command references a file with a file number that is not OPEN 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 diskette.
	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 with a file mode other than I, O, or R.

Code	Number	Message
	55	File already open A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.
	57	Disk I/O error An I/O error occurred on a diskette I/O operation. It is a fatal error, so the operating system cannot recover from the error.
	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 diskette storage space is in use. Delete unused files.
	62	INPUT past end An INPUT statement is executed after all the data in the file has been INPUT, or for a null (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 (for example, a filename with too many characters).
	66	Direct statement in file A direct statement is encountered while LOADing and 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 used. Delete unused files.

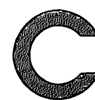
B

Mathematical Functions

Functions that are not intrinsic to MBASIC-86 may be calculated as follows.

Function	MBASIC-86 Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN (X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)-ATN(X/SQR(X*X-1)) +SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=EXP(-X)/(EXP(X)+EXP(-X))*2+1

Function	MBASIC-86 Equivalent
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC CONSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2



ASCII Character Codes

Dec	Hex	CHR	DEC	Hex	CHR
000	00H	NUL	018	12H	DC2
001	01H	SOH	019	13H	DC3
002	02H	STX	020	14H	DC4
003	03H	ETX	021	15H	NAK
004	04H	EOT	022	16H	SYN
005	05H	ENQ	023	17H	ETB
006	06H	ACK	024	18H	CAN
007	O7H	BEL	025	19H	EM
008	08H	BS	026	1AH	SUB
009	09H	HT	027	1BH	ESCAPE
010	0AH	LF	028	1CH	FS
011	0BH	VT	029	1DH	GS
012	0CH	FF	030	1EH	RS
013	0DH	CR	031	1FH	US
014	0EH	SO	032	20H	SPACE
015	0FH	SI	033	21H	!
016	10H	DLE	034	22H	“
017	11H	DC1	035	23H	#

Dec	Hex	CHR	DEC	Hex	CHR
036	24H	$	071	47H	G
037	25H	%	072	48H	H
038	26H	&	073	49H	I
039	27H	’	074	4AH	J
040	28H	(	075	4BH	K
041	29H	)	076	4CH	L
042	2AH	*	077	4DH	M
043	2BH	+	078	4EH	N
044	2CH	,	079	4FH	O
045	2DH	-	080	50H	P
046	2EH	.	081	51H	Q
047	2FH	/	082	52H	R
048	30H	0	083	53H	S
049	31H	1	084	54H	T
050	32H	2	085	55H	U
051	33H	3	086	56H	V
052	34H	4	087	57H	W
053	35H	5	088	58H	X
054	36H	6	089	59H	Y
055	37H	7	090	5AH	Z
056	38H	8	091	5BH	[
057	39H	9	092	5CH	\
058	3AH	:	093	5DH	]
059	3BH	;	094	5EH	^
060	3CH	<	095	5FH	_
061	3DH	=	096	60H	‘
062	3EH	>	097	61H	a
063	3FH	?	098	62H	b
064	40H	@	099	63H	c
065	41H	A	100	64H	d
066	42H	B	101	65H	e
067	43H	C	102	66H	f
068	44H	D	103	67H	g
069	45H	E	104	68H	h
070	46H	F	105	69H	i

Dec	Hex	CHR	DEC	Hex	CHR
106	6AH	j	117	75H	u
107	6BH	k	118	76H	v
108	6CH	l	119	77H	w
109	6DH	m	120	78H	x
110	6EH	n	121	79H	y
111	6FH	o	122	7AH	z
112	70H	p	123	7BH	{
113	71H	q	124	7CH	¦
114	72H	r	125	7DH	}
115	73H	s	126	7EH	~
116	74H	t	127	7FH	DEL

Dec = decimal, Hex = hexadecimal (H), CHR = character.
LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Rubout

Index

A

ABS 118
ALL 24, 28
Arithmetic operators 10
Array variables 8
ASC 118, 120
ASCII character codes 153
ASCII format 106
Assembly language subroutine 23, 29, 36, 141
ATN 119
AUTO 2, 22

