MS-BASIC

COPYRIGHT

© 1983 by VICTOR.®
© 1979 by Microsoft Corporation.

Published by arrangement with Microsoft Corporation, whose software has been customized for use on various desktop microcomputers produced by VICTOR. Portions of the text hereof have been modified accordingly.

All rights reserved. This publication contains proprietary information which is protected by copyright. No part of this publication may be reproduced, transcribed, stored in a retrieval system, translated into any language or computer language, or transmitted in any form whatsoever without the prior written consent of the publisher. For information contact:

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

TRADEMARKS

VICTOR is a registered trademark of Victor Technologies, Inc.
MS-DOS is a registered trademark of Microsoft Corporation.
CP/M-86 is a registered trademark of Digital Research, Inc.

NOTICE

VICTOR makes no representations or warranties of any kind whatsoever with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. VICTOR shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this publication or its contents.

VICTOR reserves the right to revise this publication from time to time and to make changes in the content hereof without obligation to notify any person of such revision or changes.

First VICTOR printing April, 1983.

ISBN 0-88182-066-0

Printed in U.S.A.

CONTENTS

I. General Information About MS-BASIC

1.1 Introduction 1
1.2 Modes of Operation 1
1.3 Line Format and Line Numbers 1
1.4 Character Set 2
 1.4.1 Special Characters and Terminal Keys 2
 1.4.2 Alternate Characters 3
1.5 Constants 3
 1.5.1 Single and Double Precision Form for Numeric Constants 4
1.6 Variables 5
 1.6.1 Variable Names and Declaration Characters 5
 1.6.2 Array Variables 6
 1.6.3 Space Requirements 6
1.7 Type Conversion 6
1.8 Expressions and Operations 7
 1.8.1 Arithmetic Operators 8
  1.8.1.2 Integer Division and Modulus Arithmetic 9
  1.8.1.3 Overflow and Division by Zero 9
 1.8.2 Relational Operators 9
 1.8.3 Logical Operators 10
 1.8.4 Functional Operators 12
 1.8.5 String Operations 12
1.9 Input Editing 13
1.10 Error Messages 13

2. MS-BASIC Commands and Statements

2.1 AUTO 15
2.2 CALL 16
2.3 CHAIN 17
2.4 CLEAR 18
2.5 CLOSE 19
2.6 COMMON 20
2.7 CONT 21
2.8 DATA 22
2.9 DEF FN 23
2.10 DEF INT/SNG/DBL/STR 24
2.11 DEF SEG 24
2.12 DEF USR 25
2.13 DELETE 26
2.14 DIM 26
2.15 EDIT 27
2.16 END 30
2.17 ERASE 30
2.18 ERR and ERL Variables 31
2.19 ERROR 31

2.20 FIELD 32
2.21 FOR . . . NEXT 33
2.22 GET 34
2.23 GOSUB . . . RETURN 35
2.24 GOTO 35
2.25 IF . . . THEN [. . . ELSE] and IF . . . GOTO 36
2.26 INPUT 37
2.27 INPUT# 38
2.28 KILL 39
2.29 LET 40
2.30 LINE INPUT 40
2.31 LINE INPUT# 41
2.32 LIST 41
2.33 LLIST 42
2.34 LOAD 43
2.35 LPRINT and LPRINT USING 43
2.36 LSET AND RSET 43
2.37 MERGE 44
2.38 MID$. 45
2.39 NAME 45
2.40 NEW 46
2.41 NULL 46
2.42 ON ERROR GOTO 46
2.43 ON . . . GOTO and ON . . . GOSUB 47
2.44 OPEN 47
2.45 OPTION BASE 48
2.46 OUT 48
2.47 POKE 49
2.48 PRINT 49
2.49 PRINT USING 51
2.50 PRINT# AND PRINT# USING 54
2.51 PUT 56
2.52 RANDOMIZE 57
2.53 READ 57
2.54 REM 58
2.55 RENUM 59
2.56 RESTORE 60
2.57 RESUME 61
2.58 RUN 61
2.59 SAVE 62
2.60 STOP 63
2.61 SWAP 63
2.62 TRON/TROFF 64
2.63 WAIT 65
2.64 WHILE . . . WEND 65
2.65 WIDTH 66
2.66 WRITE 67
2.67 WRITE# 67

3. MS-BASIC Functions

3.1 ABS 69
3.2 ASC 69
3.3 ATN 70
3.4 CDBL 70
3.5 CHR$. 70
3.6 CINT 71

3.7 COS 71
3.8 CSNG 72
3.9 CVI, CVS, CVD 72
3.10 DATE$ 72
3.11 EOF 73
3.12 EXP 73
3.13 FIX 74
3.14 FRE 74
3.15 HEX$ 75
3.16 INKEY$ 75
3.17 INP 75
3.18 INPUT$ 76
3.19 INSTR 76
3.20 INT 77
3.21 LEFT$ 77
3.22 LEN 78
3.23 LOC 78
3.24 LOG 78
3.25 LPOS 78
3.26 MID$ 79
3.27 MKI$, MKS$, MKD$ 79
3.28 OCT$ 80
3.29 PEEK 80
3.30 POS 80
3.31 RIGHT$ 80
3.32 RND 81
3.33 SGN 81
3.34 SIN 82
3.35 SPACE$ 82
3.36 SPC 82
3.37 SQR 83
3.38 STR$ 83
3.39 STRINGS$ 83
3.40 TAB 84
3.41 TAN 84
3.42 TIME$ 85
3.43 USR 85
3.44 VAL 85
3.45 VARPTR 86

Appendixes

APPENDIX A: CONVERTING PROGRAMS TO MS-BASIC

A.1 String Dimensions 87
A.2 Multiple Assignments 87
A.3 Multiple Statements 88
A.4 MAT Functions 88

APPENDIX B: MS-BASIC DISK I/O

B.1 Program File Commands 89
B.2 Protected File 90
B.3 Disk Data Files - Sequential and Random I/O 90

APPENDIX C: USING MS-BASIC WITH THE CP/M-86 OPERATING SYSTEM

C.1 Initialization 98
C.2 Disk Files 99
C.3 Files Command 99
C.4 Reset Command 100
C.5 LOF Function 100
C.6 EOF 100
C.7 Miscellaneous 100

APPENDIX D: USING MS-BASIC WITH THE MS-DOS OPERATING SYSTEM

D.1 Initialization 101
D.2 Disk Files 102
D.3 Files Command 102
D.4 Reset Command 103
D.5 LOF Function 103
D.6 EOF 103
D.7 Miscellaneous 103

APPENDIX E: ASSEMBLY LANGUAGE SUBROUTINES

E.1 Memory Allocation 104
E.2 Using the CALL Statement 104
E.3 Using USR Function Calls 108

APPENDIX F: MS-BASIC COMPILER

F.1 Operational Differences 111
F.2 Language Differences 112
F.3 Expression Evaluation 114
F.4 Integer Variables 115

APPENDIX G: SUMMARY OF ERROR CODES AND ERROR MESSAGES 116

APPENDIX H: MATHEMATICAL FUNCTIONS 121

APPENDIX I: ASCII CHARACTER CODES 122

FIGURES AND TABLES

FIGURES

B-1 Creating a Sequential Data File 91
B-2 Accessing a Sequential File 91
B-3 Adding Data to a Sequential File 92
B-4 Creating a Random File 94
B-5 Accessing a Random File 95
B-6 Example Program Using a Random Access File 96
E-1 Stack Layout When Call Statement is Activated 105
E-2 Stack Layout During Execution of a Call Statement 106

TABLES

1-1 MS-BASIC Special Characters and Terminal Keys 2
1-2 Single- and Double-Precision Form Constants Examples 5
1-3 Precedence Order of Arithmetic Variables 8
1-4 Algebraic Expressions and Their MS-BASIC Counterparts 8
1-5 Relational Operators 9
1-6 Outcomes of Logical Operations 10

1. GENERAL INFORMATION ABOUT MS-BASIC

1.1 INTRODUCTION

MS-BASIC is the most extensive implementation of MS-BASIC available for microprocessors. MS-BASIC meets the ANSI qualifications for MS-BASIC, as set forth in document BSRX3.60-1978. Each release of MS-BASIC is compatible with previous versions.

The manual is divided into three large chapters and nine appendixes. Chapter 1 covers a variety of topics, largely pertaining to data representation in MS-BASIC. Chapter 2 contains the syntax and semantics of every command and statement in MS-BASIC, ordered alphabetically. Chapter 3 describes all of MS-BASIC's intrinsic functions, also ordered alphabetically. The appendixes tell how to use MS-BASIC on the computer and its two operating systems, CP/M-86 and MS-DOS. They also contain a list of error messages and codes, a list of mathematical functions, and a list of ASCII character codes.

1.2 MODES OF OPERATION

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

In the direct mode, MS-BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using MS-BASIC 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.

1.3 LINE FORMAT AND LINE NUMBERS

Program lines in a MS-BASIC program have the following format (square brackets indicate something that is optional):

nnnnn BASIC statement [:BASIC statement . . .] <RET>

At the programmer's option, more than one MS-BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon. A MS-BASIC program line always begins with a line number, ends with a carriage return, and may contain up to of 255 characters.

It is possible to extend a logical line over more than one physical line by use of the ALT-J key. ALT-J lets you continue typing a logical line on the next physical line without entering a <RET>

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

1.4 CHARACTER SET

The MS-BASIC character set is comprised of alphabetic characters, numeric characters and special characters. The alphabetic characters in MS-BASIC are the uppercase and lowercase letters of the alphabet. The numeric characters are the digits 0 through 9.

1.4.1 SPECIAL CHARACTERS AND TERMINAL KEYS

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

Table 1-1: MS-BASIC 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 parentheses
)	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
backspace	Deletes last character typed.
<escape>	Escapes Edit Mode subcommands. See Section 2.16.

<tab>	Moves print position to next tab stop. Tab stops are every eight columns.
ALT-J	Moves to next physical line.
<carriage return>	Terminates input of a line.

