Language specification for MSX BASIC

Ver 1.3 (19th August '83)

CHAPTER 1

GENERAL INFORMATION ABOUT MSX BASIC

MSX BASIC is an extended version to the Microsoft standard Basic version 4.5, which includes supports to graphics, music and various peripherals attached to MSX Home and Personal computer. Generally, MSX BASIC is designed to follow the GW-BASIC which is a standard Basic in 16-bit machine world. But the major effort was made to make the whole system as flexible and expandable as possible.

Also MSX BASIC is featured with up to 14 digits accuracy double precision BCD arithmetic function. This means arithmetic operations no more generate strange round errors that confuse novice users. Every trancendental functions are also calculated with this accuracy. 16 bit signed integer operation is also available for faster execution.

1.1 MODES OF OPERATION

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

In direct mode, MSX 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. Direct mode is useful for debugging and for using MSX BASIC as a "calculator" for quick computations that do not require a complete program.

Indirect mode is 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.

1.2 LINE FORMAT

MSX BASIC program lines have the following format (square brackets indicate optional input):

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

More than one BASIC statement may be placed on a line, but each must be separated from the last by a colon.

An MSX BASIC program line always begins with a line number and ends with a carriage return. A line may contain a maximum of 255 characters.

1.2.1 Line Numbers

Every MSX BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory. Line numbers are also used as references in branching and editing. Line numbers must be in the range 0 to 65529.

A period (.) may be used in LIST, AUTO. and DELETE commands to refer to the current line.

1.3 CHARACTER SET

The MSX BASIC character set consists of alphabetic characters, numeric characters, special characters ,graphic characters and both hirakana and katakana characters.

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

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

In addition, the following special characters are recognized by MSX BASIC:

Character	Action
	Blank
=	Equals sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket

,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
¥	Yen sign or integer division symbol
@	At sign
_	Underscore
<rubout>	Deletes last character typed.
<escape>	Escapes
<tab>	Moves print position to next tab stop. Tab stops are set every eight columns.
<line feed>	Moves to next physical line.
<carriage return>	Terminates input of a line.

1.4 CONSTANTS

Constants are the values MSX 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:

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

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

1.	Integer constants	Whole numbers between -32768 and 32767. Integer constants do not contain 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 10-64 to 10+63.
		Examples:

235.988E-7 = .0000235988
2359E6 =2359000000

(Double precision floating-point constants are denoted by the letter D instead of E.)

4. Hex constants — Hexadecimal numbers, denoted by the prefix &H.

Examples:

&H76
&H32F

5. Octal constants — Octal numbers, denoted by the prefix &O or &.

Examples:

&0347
&1234

6. Binary constants — Binary numbers, denoted by the prefix &B.

Examples:

&B01110110
&B11100111

1.4.1 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 6 digits of precision, and printed with up to 6 digits of precision. Double precision numeric constants are stored with 14 digits of precision and printed with up to 14 digits. Double precision is the default for constant in MSX BASIC.

A single precision constant is any numeric constant that has one of the following characteristics:

1. Exponential form using E.
2. A trailing exclamation point (!).
   Examples:

   -1.09E-06
   22.5!

A double precision constant is any numeric constant that has one of these characteristics:

1. Any digits of number without any exponential or type specifier.

2. Exponential form using D.

3. A trailing number sign (#).

Examples:

```
 3489
 345692811
-1.09432D-06
 3489.0#
 7654321.1234
```

1.5 VARIABLES

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

1.5.1 Variable Names And Declaration Characters

MSX BASIC variable names may be any length. Up to 2 characters are significant. Variable names can contain letters and numbers. However, 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 and may not contain embedded reserved words. Reserved words include all MSX BASIC commands, statements, function names, and operator names. If a variable begins with FN, it is assumed to be a call to a user-defined function.

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 precision, 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 double precision.

Examples of MSX BASIC variable names:

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 double precision value.

There is a second method by which variable types may be declared. The MSX BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL may be included in a program to declare the types for certain variable names. Refer to the description for these statements.

1.5.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements is determined by memory size.

1.5.3 Space Requirements

The following table lists only the number of bytes occupied by the values represented by the variable names.

Variables	Type	Bytes
	Integer	2
	Single Precision	4
	Double Precision	8

Arrays	Type	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.

1.6 TYPE CONVERSION

When necessary, MSX 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
 .85714285714286
```

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
 .857143
```

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 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 truncated.

Example:

```
10 C%=55.88
20 PRINT C%
RUN
 55
```

5. If a double precision variable is assigned a single precision value, only the first six digits of the converted number will be valid. This is because only six digits of accuracy were supplied with the single precision value.

Example:

```
10 A!=SQR(2)
20 B=A!
30 PRINT;A!,B
```

```
RUN
 1.41421         1.41421
```

1.7 EXPRESSIONS AND OPERATORS

An expression may be a string or numeric constant, a variable, or a combination of constants and variables with operators which produces a single value.