B

Boolean operations 14

C

CALL 23, 25
CDBL 119, 120, 121
CHAIN 24, 29, 105
Character set 2
CHR$ 118, 120
CINT 120, 121, 128
CLEAR 26
CLOSE 27
Common 24, 26, 28
Concatenation of strings 17
Constants 4
CONT 30, 64, 107
Control characters 4
COS 121
CSNG 120, 121
CVD 122, 133
CVI 122, 133
CVS 122, 133

D

DATA 31, 97, 103
DEFDBL 7, 25, 28, 34
DEF FN 32
DEFINT 7, 25, 28, 34

DEF SEG 23, 35
DEFSNG 7, 25, 28, 34
DEFSTR 7, 25, 28, 34
DEF USR 36, 141
DELETE 2, 37
Deleting
  characters 18
  lines 18
  programs 18
DIM 28, 38
Direct mode 1
Division by zero 12
Double precision constants 6
Double precision variable 7

E

EDIT 2, 18, 39
Edit mode 39
END 27, 30, 44, 55, 107
EOF 123
ERASE 45
ERL 46, 100
ERR 46
ERROR 47
Error messages 19, 145
Error trapping 77
Escape key 39
EXP 123

F

Field 49, 54, 95, 142
FILES 50
FIX 120, 124, 128
Fixed point constants 5
Floating point constants 5
FOR . . . NEXT 51
Format notation 21
FORTRAN subroutines 29
FRE 124
Functional operators 17
Functions 117

G

GET 49, 54, 130
GOSUB . . . RETURN 55, 99, 100
GOTO 30, 55, 56, 99, 100

H

Hex constants 5
HEX$ 125, 134

I

IF . . . GOTO 57
IF . . . THEN 46, 57
IF . . . THEN . . . ELSE 57
Indirect mode 1
INKEY$ 126
INP 82, 126
INPUT 59
INPUT$ 127
INPUT# 61
Input editing 18
INSTR 128
INT 120, 124, 128
Integer constants 5
Integer division 12
Integer variable 7

K

KILL 62

L

LEFT$ 129, 135
LEN 129
LET 46, 63
Line feed key 2
Line format 2
LINE INPUT 64
LINE INPUT# 64, 65
Line Numbers 2

LIST 2, 66, 106
LLIST 68
LOAD 69, 106
LOC 130
LOF 130
LOG 131
Logical operators 14
LPOS 112, 131, 135
LPRINT 70, 112, 140
LPRINT USING 70
LSET 49, 71

M

Mathematical functions 151
MERGE 24, 72
MID$ 73, 132, 135
MKD$ 71, 122, 133
MKI$ 71, 122, 133
MKS$ 71, 122, 133
Modes of Operation
  direct mode 1
  indirect mode 1
Modulus arithmetic 12

N

NAME 74
NEW 18, 27, 75, 110
NULL 76
Numeric constants 4, 6

O

OCT$ 125, 134
Octal constants 5
Operators
  arithmetic 10
  functional 17
  logical 14
  relational 12
ON ERROR GOTO 77
ON . . . GOSUB 78, 100
ON . . . GOTO 78, 100
OPEN 49, 54, 62, 65, 79, 95, 105, 115, 130
OPTION BASE 28, 38, 81
OUT 82, 126

P

PEEK 35, 83, 134
POKE 35, 83, 134
POS 112, 135
PRINT 84, 114, 140
PRINT USING 87
PRINT# 92, 115
PRINT# USING 92
PUT 49, 71, 95, 130

R

Random file 49, 54, 71, 130, 142
Random numbers 96, 136
RANDOMIZE 96, 136
READ 31, 97, 103
Relational Operators 12
REM 28, 99
RENUM 24, 100
RESET 49, 71, 102
RESTORE 31, 97, 102
RESUME 104
RIGHT$ 135
RND 96, 136
RUN 105

S

SAVE 69, 72, 106
Sequentail file 61, 65, 92, 115, 142
SGN 136
SIN 137
Single precision constants 6
Single precision variable 7