1.4.2 ALTERNATE CHARACTERS

The following alternate characters are in MS-BASIC:

ALT-A	Enters Edit Mode on the line being typed.
ALT-C	Interrupts program execution and returns to MS-BASIC command level.
ALT-G	Rings the bell at the terminal.
ALT-H	Backspace. Deletes the last character typed.
ALT-I	Tab. Tab stops are every eight columns.
ALT-R	Retypes the line that is currently being typed.
ALT-S	Suspends program execution.
ALT-Q	Resumes program execution after ALT-S.
ALT-U	Deletes the line that is currently being typed.

1.5 CONSTANTS

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

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

Example:

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

Numeric constants are positive or negative numbers. Numeric constants in MS-BASIC 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 constants	Positive or negative real numbers, i.e., numbers that contain decimal points.
3.	Floating Point constants	Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or

fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10-38 to 10+38.

Examples:

235.9881E-7 = .00002359881
2359E6 = 2359000000

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

4. Hex constants — Hexadecimal numbers with the prefix &H.

Examples:

&H76
&H32F

5. Octal constants — Octal numbers with the prefix &O or &.

Examples:

&O347
&1234

1.5.1 SINGLE-AND DOUBLE-PRECISION FORM FOR NUMERIC CONSTANTS

Numeric constants may be either single-precision or double-precision numbers. Single-precision numeric constants are stored with 7 digits of precision, and printed with up to 7 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 (!)

A double-precision constant is any numeric constant that has:

- Eight or more digits
- Exponential form using D
- A trailing number sign (#)

Table 1-2: Single- and Double-Precision Form Constants Examples

SINGLE-PRECISION CONSTANTS	DOUBLE-PRECISION CONSTANTS
46.8	345692811
-1.09E-06	-1.09432D-06
3489. 0	3489.0#
22.5!	7654321.1234

1.6 VARIABLES

Variables are names used to represent values that are used in a MS-BASIC program. The value of a variable may be assigned while designing a program, or it may be assigned as the results of calculations performed by a program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 VARIABLE NAMES AND DECLARATION CHARACTERS

MS-BASIC variable names may be any length up to 40 characters. A variable name may contain letters and numbers, and the decimal point. The first character must be a letter. Special type declaration characters are also allowed — see below.

A variable name may not 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 BASIC commands, statements, function names and operator names.

Variables may represent either a numeric value or a string.

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

Numeric variable names may declare integer, single- or double-precision values. The 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 MS-BASIC variable names follow.

EXAMPLES:

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 MS-BASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Section 2.10.

1.6.2 ARRAY VARIABLES

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 285. The maximum number of elements per dimension is 32767.

1.6.3 SPACE REQUIREMENTS

VARIABLES:		BYTES
	Integer	2
	Single-precision	4
	Double-precision	8

ARRAYS:		BYTES
	Integer	2 per element
	Single-precision	4 per element
	Double-precision	8 per element

STRINGS:

3 bytes overhead plus the present contents of the string.

1.7 TYPE CONVERSION

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

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

Example:

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

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

Examples:

10 D# = 6#/7 20 PRINT D# RUN .857142857142857 1	The arithmetic was performed in double precision and the result was returned in D# as a double-precision value.
10 D = 6#/7 20 PRINT D RUN .8571429	The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded and printed as a single-precision value.

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

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

Example:

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

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

Example:

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

1.8 EXPRESSIONS AND OPERATORS

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

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

1. Arithmetic

2. Relational

3. Logical
4. Functional

1.8.1 ARITHMETIC OPERATORS

The order of precedence of arithmetic operators is shown in Table 1-3:

Table 1-3: Precedence Order of Arithmetic Operators

OPERATOR	OPERATION	SAMPLE EXPRESSION
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, floating point division	X*Y X/Y
+,-	Addition, subtraction	X+Y

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

Here are some sample algebraic expressions and their MS-BASIC counterparts:

Table 1-4: Algebraic Expressions and Their MS-BASIC Counterparts

ALGEBRAIC EXPRESSION	MS-BASIC EXPRESSION
X+2Y	X+Y*2
X-Y/Z	X-Y/Z
X (Y/Z)	X*Y/Z
$\frac{X + Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
$X^{(Y^Z)}$	X^(Y^Z)
X(-Y)	X*(-Y)

NOTE: Two consecutive operators must be separated by parentheses.

1.8.1.1 Integer Division and Modulus Arithmetic

Two additional operators available in MS-BASIC are integer division and modulus arithmetic.

Integer division is denoted by the backslash (\), ALT-+. 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.

Example:

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

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

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

Example:

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

The precedence of modulus arithmetic is just after integer division.

1.8.1.2 Overflow and Division By Zero

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

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

1.8.2 RELATIONAL OPERATORS

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See IF, Section 2.25.)

Table 1-5: 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

NOTE: The equal sign is also used to assign a value to a variable. See LET, Section 2.29.

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

Example:

```
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.

Example:

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

1.8.3 LOGICAL OPERATORS

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

Table 1-6: Outcomes of Logical Operations

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

EQV

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

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
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 (see IF, Section 2.26).

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

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

Example:

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 1000, so 15 AND 14 = 14 (binary 1110)

-1 AND 8=8 — -1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8

4 OR 2=6 — 4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)

10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 FUNCTIONAL OPERATORS

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

MS-BASIC also allows "user defined" functions that are written by the programmer. (See DEF FN, Section 2.9.)

1.8.5 STRING OPERATIONS

Strings may be concatenated using +.

Example:

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

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

```
=  <>  <  >  <=  >=
```

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

Examples:

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

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

1.9 INPUT EDITING

If an incorrect character is entered while typing a line, it can be deleted with the BACKSPACE key or with ALT-H. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type ALT-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. MS-BASIC will automatically replace the old line with the new line.

Section 2.15 "EDIT" describes more sophisticated editing capabilities provided in MS-BASIC.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Section 2.40.) NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

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

2. MS-BASIC COMMANDS AND STATEMENTS

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

FORMAT:

Shows the correct format for the instruction. See below for format notation.

PURPOSE:

Tells what the instruction is used for.

REMARKS:

Describes in detail how the instruction is used.

EXAMPLE:

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

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

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

2.1 AUTO

FORMAT:

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

PURPOSE:

Generates a line number automatically after each carriage return.

REMARKS:

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

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. Typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing ALT-C. The line in which ALT-C is command level.

EXAMPLE:

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

2.2 CALL

FORMAT:

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

variable name contains the segment offset that is the starting point in memory of the subroutine being CALLed. Note that the variable name must be assigned to the segment offset before the CALL statement is issued (see example below).

argument list contains the variables or constants, separated by commas, that are to be passed to the routine.

PURPOSE:

Calls an assembly language subroutine.

REMARKS:

The CALL statement is the recommended way of calling 8086 machine language programs with MS-BASIC. It is suggested that the old style user-call USR(n) not be used. See Appendix E for comparison of the two methods and for a complete description of using 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 the segment offset specified by the <*variable name*> portion of the CALL statement. Values are returned to MS-BASIC by including the variable name which will receive the result in the <*argument list*>.

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. MS-BASIC follows the rules described for the MEDIUM case.

EXAMPLE:

```
100 DEF SEG=&H8000
110 FOO=&H7FA
120 CALL FOO (A,B$,C)
.
.
.
```

Line 100 sets the segment address to 8000 Hex. The variable FOO is set to &H7FA, so that the call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

2.3 CHAIN

FORMAT:

CHAIN [MERGE] <*filename*>[,[<*line number exp*>] [,ALL][,DELETE<*range*>]]

PURPOSE:

Calls a program and passes 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. (See Section 2.6.)

EXAMPLE:

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

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

EXAMPLE:

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

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

EXAMPLE:

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

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

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

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

The MS-BASIC compiler does not support the ALL, MERGE, DELETE, and <*line number exp*> options to CHAIN. Thus, the statement format is CHAIN <*filename*>. If you wish to maintain compatibility with the MS-BASIC compiler, it is recommended that COMMON be used to pass variables and that overlays not be used. The CHAIN statement leaves the files open during CHAINing.

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

2.4 CLEAR

FORMAT:

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

PURPOSE:

Sets all numeric variables to zero, all string variables to null, and closes all open files; and, optionally, sets 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 MS-BASIC.

<expression2> sets aside stack space for MS-BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

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

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

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

EXAMPLES:

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```

2.5 CLOSE

FORMAT:

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

PURPOSE:

Concludes I/O to a disk file.

REMARKS:

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

The association between a particular file and file number ends upon executing a CLOSE. The file may then be reOPENed using the same or a different file number. Likewise, that file number may now be reused to OPEN any file.

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

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

EXAMPLE:

See Appendix B, "MS-BASIC Disk I/O."

2.6 COMMON

FORMAT:

COMMON <*list of variables*>

PURPOSE:

Passes variables to a CHAINed program.

REMARKS:

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear 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: The MS-BASIC Compiler supports a modified version of the COMMON statement. The COMMON statement must appear in a program 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 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with FORTRAN and assembly language routines without having to explicitly pass parameters in the CALL statement.

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

EXAMPLE:

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

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

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

2.7 CONT

FORMAT:

CONT

PURPOSE:

Continues program execution after an Alt-C has been typed, or a STOP or END statement has been executed.

REMARKS:

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

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

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

EXAMPLE:

```
10 Input A, B, C
20 K=A^2*5.3:L=8^31.26
30 STOP
40 M=C*K+100:Print M
RUN
```

```
? 1,2,3
BREAK IN 30
Ok
Print L
  30.7692
Ok
CONT
  115.9
Ok
```

2.8 DATA

FORMAT:

DATA ***<list of constants>***

PURPOSE:

Stores the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section 2.53)

REMARKS:

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

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

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

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.56).

EXAMPLES:

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

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

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

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

2.9 DEF FN

FORMAT:

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

PURPOSE:

Defines and names a function that is written by the user.

REMARKS:

<*name*> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <*parameter list*> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are 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 will be given in the function call.

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

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

EXAMPLE:

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

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

2.10 DEFINT/SNG/DBL/STR

FORMAT:

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

where <type> is INT, SNG, DBL, or STR.

PURPOSE:

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

REMARKS:

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

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

EXAMPLES:

10 DEFDBL L-P	All variables beginning with the letters L, M, N, O, and P will be double-precision variables.
10 DEFSTR A	All variables beginning with the letter A will be string variables.
10 DEFINT I-N,W-Z	All variable beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

2.11 DEF SEG

FORMAT:

DEF SEG [=<*address*>]

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

PURPOSE:

Assigns the current segment address to be referenced by a subsequent CALL (see Section 2.2), a USR function call, or a PEEK or POKE statement.

REMARKS:

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

Entry of any value outside the *<address>* range 0-65535 will result in an "Illegal Function Call" error, and the previous value will be retained.

If the *<address>* option is omitted, the segment to be used is set to the MS-BASIC data segment (DS). This is the initial default value.

If the *<address>* option is given, it should be based on a 16-byte boundary. For PEEK, POKE, or CALL statements, the value is shifted left 4 bits (this is done by the microprocessor, not by MS-BASIC) to form the code segment address for the subsequent call instruction. BASIC-86 does not perform additional checking to assure that the resultant segment address is valid.

DEF and SEG *MUST* be separated by a space. Otherwise, MS-BASIC 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 MS-BASIC DS

2.12 DEF USR

FORMAT:

DEF USR[<*digit*>]=<*Integer expression*>

PURPOSE:

Specifies the starting address of an assembly language subroutine.

REMARKS:

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

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

EXAMPLE:

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

2.13 DELETE

FORMAT:

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

PURPOSE:

Deletes program lines.

REMARKS:

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

EXAMPLES:

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

2.14 DIM

FORMAT:

DIM <*list of subscripted variables*>

PURPOSE:

Specifies the maximum values for array variable subscripts and allocates storage accordingly.

REMARKS:

If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.45).

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

EXAMPLE:

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

2.15 EDIT

FORMAT:

EDIT <*line number*>

PURPOSE:

Enters Edit Mode at the specified line.

REMARKS:

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

EDIT MODE SUBCOMMANDS

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

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

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

NOTE: In the descriptions that follow, <*ch*> represents any character, <*text*> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and ESC represents 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, the Backspace moves the cursor key one space to the left each time it is pressed. Characters are printed as you backspace over them.

2. Inserting Text

   I   I<*text*> inserts <*text*> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Backspace key on the terminal may be used to delete characters to the left of the cursor for each character that you backspace over. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Alt-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, type Escape or Carriage Return.

3. Deleting Text

   D   [i]D deletes i characters to the right of the cursor. The deleted characters are echoed 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 will stop at the end of the line. All characters passed over during the search are printed.

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

6. Ending and Restarting Edit Mode

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

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

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

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

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

NOTE: If MS-BASIC receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell character (Alt-G) and the command is ignored.

SYNTAX ERRORS

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

EXAMPLE:

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

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

ALT-A

Type an Alt-A to enter Edit Mode on the line you are currently typing. MS-BASIC responds with a carriage return, an exclamation point (!) and a space. The cursor will be 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." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

2.16 END

FORMAT:

END

PURPOSE:

Terminates program execution, closes all files and returns to command level.

REMARKS:

END statements may be placed anywhere in the program. 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. MS-BASIC always returns to command level after an END is executed.

EXAMPLE:

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

2.17 ERASE

FORMAT:

ERASE *<list of array variables>*

PURPOSE:

Eliminates arrays from a program.

REMARKS:

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

NOTE: The MS-BASIC compiler does not support ERASE.

EXAMPLE:

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

2.18 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 number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ... 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 may appear to the left of the equal sign in a LET (assignment) statement. The MS-BASIC error codes are listed in Appendix A.

2.19 ERROR

FORMAT:

ERROR *<integer expression>*

PURPOSE:

Simulates the occurrence of a MS-BASIC error; or allows 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 MS-BASIC (see Appendix G), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See first example.)

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

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

EXAMPLES:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
```

```
40 END
Ok
RUN
String too long in 30
```

Or, in direct mode:

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

2.20 FIELD

FORMAT:

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

PURPOSE:

Allocates space for variables in a random file buffer.

REMARKS:

A FIELD statement must be executed to get data out of a random buffer after a GET or to enter data before a PUT.

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

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

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

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

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

EXAMPLE:

See Appendix B.

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

2.21 FOR . . . NEXT

FORMAT:

FOR <*variable*>=x TO y [STEP z]

.
.
.

NEXT [<*variable*>][,<*variable*> . . .]

where x, y and z are numeric expressions.

PURPOSE:

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

REMARKS:

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

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

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

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

EXAMPLES:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
```

```
50 PRINT K
60 NEXT I
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

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

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

```
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. (Note: Previous versions of BASIC set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.)

2.22 GET

FORMAT:

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

PURPOSE:

Reads a record from a random disk file into a random buffer.

REMARKS:

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

EXAMPLE:

See Appendix B.

NOTE: After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

2.23 GOSUB ... RETURN

FORMAT:

GOSUB <*line number*>

.
.
.

RETURN

PURPOSE:

Branches to and returns from a subroutine.

REMARKS:

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

The RETURN statement(s) in a subroutine causes MS-BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, if logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be easily distinguishable from the main program. Putting a STOP, END or GOTO statement before a subroutine will direct program control around it, and prevents inadvertent entry into 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
```

2.24 GOTO

FORMAT:

GOTO <*line number*>

PURPOSE:

Branches 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
```

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

FORMATS:

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

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

IF <*expression*> GOTO <*line number*>

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

PURPOSE:

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

REMARKS:

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

NESTING OF IF STATEMENTS

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

EXAMPLES:

```
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.

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

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

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

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

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

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

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

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

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

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 1110.

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

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

2.26 INPUT

FORMAT:

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

PURPOSE:

Allows input from the terminal during program execution.

REMARKS:

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

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user 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 may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

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

Examples:

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

LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4      (User types 7.4)
THE AREA OF THE CIRCLE IS 171.9464

WHAT IS THE RADIUS?
etc.
```

2.27 INPUT

FORMAT:

INPUT#<*file number*>,<*variable list*>

PURPOSE:

Reads data items from a sequential disk file and assigns them to program variables.

REMARKS:

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

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

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

EXAMPLE:

See Appendix B.

2.28 KILL

FORMAT:

KILL <*filename*>

PURPOSE:

Deletes a file from disk.

REMARKS:

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

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

EXAMPLE:

```
200 KILL "DATA 1"
```

(See Appendix B)

2.29 LET

FORMAT:

[LET] <*variable*> = <*expression*>

PURPOSE:

Assigns the value of an expression to a variable.

REMARKS:

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

EXAMPLE:

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

or

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

2.30 LINE INPUT

FORMAT:

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

PURPOSE:

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

REMARKS:

The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <*string variable*>. 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 typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be by-passed by typing ALT-C. MS-BASIC will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

EXAMPLE:

See Example, Section 2.31, LINE INPUT#.

2.31 LINE INPUT#

FORMAT:

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

PURPOSE:

Reads an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

REMARKS:

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

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

EXAMPLE:

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

2.32 LIST

FORMAT 1:

LIST [<*line number*>]

FORMAT 2:

LIST [<*line number*>[-[<*line number*>]]]

PURPOSE:

Lists all or part of the program currently in memory at the terminal.

REMARKS:

MS-BASIC always returns to command level after a LIST is executed.

Format 1: If *<line number>* is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing ALT-C.) If *<line number>* is included, only the specified line will be 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.

EXAMPLE:

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.

2.33 LLIST

FORMAT:

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

PURPOSE:

Lists all or part of the program currently in memory at the line printer.

REMARKS:

LLIST assumes a 132-character wide printer.

MS-BASIC 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.

2.34 LOAD

FORMAT:

LOAD <*filename*>[,R]

PURPOSE:

Loads a file from disk into memory.

REMARKS:

<filename> is the name that was used when the file was SAVEd. For more information on file names and extensions see Appendix C if you are CP/M-86 and Appendix D, if you are using MS-DOS.

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

EXAMPLE:

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

2.35 LPRINT/LPRINT USING

FORMAT:

LPRINT [<*list of expressions*>]

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

PURPOSE:

Prints data at the line printer.

REMARKS:

Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.48 and Section 2.49.

LPRINT assumes a 132-character-wide printer.

2.36 LSET AND RSET

FORMAT:

LSET <*string variable*> = <*string expression*>

RSET <*string variable*> = <*string expression*>

PURPOSE:

Moves 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, Section 3.26.

Example:

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

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

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

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

2.37 MERGE

FORMAT:

MERGE *<filename>*

PURPOSE:

Merges a specified disk file into the program currently in memory.

REMARKS:

<filename> is the name used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. (See Appendix C if your operating system is CP/M-86, or Appendix D if you're using MS-DOS.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.)

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

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

EXAMPLE:

```
MERGE "NUMBRS"
```

2.38 MID$

FORMAT:

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

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

PURPOSE:

Replaces a portion of one string with another string.

REMARKS:

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

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. See Section 3.25.

2.39 NAME

FORMAT:

NAME <*old filename*> AS <*new filename*>

PURPOSE:

Changes the name of a disk file.

REMARKS:

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

EXAMPLE:

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

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

2.40 NEW

FORMAT:

NEW

PURPOSE:

Deletes the program currently in memory and clears all variables.

REMARKS:

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

2.41 NULL

FORMAT:

NULL *<integer expression>*

PURPOSE:

Sets 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 >=3. 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 will be printed after each line.

2.42 ON ERROR GOTO

FORMAT:

ON ERROR GOTO *<line number>*

PURPOSE:

Enables error trapping and specifies the first line of the error handling subroutine.

REMARKS:

Once error trapping has been enabled, all errors detected—including

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

NOTE: If an error occurs during execution of an error handling subroutine, the MS-BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

EXAMPLE:

```
10 ON ERROR GOTO 1000
```

2.43 ON . . . GOSUB AND ON . . . GOTO

FORMAT:

ON <*expression*> GOTO <*list of line numbers*>
ON <*expression*> GOSUB <*list of line numbers*>

PURPOSE:

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

REMARKS:

The value of <*expression*> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a noninteger, the fractional portion is rounded.) In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine. If the value of <*expression*> is zero or greater than the number of items in the list (but less than or equal to 255), MS-BASIC continues with the next executable statement. If the value of <*expression*> is negative or greater than 255, an "Illegal function call" error occurs.

EXAMPLE:

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

2.44 OPEN

FORMAT:

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

PURPOSE:

Allows I/O to a disk file.

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

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

O   specifies sequential output mode

I   specifies sequential input mode

R   specifies random input/output mode

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

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

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

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

EXAMPLE:

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

See also Appendix B.

2.45 OPTION BASE

FORMAT:

OPTION BASE n

where n is 1 or 0

PURPOSE:

Declares the minimum value for array subscripts.

REMARKS:

The default base is 0. If the statement

```
OPTION BASE 1
```

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

2.46 OUT

FORMAT:

OUT I,J

where I and J are integer expressions in the range 0 to 65535. I is a machine port number, and J is the data to be transmitted.

PURPOSE:

Sends a byte to a machine output port.

REMARKS:

The integer expression I is the port number, and the integer expression J is the data to be transmitted.

EXAMPLE:

```
100 OUT 12345,225
```

In assembly language, this is equivalent to:

```
MOV   DX,12345
MOV   AL,255
OUT   DX,AL
```

2.47 POKE

FORMAT:

POKE I,J

where I and J are integer expressions

PURPOSE:

Writes a byte into a memory location.

REMARKS:

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

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.28.

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
```

2.48 PRINT

FORMAT:

PRINT [<***list of expressions***>]

PURPOSE:

Outputs data at the terminal.