SPACE$ 137, 138
Space requirements 8
Special characters 3
SPC 137, 138
SQR 138
STEP 51
STOP 27, 30, 44, 55, 107
STR$ 139, 141
String constants 4
String operations 17
String variables 7
STRING$ 139
SWAP 108
SYSTEM 109

T

TAB 140
TAN 140
TRON 110
TROFF 110
Type conversion 8

U

USR 141

V

VAL 139, 141
Variable naming conventions 6
VARPTR 142

W

WAIT 111
WIDTH 112
WHILE . . . WEND 113
WRITE 114
WRITE# 94, 115

HOW TO ORDER ADDITIONAL DOCUMENTATION

If you want to order additional documentation by phone:

And you live in:	Call:	Between the hours of:
New Hampshire, Alaska or Hawaii	603-884-6660	8:30 AM and 6:00 PM Eastern Time
Continental USA or Puerto Rico	1-800-258-1710	8:30 AM and 6:00 PM Eastern Time
Canada (Ottawa-Hull)	613-234-7726	8:00 AM and 5:00 PM Eastern Time
Canada (British Columbia)	1-800-267-6146	8:00 AM and 5:00 PM Eastern Time
Canada (all other)	112-800-267-6146	8:00 AM and 5:00 PM Eastern Time

If you want to order additional documentation by direct mail:

And you live in:	Write to:
USA or Puerto Rico	DIGITAL EQUIPMENT CORPORATION Attn: Accessories and Supplies Centers P.O. Box CS2008 Nashua, NH 03061 NOTE: Prepaid orders from Puerto Rico must be placed with the local DIGITAL subsidiary (Phone 809-754-7575)
Canada	DIGITAL EQUIPMENT OF CANADA LTD. 940 Belfast Road Ottawa, Ontario K1G 4C2 Attn: A&SG Business Manager
Other than USA, Puerto Rico or Canada	DIGITAL EQUIPMENT CORPORATION Accessories and Supplies Center A&SG Business Manager c/o Digital's local subsidiary or approved distributor

Digital Equipment Corporation
Personal Computer Software Products
Software License Transfer Agreement

I understand and agree to abide by the Software License Agreement that accompanies the software product.

I have obtained the Software Product from the following Transferor:

____
(Name of Transferor)

____
(Address)

The Software Product(s) I am receiving is: (List Product(s) and include Serial Number, if any):

______________________ ______________________

______________________ ______________________

______________________ ______________________

______________________ ______________________

The Software Product(s) will be used on CPU Serial Number: ______________

(Transferee: name of person/organization receiving the Software Product from Transferor.)

By: ____
(Signature of Transferee)

____
(Print Name)

Company: ____

Address: ____

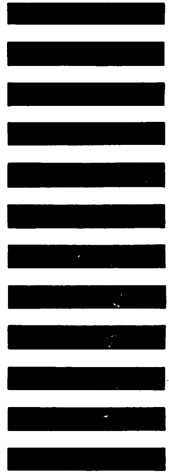
____

Date: ____

Do Not Tear – Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CONTINENTAL BOULEVARD
MERRIMACK NH 03054
ATTN: PERSONAL COMPUTER SOFTWARE

Do Not Tear – Fold Here and Tape

Cut Along Dotted Line

Rainbow™ 100
MBASIC™–86 Reference Manual
AA–P602B–TV

READER'S COMMENTS

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ First-time computer user
- ☐ Experienced computer user
- ☐ Application package user
- ☐ Programmer
- ☐ Other (please specify)__________

Name__________

Date__________

Organization__________

Street__________

City__________

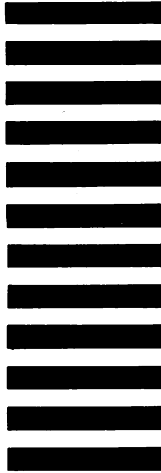
State__________

Zip Code
or Country__________

Do Not Tear – Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MRO1-2/L12
MARLBOROUGH, MA 01752

Do Not Tear – Fold Here and Tape

Cut Along Dotted Line