REMARKS:

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

PRINT POSITIONS

The position of each printed item is determined by the punctuation used to separate the items in the list. MS-BASIC 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 is printed at the end of the line. If the printed line is longer than the terminal width, MS-BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 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-17. A question mark may be used in place of the word PRINT in a PRINT statement.

Examples:

```
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.

```
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.

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

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

2.49 PRINT USING

FORMAT:

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

PURPOSE:

Prints strings or numbers using a specified format.

REMARKS AND EXAMPLES:

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

STRING FIELDS

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

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

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

EXAMPLE:

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

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

EXAMPLE:

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

NUMERIC FIELDS

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

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

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

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

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

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

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

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

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

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

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

68.95-   22.45   7.01-
```

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

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

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

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

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

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

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

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

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

^^^^ Four carets (or up-arrows) may 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 may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or – is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

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

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

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

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

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

The literal character itself may 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 is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

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

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

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

2.50 PRINT# AND PRINT# USING

FORMAT:

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

PURPOSE:

Writes data to a sequential disk 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 Section 2.49, PRINT USING. The expressions in <*list of expressions*> are the numeric and/or string expressions that will be written to the file.

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

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

EXAMPLE:

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

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

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

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

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

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

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

The image written to disk is:

```
CAMERA,93604-1
```

which can be read back into two string variables.

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

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

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

would write the following image to disk:

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

and the statement

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

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

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

writes the following image to disk:

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

and the statement

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

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

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

EXAMPLE:

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

See Appendix B. See also WRITE#, Section 2.67.

2.51 PUT

FORMAT:

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

PURPOSE:

Writes a record from a random buffer to a random disk file.

REMARKS:

<*file number*> is the number under which the file was OPENed. If <*record number*> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

EXAMPLE:

See Appendix B.

NOTE: PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

With WRITE#, MS-BASIC pads the buffer with spaces up to the carriage return. Any to read or write past the end of the buffer causes a "Field overflow" error.

2.52 RANDOMIZE

FORMAT:

RANDOMIZE [<*expression*>]

PURPOSE:

Reseeds the random number generator.

REMARKS:

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

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

before executing RANDOMIZE.

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

EXAMPLE:

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

2.53 READ

FORMAT:

READ <*list of variables*>

PURPOSE:

Reads values from a DATA statement and assigns them to variables. (See DATA, Section 2.9.)

REMARKS:

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

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

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Section 2.56).

EXAMPLE:

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

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

EXAMPLE:

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

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

2.54 REM

FORMAT:

REM <*remark*>

PURPOSE:

Allows explanatory remarks to be inserted in a program.

REMARKS:

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

REM statements may be branched into from a GOTO or GOSUB statement. Execution will continue the first executable statement after the REM statement.

Remarks may 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 in 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
      .
      .
      .
```

2.55 RENUM

FORMAT:

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

PURPOSE:

Renumbers 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.

2.56 RESTORE

FORMAT:

RESTORE [<*line number*>]

PURPOSE:

Allows 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 program's first DATA statement. If <*line number*> is specified, the next READ statement accesses the first item in the specified DATA statement.

EXAMPLE:

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

2.57 RESUME

FORMATS:

RESUME

RESUME 0

RESUME NEXT

RESUME *<line number>*

PURPOSE:

Continues program execution after an error recovery procedure has been performed.

REMARKS:

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

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

RESUME *<line number>* Execution resumes at *<line number>*.

A RESUME statement 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
      .
      .
      .
```

2.58 RUN

FORMAT 1:

RUN [*<line number>*]

PURPOSE:

Executes the program currently in memory.

REMARKS:

If *<line number>* is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. MS-BASIC always returns to command level after a RUN is executed.

EXAMPLE:

```
RUN
```

FORMAT 2:

RUN <*filename*>[,R]

PURPOSE:

Loads a file from disk into memory and runs it.

REMARKS:

<*filename*> is the name used when the file was SAVEd. See Appendix C for CP/M-86 and Appendix D for MS-DOS.

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 Appendix B.

NOTE: The MS-BASIC Compiler supports the RUN and RUN <*line number*> forms of the RUN statement. The MS-BASIC Compiler does not support the "R" option with RUN. If you want this feature, the CHAIN statement should be used.

2.59 SAVE

FORMAT:

SAVE <*filename*>[,A | ,P]

PURPOSE:

Saves a program file on disk.

REMARKS:

<*filename*> is a quoted string that conforms to your operating system's requirements for filenames. Your operating system may append a default filename extension if one was not supplied in the SAVE command. See Appendix C if you are using CP/M-86, or Appendix D if your operating system is MS-DOS. If <*filename*> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, MS-BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires and ASCII format file, and some operating system commands such as LIST may 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.

Examples:

```
SAVE"COM2",A

SAVE"PROG",P
```

See also Appendix B, "MS-BASIC Disk I/O."

2.60 STOP

FORMAT:

STOP

PURPOSE:

Terminates program execution and returns to command level.

REMARKS:

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

```
Break in line nnnnn
```

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

MS-BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.7).

EXAMPLE:

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

2.61 SWAP

FORMAT:

SWAP ***<variable>,<variable>***

PURPOSE:

Exchanges the values of two variables.

REMARKS:

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

EXAMPLE:

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

2.62 TRON/TROFF

FORMAT:

TRON

TROFF

PURPOSE:

Traces 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
```

2.63 WAIT

FORMAT:

WAIT <*port number*>, I[,J]

where I and J are integer expressions

PURPOSE:

Suspends 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 ORed with the integer expression J, and then ANDed with I. If the result is zero, MS-BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

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

EXAMPLE:

```
100 WAIT 32,2
```

2.64 WHILE . . . WEND

FORMAT:

WHILE <*expression*>

.

.

[<*loop statements*>]

.

.

WEND

PURPOSE:

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

REMARKS:

If <*expression*> is not zero (i.e., true), <*loop statements*> are executed until the WEND statement is encountered. MS-BASIC 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 may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

EXAMPLE:

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

2.65 WIDTH

FORMAT:

WIDTH [LPRINT] <*integer expression*>

PURPOSE:

Sets the printed line width (in number of characters) for the terminal or line printer.

REMARKS:

If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer. <*integer expression*> must have a value in the range 15 to 255. The default width is 72 characters.

If <*integer expression*> is 255, the line width is "infinite;" that is, MS-BASIC 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
```

2.66 WRITE

FORMAT:

WRITE[<*list of expressions*>]

PURPOSE:

Outputs data at the terminal.

REMARKS:

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

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

WRITE outputs numeric values using the same format as the PRINT statement, Section 2.48.

EXAMPLE:

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

2.67 WRITE#

FORMAT:

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

PURPOSE:

Writes 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 the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

EXAMPLE:

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

WRITE#1,A$,B$

writes the following image to disk:

"CAMERA","93604-1"

A subsequent INPUT# statement, such as:

INPUT#1,A$,B$

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

3. MS-BASIC FUNCTIONS

This chapter discusses the intrinsic functions provided by MS-BASIC are presented in this chapter. The functions may 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 have been 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, MS-BASIC will round the fractional portion and use the resulting integer.

NOTE: With the MS-BASIC interpreter, only integer and single precision results are returned by the functions described in this chapter. Double precision functions are supported only by the MS-BASIC Compiler.

3.1 ABS

FORMAT:

ABS(X)

ACTION:

Returns the absolute value of the expression X.

EXAMPLE:

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

3.2 ASC

FORMAT:

ASC(X$)

ACTION:

Returns a numerical value that is the ASCII code of the first character

of the string X$. (See Appendix D for ASCII codes.) If X$ 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.

3.3 ATN

FORMAT:

ATN(X)

ACTION:

Returns 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
```

3.4 CDBL

FORMAT:

CDBL(X)

ACTION:

Converts X to a double precision number.

EXAMPLE:

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

3.5 CHR$

FORMAT:

CHR$(I)

ACTION:

Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix D.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a terminal screen and return the cursor to the home position.

EXAMPLE:

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

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

3.6 CINT

FORMAT:

CINT(X)

ACTION:

Converts 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 type. See also the FIX and INT functions, both of which return integers.

3.7 COS

FORMAT:

COS(X)

ACTION:

Returns 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
```

3.8 CSNG

FORMAT:

CSNG(X)

ACTION:

Converts X to a single precision number.

EXAMPLE:

```
10 A# = 975.34217#
20 PRINT A#; CSNG(A#)
RUN
 975.34217 975.341
Ok
```

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

3.9 CVI, CVS, CVD

FORMAT:

CVI(<*2-byte string*>)

CVS(<*4-byte string*>)

CVD(<*8-byte string*>)

ACTION:

Converts string values to numeric values. Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

EXAMPLE:

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

See also Appendix B.

3.10 DATE$

FORMAT:

DATE$

ACTION:

Sets or retrieves the current date. Returns a ten-character string variable with the following format:

mm-dd-yyyy

where: mm = month (1-12)
dd = date (1-31)
yyyy= year (1980-99)
\- = delimiter
/ = delimiter

Leading zeros are presumed for single-digit months and dates. Double-digit years are presumed to begin with 19, so that specifying 99 indicates the year 1999. DATE$ can be used like any string variable, although it is constantly being incremented by a hardware clock. Note that either of the common delimiters "/" or "-" can be used between digits.

NOTE: DATE$ does not support the European format dd-mm-yy.

EXAMPLE:

```
DATE$ = "10-21-82"
Ok
PRINT DATE$
10-21-1982
Ok
```

3.11 EOF

FORMAT:

EOF(<*file number*>)

ACTION:

Returns -1 (true) if the end of a sequential 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
  .
  .
  .
```

3.12 EXP

FORMAT:

EXP(X)

ACTION:

Returns e to the power of X. X must be <=87.3365. 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
54.5982
Ok
```

3.13 FIX

FORMAT:

FIX(X)

ACTION:

Returns 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
```

3.14 FRE

FORMAT:

FRE(0)

FRE(X$)

ACTION:

Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by MS-BASIC.

FRE("") forces a garbage collection before returning the number of free bytes. BE PATIENT: garbage collection may take 1 to 1-1/2 minutes. MS-BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

EXAMPLE:

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

3.15 HEX$

FORMAT:

HEX$(X)

ACTION:

Returns 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.

3.16 INKEY$

FORMAT:

INKEY$

ACTION:

Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed and all characters are passed through to the program except for ALT-C, which terminates the program. (With the MS-BASIC Compiler, ALT-C is also passed through to the program.)

EXAMPLE:

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

3.17 INP

FORMAT:

INP(I)

where I is a valid machine port number in the range 0 to 65535.

ACTION:

Returns the byte read from port I.

REMARKS:

INP is the complementary function to the OUT statement.

EXAMPLE:

```
100 A=INP (54321)
```

In assembly language, this is equivalent to:

```
MOV    DX,54321
 IN    AL,DX
```

3.18 INPUT$

FORMAT:

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

ACTION:

Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all ALT characters are passed through except ALT-C, which is used to interrupt the execution of the INPUT$ function.

Example:

```
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
      .
      .
      .
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
      .
      .
      .
```

3.19 INSTR

FORMAT:

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

ACTION:

Searches 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$ may 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, error message "Illegal function call in <line number>" will be returned.

3.20 INT

FORMAT:

INT(X)

ACTION:

Returns 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.

3.21 LEFT$

FORMAT:

LEFT$(X$,I)

ACTION:

Returns 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$) will be returned. If I=0, the null string (length zero) is returned.

EXAMPLE:

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

Also see the MID$ and RIGHT$ functions.

3.22 LEN

FORMAT:

LEN(X$)

ACTION:

Returns the number of characters in X$. Nonprinting characters and blanks are counted.

EXAMPLE:

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

3.23 LOC

FORMAT:

LOC(<*file number*>)

ACTION:

With random disk files, LOC returns the record number just read or written from a GET or PUT. If the file was opened but no disk 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
```

3.24 LOG

FORMAT:

LOG(X)

ACTION:

Returns the natural logarithm of X. X must be greater than zero.

EXAMPLE:

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

3.25 LPOS

FORMAT:

LPOS(X)

ACTION:

Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

EXAMPLE:

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

3.26 MID$

FORMAT:

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

ACTION:

Returns a string of 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 >rLEN(x$), MID$ returns a null string.

EXAMPLE:

```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
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>" will be returned.

3.27 MKI$, MKS$, MKD$

FORMAT:

MKI$(<*integer expression*>)

MKS$(<*single-precision expression*>)

MKD$(<*double-precision expression*>)

ACTION:

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 a 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 Appendix B.

3.28 OCT$

FORMAT:

OCT$(X)

ACTION:

Returns 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.

3.29 PEEK

FORMAT:

PEEK(I)

ACTION:

Returns the byte (decimal integer in the range 0 to 255) read from memory location I. I must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement, Section 2.47.

EXAMPLE:

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

3.30 POS

FORMAT:

POS(I)

ACTION:

Returns 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.

3.31 RIGHT$

FORMAT:

RIGHT$(X$,I)

ACTION:

Returns 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$,8)
RUN
BASIC
Ok
```

Also see the MID$ and LEFT$ functions.

3.32 RND

FORMAT:

RND[(X)]

ACTION:

Returns 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 RANDOMIZE, Section 2.52). 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 I
```

Will print 5 random numbers between 0 and 100.

3.33 SGN

FORMAT:

SGN(X)

ACTION:

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)+2 GOTO 100,200,300
```

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

3.34 SIN

FORMAT:

SIN(X)

ACTION:

Returns 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
```

3.35 SPACE$

FORMAT:

SPACE$(X)

ACTION:

Returns 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.

3.36 SPC

FORMAT:

SPC(I)

ACTION:

Prints I blanks on the terminal. 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.

3.37 SQR

FORMAT:

SQR(X)

ACTION:

Returns 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
```

3.38 STR$

FORMAT:

STR$(X)

ACTION:

Returns 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.

3.39 STRING$

FORMATS:

STRING$(I,J)

STRING$(I,X$)

ACTION:

Returns 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
```

3.40 TAB

FORMAT:

TAB(I)

ACTION:

Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to the same position on the next line. 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

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
```

3.41 TAN

FORMAT:

TAN(X)

ACTION:

Returns 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
```

3.42 TIME$

FORMAT:

TIME$

ACTION:

Sets or retrieves the current time. Returns an eight-character string variable with the following format:

hh:mm:ss

where: hh = hour of day (0-23)
mm = minutes (0-59)
ss = seconds (0-59)
: = delimiter

Minutes and seconds are optional, and the time defaults to "00:00:00". Leading zeros are optional. TIME$ can be used like any string variable, although it is being incremented constantly by a hardware clock.

EXAMPLE:

```
TIME$ = "08:00"
Ok
PRINT TIME$
08:00:04
Ok
```

3.43 USR

Format:

USR[<*digit*>](X)

ACTION:

Calls the user's assembly language subroutine with the argument X. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix E.

EXAMPLE:

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

3.44 VAL

FORMAT:

VAL(X$)

Returns the numerical value of string X$. The VAL function also strips

leading blanks, tabs, and linefeeds from the argument string. For example:

```
VAL(" -3)
```

returns -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.

3.45 VARPTR

FORMAT 1:

VARPTR(<*variable name*>)

FORMAT 2:

VARPTR(#<*file number*>)

ACTION:

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

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(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 disk 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))
```

If you have programs written in a BASIC other than MS-BASIC, some minor adjustments may be necessary before running them. Here are some specific things to look for when converting BASIC programs.

A.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the MS-BASIC statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for MS-BASIC string concatenation.

In MS-BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

OTHER BASIC	MS-BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

OTHER BASIC	MS-BASIC
A$(I)=X$	MID$(A$,1,1)=X$
A$(I,J9)=X$	MID$(A$,I,J-I+1)=X$

A.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

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

to set B and C equal to zero. MS-BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

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

A.3 MULTIPLE STATEMENTS

Some BASICs use a backslash \ to separate multiple statements on a line. With MS-BASIC, be sure all statements on a line are separated by a colon (:).

A.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR . . . NEXT loops to execute properly.

Disk I/O procedures for the beginning MS-BASIC user are examined in this appendix. If you are new to MS-BASIC or if you're getting disk-related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. (See Appendix C for CP/M-86 and Appendix D for MS-DOS.)

B.1 PROGRAM FILE COMMANDS

Here is a review of the commands and statements used in program file manipulation.

SAVE <*filename*>[,A] — Writes to disk the program currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, MS-BASIC uses a compressed binary format.)

LOAD <*filename*>[,R] — Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files. (LOAD <*filename*>,R and RUN <*filename*>,R are equivalent.)

RUN <*filename*>[,R] — RUN <*filename*> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN <*filename*>,R and LOAD <*filename*>,R are equivalent.)

MERGE <*filename*> — Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines

have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and MS-BASIC returns to command level.

KILL *<filename>*

Deletes the file from the disk. *<filename>* may be a program file, or a sequential or random access data file.

NAME *<old filename>* AS*<new filename>*

To change the name of a disk file, execute the NAME statement, NAME *<oldfile>* AS *<newfile>*. NAME may be used with program files, random files, or sequential files.

B.2 PROTECTED FILE

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

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

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

B.3 DISK DATA FILES — SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a MS-BASIC program: sequential files and random access files.

B.3.1 SEQUENTIAL FILES

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes time to access the data. Data that is written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

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

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode. `OPEN "O",#1,"DATA"`
2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.) `PRINT#1,A$;B$;C$`
3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode. `CLOSE #1` `OPEN "I",#1,"DATA"`

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

Figure B-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

Figure B-1: Creating a Sequential Data File

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

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

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

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

NAME? etc.
```

Now look at Figure B-2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978.

Figure B-2: Accessing a Sequential File

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

The program in Figure B-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end"

error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

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

and change line 40 to GOTO 15.

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

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

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

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

B.3.1.1 Adding Data To A Sequential File

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

1. OPEN "NAMES" in "I" mode.

2. OPEN a second file called "COPY" in "O" mode.

3. Read in the data in "NAMES" and write it to "COPY".

4. CLOSE "NAMES" and KILL it.

5. Write the new information to "COPY".

6. Rename "COPY" as "NAMES" and CLOSE.

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

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

Figure B-3: Adding Data to a Sequential File

```
10  ON ERROR GOTO 2000
20  OPEN "I",#1,"NAMES"
30  REM IF FILE EXISTS, WRITE IT TO "COPY"
40  OPEN "O",#2,"COPY"
```

```
50  IF EOF(1) THEN 90
60  LINE INPUT#1,A$
70  PRINT#2,A$
80  GOTO 50
90  CLOSE #1
100  KILL "NAMES"
110  REM ADD NEW ENTRIES TO FILE
120  INPUT "NAME";N$
130  IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140  LINE INPUT "ADDRESS? ";A$
150  LINE INPUT "BIRTHDAY? ";B$
160  PRINT#2,N$
170  PRINT#2,A$
180  PRINT#2,B$
190  PRINT:GOTO 120
200  CLOSE
205  REM CHANGE FILENAME BACK TO "NAMES"
210  NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

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

B.3.2 RANDOM FILES

Creating and accessing random files requires more program steps than sequential files, but there are advantages to taking the extra trouble. One advantage is that random files require less room on the disk, because MS-BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.) The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk — it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

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

B.3.2.1 Creating A Random File

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

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes. — `OPEN "R",#1,"FILE",32`

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

   `FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$`

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.

   ```
   LSET N$=X$
   LSET A$=MKS$(AMT)
   LSET P$=TEL$
   ```

4. Write the data from the buffer to the disk using the PUT statement.

   `PUT #1,CODE%`

Figure B-4 takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Figure B-4: Creating a Random File

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

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

B.3.2.2 Accessing A Random File

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

1. OPEN the file in "R" mode.

   `OPEN "R",#1,"FILE",32`

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

   `FIELD #1 20 AS N$, 4 AS A$, 8 AS P$`

   NOTE: In a program that performs both input and output on the same random file, you can often use just

one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer. — `GET #1,CODE%`

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values. — `PRINT N$` `PRINT CVS(A$)`

The program in Figure B-5 accesses the random file "FILE" that was created in Figure B-4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed.

Figure B-5: Accessing a Random File

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

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

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

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

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

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

Figure B-6: Example Program Using a Random Access File

```
120  OPEN"R",#1,"INVEN.DAT",39
125  FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130  PRINT:PRINT "FUNCTIONS:":PRINT
135  PRINT 1,"INITIALIZE FILE"
140  PRINT 2,"CREATE A NEW ENTRY"
150  PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160  PRINT 4,"ADD TO STOCK"
170  PRINT 5,"SUBTRACT FROM STOCK"
180  PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220  PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225  IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
         "BAD FUNCTION NUMBER":GO TO 130
230  ON FUNCTION GOSUB 900,250,390,480,560,680
240  GOTO 220
250  REM BUILD NEW ENTRY
260  GOSUB 840
270  IF ASC(F$)>255 THEN INPUT"OVERWRITE";A$:
         IF A$>"Y" THEN RETURN
280  LSET F$=CHR$(0)
290  INPUT "DESCRIPTION";DESC$
300  LSET D$=DESC$
310  INPUT "QUANTITY IN STOCK";Q%
320  LSET Q$=MKI$(Q%)
330  INPUT "REORDER LEVEL";R%
340  LSET R$=MKI$(R%)
350  INPUT "UNIT PRICE";P
360  LSET P$=MKS$(P)
370  PUT#1,PART%
380  RETURN
390  REM DISPLAY ENTRY
400  GOSUB 840
410  IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420  PRINT USING "PART NUMBER ###";PART%
430  PRINT D$
440  PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450  PRINT USING "REORDER LEVEL #####";CVI(R$)
460  PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470  RETURN
480  REM ADD TO STOCK
490  GOSUB 840
500  IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510  PRINT D$:INPUT "QUANTITY TO ADD" ;A%
520  Q%=CVI(Q$)+A%
530  LSET Q$=MKI$(Q%)
540  PUT#1,PART%
550  RETURN
560  REM REMOVE FROM STOCK
570  GOSUB 840
580  IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590  PRINT D$
600  INPUT "QUANTITY TO SUBTRACT";S%
610  Q%=CVI(Q$)
620  IF (Q%-S%)<0 THEN PRINT "ONLY";Q%; "IN STOCK":GOTO 600
630  Q%=Q%-S%
640  IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
         " REORDER LEVEL";CVI(R$)
```

```
650  LSET Q$=MKI$(Q%)
660  PUT#1,PART%
670  RETURN
680  DISPLAY ITEMS BELOW REORDER LEVEL
690  FOR I=1 TO 100
710  GET#1,I
720  IF CVI(Q$)CVI(R$) THEN PRINT D$;" QUANTITY";
          CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730  NEXT I
740  RETURN
840  INPUT "PART NUMBER";PART%
850  IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART
          NUMBER": GOTO 840 ELSE GET#1,PART%:RETURN
890  END
900  REM INITIALIZE FILE
910  INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920  LSET F$=CHR$(255)
930  FOR I=1 TO 100
940  PUT#1,I
950  NEXT I
960  RETURN
```

APPENDIX C: Using MS-BASIC with the CP/M-86 Operating System

The CP/M-86 version of MS-BASIC is supplied on a standard 5¼ inch diskette. The name of the file is MSBASIC.COM for MS-DOS or MSBASIC.CMD for CP/M-86.

To run MSBASIC.CMD, bring up CP/M-86 and type the following:

```
A>MSBASIC<RET>
```

The system will reply:

```
MS-BASIC Version 5.xx
                [CP/M-86 Version]
        Copyright 1977-1982 (C) by Microsoft
                Created: dd-mmm-yy
                        xxxxx Bytes free
                        Ok
```

C.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the MS-BASIC command to CP/M-86. The format of the command line is (the command line may appear differently on your screen than on this page):

A>MSBASIC[<*filename*>] [/F:<*number of files*>]
[/M:<*highest memory location*>]
[/S:<*maximum record size*>]

If *<filename>* is present, MS-BASIC proceeds as if a RUN *<filename>* command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows MS-BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M-86. Such programs should include a SYSTEM statement (see below) to return to CP/M-86 when they have finished, allowing the next program in the batch stream to execute.

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

The /M:*<highest memory location>* option sets the highest memory location that will be used by MS-BASIC. In some cases it is desirable to set the amount of memory well below the CP/M-86's FDOS to

reserve space for assembly language subroutines. In all cases, *<highest memory location>* should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used.

/S:*<maximum record size>* may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

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

Examples:

A>MSBASIC PAYROLL.BAS — Use all memory and 3 files, load and execute PAYROLL.BAS.

A>MSBASIC INVENT/F:6 — Use all memory and 6 files, load and execute INVENT.BAS.

A>MSBASIC /M:32768 — Use first 32K of memory and 3 files.

A>MSBASIC DATACK/F:2/M:&H9000
Use first 36K of memory, 2 files, and execute DATACK.BAS.

C.2 DISK FILES

Disk filenames follow the normal CP/M-86 naming conventions. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the currently selected drive is assumed. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN *<filename>* commands if no "." appears in the filename and the filename is less than 9 characters long.

Large random files are supported. The maximum logical record number is 32767. If a record size of 256 is specified, then files up to 8 megabytes can be accessed.

C.3 FILES COMMAND

Format:

FILES[*<filename>*]

Purpose:

Prints the names of files residing on the current disk.

Remarks:

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

Examples:

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

C.4 RESET COMMAND

Format:

RESET

Purpose:

Closes all disk files and writes the directory information to a diskette before it is removed from a disk drive.

Remarks:

Always execute a RESET command before removing a diskette from a disk drive. Otherwise, when the diskette is used again, 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.

C.5 LOF FUNCTION

Format:

LOF(<*file number*>)

Action:

Returns 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) THEN PRINT "INVALID ENTRY"
```

C.6 EOF

With CP/M-86, the EOF function may be used with random files. If a GET is done past the end of file, EOF will return -1. This may be used to find the size of a file using a binary search or other algorithm.

C.7 MISCELLANEOUS

1. CSAVE and CLOAD are not implemented.
2. To return to CP/M-86, use the SYSTEM command or statement. SYSTEM closes all files and then performs a CP/M-86 warm start. Alt-C always returns to MS-BASIC, not to CP/M-86.
3. FRCINT is at 103 hex and MAKINT is at 107 hex.

APPENDIX D: Using MS-BASIC with the MS-DOS Operating System

The MS-DOS version of MS-BASIC is supplied on a standard 5¼ inch diskette. The name of the file is MSBASIC.COM. (A 48K or larger MS-DOS system is recommended.)

To run MS-BASIC, bring up MS-DOS and type:

```
A>MSBASIC<RET>
```

The system will reply:

```
        MS-BASIC Rev. 5.xx
         [86-DOS Version]
Copyright 1977-1981 (C) by Microsoft
       Created: dd-mmm-yy
         xxxxx Bytes free
               Ok
```

D.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the MS-BASIC command to MSDOS. The format of the command line is (the command line may appear somewhat differently on your screen than on this page):

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

If *<filename>* is present, MS-BASIC proceeds as if a RUN *<filename>* command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows MS-BASIC programs to be executed in batch mode using the SUBMIT facility of MS-DOS. Such programs should include a SYSTEM statement (see below) to return to MS-DOS when they have finished. This allows the next program in the batch stream to execute.

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

The /M:*<highest memory location>* option sets the highest memory location that will be used by MS-BASIC.

?S:<*maximum record size*> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

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

Examples:

A>MSBASIC PAYROLL.BAS	Use all memory and 3 files, load and execute PAYROLL.BAS.
A>MSBASIC INVENT/F6	Use all memory and 6 files, load and execute INVENT.BAS.
A>MSBASIC /M:32768	Use first 32K of memory and 3 files.

A>MSBASIC DATACK/F:2/M:H9000

Use first 36K of memory, 2 files, and execute DATACK.BAS.

D.2 DISK FILES

Disk file names follow the normal MS-DOS naming conventions. All file names may include A: or B: as the first two characters (to specify a disk drive); otherwise the currently selected drive is assumed. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the file name and the file name is less than 9 characters long.

Large random files are supported. The maximum logical record number is 32767. If a record size of 256 is specified, then files up to 8 megabytes can be accessed.

D.3 FILES COMMAND

Format:

FILES[<*filename*>]

Purpose:

Prints the names of files residing on the current disk.

Remarks:

If <*filename*> is omitted, all the files on the currently selected drive will be listed. <*filename*> is a string formula which may contain question marks (?) to match any character in the file name or extension. Asterisk (*) as the first character of the file name or extension will match any file or any extension.

Examples:

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

D.4 RESET COMMAND

Format:

RESET

Purpose:

Closes all disk files and writes the directory information to a diskette before it is removed from a disk drive.

Remarks:

Always execute a RESET command before removing a diskette from a disk drive. If you forget, the diskette will not have the current directory information written on the directory track when it is used again.

D.5 LOF FUNCTION

Format:

LOF(<*file number*>)

Purpose:

Returns the length of the file in bytes.

Example:

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

D.6 EOF

Format:

EOF (<*file number*>)

Purpose:

Returns -1 if the end of a sequential or random file has been reached. If a GET is done past the end of the file, EOF will return -1. EOF may be used to find the size of a file using a binary search or other algorithm.

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
```

D.7 MISCELLANEOUS

1. CSAVE and CLOAD are not implemented.

2. Use the SYSTEM command or statement to return to MS-DOS. SYSTEM closes all files and then performs a MS-DOS warm start. Alt-C always returns to MS-BASIC, not to MS-DOS.

3. FRCINT is at 103 hex and MAKINT is at 107 hex.

MS-BASIC has provisions for interfacing with assembly language subroutines via the USR function and the CALL statement.

The USR function allows assembly language subroutines to be called in the same way MS-BASIC Intrinsic functions are called.

E.1 MEMORY ALLOCATION

IMPORTANT: Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s) with the /M: switch.

In addition to the MS-BASIC interpreter code area, MS-BASIC uses up to 64K of memory beginning at its data (DS) segment.

If more stack space is needed when an assembly language subroutine is called, the MS-BASIC stack can be saved and a new stack set up for use by the assembly language subroutine. The stack must be restored, however, before returning from the subroutine.

You can load the assembly language subroutine into memory by using the operating system or the MS-BASIC POKE statement. If you have the *Programmer's Tool Kit, Volume II*, the routines can be assembled with the MACRO-86 assembler and linked using the MS-LINK linker, but not assembled. To load the program file, observe these guidelines:

1) The subroutines must not contain any long references.

2) Skip over the first 512 bytes of the MS-LINK output file, then read in the rest of the file.

E.2 USING THE CALL STATEMENT

The CALL statement is the recommended way of interfacing 8086 machine language programs with MS-BASIC. It is further suggested that the old style user call (x=USR(n)) not be used.

Format:

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

where: variable name contains the address that is the starting point in memory of the subroutine being CALLed.

argument list contains the variables or constants, separated by commas, that are to be passed to the routine.

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. MS-BASIC follows the rules described for the MEDIUM case (summarized in the following discussion).

Invoking the CALL statement causes the following to occur:

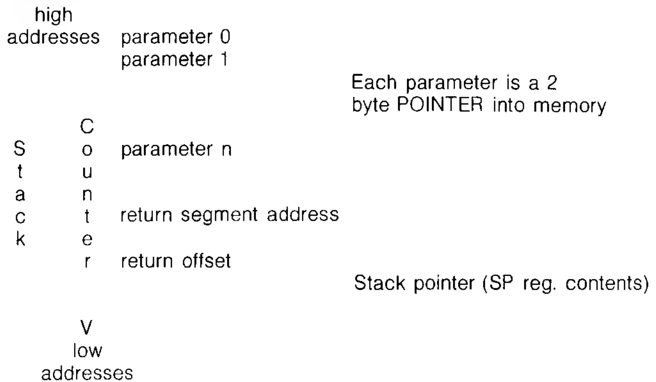
1. For each parameter in the argument list, the 2 byte offset of the parameter's location within the Data Segment [DS] is pushed onto the stack.

2. MS-BASIC return address Code segment [CS], and offset [IP] are pushed onto the Stack.

3. Control is transferred to the user's routine via an 8086 long call to the segment address given in the last DEF SEG statement, and offset given in *<variable name>*.

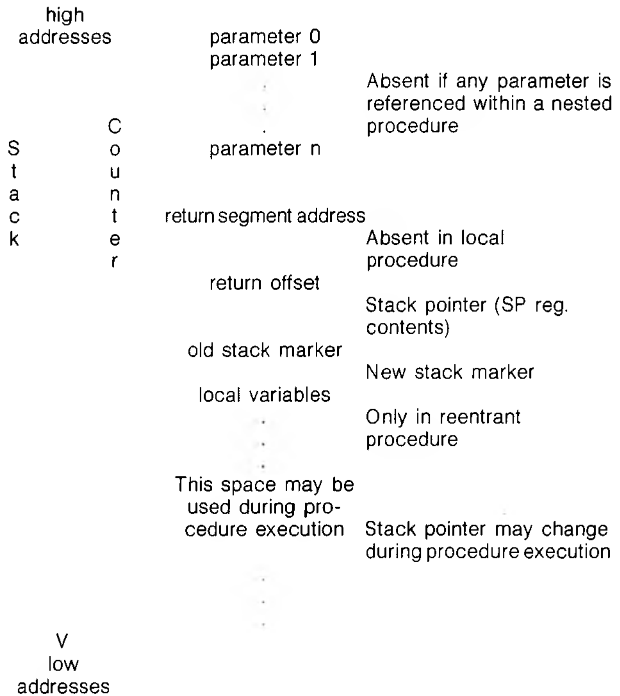
These actions are illustrated by the two following diagrams, which illustrate, first, the state of the stack at the time of the CALL statement, and, second, the condition of the stack during execution of the CALLed subroutine.

Figure E-1: Stack Layout when CALL Statement is Activated



The user's routine now has control. Parameters may be referenced by moving the Stack pointer [SP] to the Base Pointer [BP] and adding a positive offset to [BP].

Figure E-2: Stack Layout During Execution of a CALL Statement



You must observe the following rules when coding a subroutine:

1. The CALLed routine may destroy the AX, BX, CX, DX, SI, DI, and BP registers.

2. The CALLed program MUST know the number and length of the parameters passed. References to parameters are positive offsets added to [BP] (assuming the called routine moved the current stack pointer into BP; i.e., MOV BP,SP). That is, the location of p1 is at 8[BP], p2 is at 6[BP], p3 is at 4[BP], . . . etc.

3. The CALLed routine must do a RET <n> (where <n> is two times the number of parameters in the argument list) to adjust the stack to the start of the calling sequence.

4. Values are returned to MS-BASIC by including in the argument list the variable name which will receive the result.

5. If the argument is a string, the parameter's offset points to 3 bytes called the "String Descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

IMPORTANT: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
20 A$ = "BASIC"+" "
```

This will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

6. Strings may be altered by user routines, but the length MUST NOT be changed. MS-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Example:

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

Line 100 sets the segment to 8000 Hex. The value of FOO is added into the address as the low word after the DEF SEG value is left shifted 8 bits. Here, FOO is set to zero, so that the call to FOO will execute the subroutine at location 80000H.

The following sequence of 8086 assembly language demonstrates access of the parameters passed and storing a return result in the variable 'C'.

Example:

```
MOV    BP,SP          ; Get current Stack posn in BP.
MOV    BX,6[BP]       ; Get address of B$ dope.
MOV    CL,[BX]        ; Get length of B$ in CL.
MOV    DX,1[BX]       ; Get address of B$ text in DX.
       .
       .
       .
MOV    SI,8[BP]       ; Get address of 'A' in SI.
MOV    DI,4[BP]       ; Get pointer to 'C' in DI.
MOVS   WORD           ; Store variable 'A' in 'C'.
RET    6              ; Restore Stack, return.
```

IMPORTANT: The called program must know the variable type for numeric parameters passed. In the above example, the instruction MOVS WORD will copy only 2 bytes. This is fine if variables A and C are integer. We would have to copy 4 bytes if they were Single Precision and copy 8 bytes if they were Double Precision.

E.3 USING USR FUNCTION CALLS

The format of the USR function call is:

x = USR[<*digit*>](argument)

where: <*digit*> is from 0 to 9. <*digit*> specifies which USR routine is being called. (See "The DEF USR Statement"). If <*digit*> is omitted, USR0 is assumed.

argument is any numeric or string expression.

x is the variable receiving the result of the function call. Its type (numeric or string) must be consistent with the argument passed, or may be set to Integer by calling MAKINT in the user's routine before returning to MS-BASIC.

A DEF SEG statement MUST be executed prior to a USR call to assure that the Code Segment points to the subroutine being called. The segment given in the DEF SEG statement determines the starting segment of the subroutine. (See the DEF SEG statement above.)

For each USR function used, a corresponding DEF USR statement must have been executed to define the USR call offset. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register [AL] contains a value that specifies the type of argument that was given. The value in [AL] may be one of the following:

2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, the [BX] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

If the argument is an integer:

FAC-2 contains the upper 8 bits of the argument.

FAC-3 contains the lower 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-2 contains the middle 8 bits of mantissa.

FAC-3 contains the lowest 8 bits of mantissa.

If the argument is a double precision floating point number:

FAC-7 To FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the [DX] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in MS-BASIC's Data Segment.

IMPORTANT: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. See the CALL statement above.

Usually, the value returned by a USR function is the same type (integer, string, single precision or double precision) as the argument that was passed to it.

Example:

```
110 DEF USR0=&H8000 'Assumes user gave /M:32767
120 X=5 'Note that X is single precision
130 Y = USR0(X)
140 PRINT Y
```

We have loaded the following assembly language routine to simply multiply the argument passed by 2 and return an integer result.

Always be sure that your programs are defined by a PROC FAR statement.

Example:

```
DOBLE         SEGMENT
              ASSUME    CS:DOUBLE
FRCINTOFFSET  EQU       103H
MAKINTOFFSET  EQU       107H

FRCINT        LABEL     DWORD
              DW        FRCINTOFFSET
FRCSEG        DW        ?

MAKINT        LABEL     DWORD
              DW        MAKINTOFFSET
MAKSEG        DW        ?
```

```
USRPRG      PROC    FAR
            POP     SI
            POP     AX              ; Recover MS-BASIC CS
            PUSH    AX
            PUSH    SI
            MOV     FRCSEG,AX       ; Set segment for long indirect CALL
            MOV     MAKSEG,AX
            CALL    FRCINT          ; Force arg in FAC to int in [BX]
            ADD     BX,BX           ;  [BX] = [BX] * 2
            CALL    MAKINT          ; Put Result back in FAC
            RET                     ; Long return to MS-BASIC
USRPRG      ENDP

DOUBLE      ENDS
```

When FRCINT or MAKINT is called and when the subroutine terminates with a return, ES, DS, and SS must have the same value they had when the subroutine was entered. These registers point to the MS-BASIC Data Segment.

The MS-BASIC Compiler package contains the following software: MS-BASIC Compiler, MACRO assembler, and LINK loader. These manuals are also supplied: *MS-BASIC Reference Manual, MS-BASIC Compiler Programmer's Guide,* and *Programmer's Tool Kit, Volume II.* The Programmer's Tool Kit describes the use of the MACRO-86 macroassembler and the MS-LINK linker utility. The *MS-BASIC Compiler Programmer's Guide* describes the use of the compiler, its command format, compilation switches, and error messages. The MS-BASIC language that is used with the MS-BASIC Compiler is the same as the MS-BASIC described in this manual, with the following exceptions.

F.1 OPERATIONAL DIFFERENCES

The Compiler interacts with the console only to read compiler commands. These specify what files are to be compiled. There is no "direct mode," as with the MS-BASIC interpreter. Commands that are usually issued in the direct mode with the MS-BASIC interpreter are not implemented on the compiler.

The following statements and commands are not implemented and will generate an error message:

AUTO	CONT	DELETE
EDIT	ERASE	LIST
LOAD	MERGE	NEW
RENUM	LLIST	SAVE

Because there is no direct mode for typing in programs or edit mode for editing programs, use the MS-BASIC interpreter for creating and editing programs. If you use the interpreter, be sure to SAVE the file with the A (ASCII format) option.

The compiler cannot accept a physical line that is more than 253 characters in length. A logical statement, however, may contain as many physical lines as desired. Use line feed to start a new physical line within a logical statement.

To reduce the size of the compiled program, there are no program line numbers included in the object code generated by the compiler unless the /D, /X, or /E switch is set in the compiler command. As a result, error messages contain the address where the error occurred, instead of a line number. The compiler listing and the map generated by LINK are used to identify the line that has the error. It is always a good idea to debug programs using the MS-BASIC Interpreter before attempting to compile them. See the *MS-BASIC Compiler Programmer's Guide* for more information.

F.2 LANGUAGE DIFFERENCES

Most programs that run on the MS-BASIC interpreter will run on the MS-BASIC compiler with little or no change. However, it is necessary to note differences in the use of the following program statements:

1. CALL

   The <*variable-name*> field in the CALL statement must contain an external symbol, i.e., one that is recognized by LINK as a global symbol. This routine must be supplied by the user as an assembly language subroutine or a routine from the FORTRAN library.

2. CHAIN and RUN

   The CHAIN statement is used to chain to a new program overlay using the runtime module. The RUN statement is to be used to execute any executable file. (Refer to Appendix C for CP/M and Appendix D for MS-DOS).

3. CLEAR

   The CLEAR statement is only supported in compiled programs using the runtime module.

4. COMMON

   The COMMON statement must appear before any executable statements. See section 2.7 for further details.

5. DEFINT/SNG/DBL/STR

   The compiler does not "execute" DEFxxx statements; it reacts to the static occurrence of these statements, regardless of the order in which program lines are executed. A DEFxxx statement takes effect as soon as its line is encountered. Once the type has been defined for a given variable, it remains in effect until the end of the program or until a different DEFxxx statement with that variable takes effect.

6. DIM and ERASE

   The DIM statement is similar to the DEFxxx statement in that it is scanned rather than executed. That is, DIM takes effect when its line is encountered. If the default dimension (10) has already been established for an array variable and that variable is later encountered in a DIM statement, a "Redimensioned array" error results.

There is no ERASE statement in the compiler, so arrays cannot be erased and redimensioned. An ERASE statement will produce a fatal error.

Also note that the values of the subscripts in a DIM statement must be integer constants; they may not be variables, arithmetic expressions, or floating point values.

Example:

```
DIM A1 (I)
DIM A1 (3-4)
```

are both illegal.

7. END

During execution of a compiled program, an END statement closes files and returns control to the operating system. The compiler assumes an END statement at the end of the program, so "running off the end" produces proper program termination.

8. FOR/NEXT and WHILE/WEND

Loops must be statically nested when using these statements.

9. ON ERROR GOTO/RESUME *<line number>*

If a program contains ON ERROR GOTO and RESUME *<line number>* statements, the /E compilation switch must be used. If the RESUME NEXT, RESUME, or RESUME 0 form is used, the /X switch must also be included. See the *MS-BASIC Compiler Programmer's Guide* for an explanation of these switches.

10. REM

REM statements or remarks starting with a single quotation mark do not take up time or space during execution, and so may be used as freely as desired.

11. STOP

The STOP statement is identical to the END statement. Open files are closed and control returns to the operating system.

12. TRON/TROFF

In order to use TRON/TROFF, the /D compilation switch must be used. Otherwise, TRON and TROFF are ignored and a warning message is generated.

13. USRn Functions

USRn Functions are significantly different from the interpreter versions. The argument to the USRn function is ignored and an integer result is returned in the HL registers. It is recommended that USRn functions be replaced by the CALL statement.

14. %INCLUDE

The %INCLUDE *<filename>* statement allows the compiler to include source from an alternate file. The %INCLUDE statement must be the last statement statement on a line. The format of the

%INCLUDE statement is:

<line number> %INCLUDE *<filename>*

Example:

```
999        %INCLUDE SUB1000.BAS
```

15. Double-Precision Transcendental Functions

    SIN, COS, TAN, SQR, LOG, and EXP return double precision results if given a double precision argument. Exponentiation with double precision operands will return a double precision result.

16. String Variables

    The string space is maintained differently with the MS-BASIC Compiler than with the interpreter. Using PEEK, POKE, VARPTR, or assembly language routines to change string descriptors will result in a "String Space Corrupt" error.

F.3 EXPRESSION EVALUATION

During expression evaluation, the operands of each operator are converted to the same type, that of the most precise operand.

Example:

```
QR=J%+A !+Q#
```

causes J% to be converted to single precision and added to A !. this result is converted to double precision and added to Q#.

The compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter the following program:

Example:

```
I%=20000
J%=20000
K%=30000
M%=I%+J%-K%
```

yields 10000 for M%. That is, it adds I% to J% and, because the number is too large, it converts the result into a floating point number. K% is then converted to floating point and subtracted. The result of 10000 is found, and is converted back to integer and saved as M%.

The compiler, however, must make type conversion decisions during compilation. It cannot defer until the actual values are known. Thus, the compiler would generate code to perform the entire operation in integer mode. If the /D switch were set, the error would be detected. Otherwise, an incorrect answer would be produced.

In order to produce optimum efficiency in the compiled program, the compiler may perform any number of valid algebraic transformations before generating the code. For example, the program:

Example:

```
I%=20000
J%=18000
K%=20000
M%=I%+J%+K%
```

could produce an incorrect result when run. If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds J%, an overflow will occur. The compiler follows the rules for operator precedence and parenthetic modification of such precedence, but no other guarantee of evaluation order can be made.

F.4 INTEGER VARIABLES

In order to produce the fastest and most compact object code possible, make maximum use of integer variables. For example, this program:

Example:

```
FOR I=1 TO 10
A (I)=0
NEXT I
```

can execute approximately 30 times faster by simply substituting "I%" for "I". It is especially advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

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 parentheses, misspelled command or statement, incorrect punctuation, etc.).
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.
FC	5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 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 MS-BASIC 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.
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 MS-BASIC to exceed the amount of free memory remaining. MS-BASIC will allocate 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.

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.
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.

26 FOR without NEXT
A FOR was encountered without a matching NEXT.

29 WHILE without WEND
A WHILE statement does not have a matching WEND.

30 WEND without WHILE
A WEND was encountered without a matching WHILE.

DISK ERRORS

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 MS-BASIC. Report the conditions under which the message appeared to your dealer.

52 Bad file number
A statement or 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 disk.

54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.

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 disk I/O operation. It is a fatal error, i.e., 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 disk storage space is in use.

62 Input past end
An INPUT statement is exeucted 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 (e.g., a filename with too many characters).

66 Direct statement in file
A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files
An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

Functions that are not intrinsic to MS-BASIC may be calculated as follows.

FUNCTION	MS-BASIC 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
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 COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1) +1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

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

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

READER'S COMMENTS FORM

Your comments are a main source of ideas for improvement. Please use this form to provide us with feedback on this document.

DOCUMENT

TITLE: ____________________

PART NUMBER: ____________________

YOUR GENERAL REACTION:

Overall quality:	☐ Excellent	☐ Adequate	☐ Poor
Text clarity:	☐ Very clear	☐ Adequate	☐ Difficult
Usefulness of format:	☐ Helpful	☐ Adequate	☐ Inconvenient

YOUR SPECIFIC COMMENTS:

Did you find any errors in the document? ____________________

If so, describe: ____________________

Was any important information omitted from the document? __________

If so, describe: ____________________

What sections of the document were especially useful to you?

What sections were of no use to you? ____________________

How could material be presented to be more helpful to you?

READER'S NAME: ____________________
JOB TITLE: ____________________
COMPANY: ____________________
ADDRESS: ____________________

Please complete and return this form to the office, subsidiary or distributor nearest you.

OFFICES, SUBSIDIARIES, AND DISTRIBUTORS

FRANCE
Victor Technologies, Inc. S.A.R.L.
28, rue Jean Jaures
92800 Puteaux
Phone: 33 (1)-773-8564
Telex: 614764

ITALY
Harden S.p.A. Divisione Elettronica
Via Guiseppina 110
26048 Sospiro (Cremona)
Phone: (372) 63136
Telex: 320588

UNITED KINGDOM
ACT (Microsystems) Ltd.
Shenstone House
Dudley Road
Halesowen
West Midlands B63 3NT
Phone: 021-501-2284
Telex: 339396

UNITED STATES
Victor Technologies, Inc.
380 El Pueblo Road
Scotts Valley, CA 95066
Phone: 1 (408) 438-6680
Telex: 357403

WEST GERMANY
Sirius Computer GmbH
Orber Strasse 24
6000 Frankfurt 61
Phone: 0 611-410223
Telex: 4185558