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

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

Each category is described in the following sections.

1.7.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating-point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

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

1.7.1.1 Integer Division And Modulus Arithmetic

Two additional operators are available in MSX BASIC:

Integer division is denoted by the yen symbol. The operands are truncated 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=4
```

Integer division follows multiplication and floating-point division in order of precedence.

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

Example:

```
10.4 MOD 4=2 (10/4=2 with a remainder 2)
25.68 MOD 6.99=1 (25/6=4 with a remainder 1)
```

Modulus arithmetic follows integer division in order of precedence.

1.7.1.2 Overflow And Division By Zero -

If, during the evaluation of an expression, division by zero is encountered, the "Division by zero" error message is displayed and execution of program terminates.

If overflow occurs, the "Overflow" error message is displayed and execution terminates.

1.7.2 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 description for "IF" statements.)

The relational operators are:

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

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

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

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

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

More examples:

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

1.7.3 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 Table 1. The operators are listed in order of precedence.

Table 1. MSX BASIC Relational Operators Truth Table

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 .

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.

Thus, 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 to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63=binary 111111 and 16=binary 10000, so 63 AND 16=16.
15 AND 14=14	15=binary 1111 and 14=binary 1110, so 15 AND 14=14 (binary 1110).
-1 AND 8=8	-1=binary 1111111111111111 and 8=binary 1000, so -1 AND 8=8.
4 OR 2=6	4=binary 100 and 2=binary 10, so 4 OR 2=6 (binary 110).
10 OR 10=10	10=binary 1010, so 1010 OR 1010= 1010 (decimal 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.

1.7.4 Functional Operators

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

MSX BASIC also allows "user-defined" functions that are written by the programmer. See descriptions for "DEF FN".

1.7.5 String Operations

Strings may be concatenated by 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$<"9/12/83"      where B$="8/12/83"
```

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.

1.8 PROGRAM EDITING

The Full Screen Editor equiped with MSX BASIC allows the user to enter program lines as usual, then edit an entire screen before recording the changes. This time-saving capability is made possible by special keys for cursor movement, character insertion and deletion, and line

or screen erasure. Specific functions and key assignments are discussed in the following sections.

With the Full Screen Editor, a user can move quickly around the screen, making corrections where necessary. The changes are entered by placing the cursor on the first line changed and pressing <RETURN> at the beginning of each line. A program line is not actually changed until <RETURN> is entered from somewhere within the line.

Writing Programs

Within MSX BASIC, the editor is in control any time after an OK prompt and before a RUN command is issued. Any line of text that is entered is processed by the editor. Any line of text that begins with a number is considered a program statement.

Program statements are processed by the editor in one of the following ways:

1. A new line is added to the program. This occurs if the line number is valid (0 through 65529) and at least one non-blank character follows the line number.

2. An existing line is modified. This occurs if the line number matches that of an existing line in the program. The existing line is replaced with the text of the new line.

3. An existing line is deleted. This occurs if the line number matches that of an existing line, and the new line contains only the line number.

4. An error is produced.

   If an attempt is made to delete a non-existent line, an "Undefined line number" error message is displayed.

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

More than one statement may be placed on a line. If this is done, the statements must be separated by a colon (:). The colon does not have to be surrounded by spaces.

The maximum number of characters allowed in a program line, including the line number, is 250.

Editing Programs

Use the LIST statement to display an entire program or range of lines on the screen so that they can be edited. Text can then be modified by moving the cursor to the place where the change is needed and performing one of the following actions:

1. Typing over existing characters
2. Deleting characters to the right of the cursor
3. Deleting characters to the left of the cursor
4. Inserting characters
5. Appending characters to the end of the logical line

These actions are performed by special keys assigned to the various Full Screen Editor functions (see next section).

Changes to a line are recorded when a carriage return is entered while the cursor is somewhere on the line. The carriage return enters all changes for that logical line, no matter how many physical lines are included and no matter where the cursor is located on the line.

Full Screen Editor Functions

The following table lists the hexadecimal codes for the MSX BASIC control characters and summarizes their functions. The Control-key sequence normally assigned to each function is also listed. These conform as closely as possible to ASCII standard conversions.

Individual control functions are described following the table.

Table 1. MSX BASIC Control Functions. The ASCII control key is entered by pressing the key while holding down the Control key.

Hex. Code	Control Key	Special Key	Function
01	A		Graphic character header
02	B		Move cursor to start of previous word
03	C		Break when MSX BASIC is waiting for input
04	D		Ignored
05	E		Truncate line (clear text to end of logical line)
06	F		Move cursor to start of next word
07	G		Beep
08	H	Back Space	Backspace, deleting characters passed over
09	I	Tab	Tab (8 spaces)
0A	J		Line feed
0B	K	Home	Move cursor to home position
0C	L	CLS	Clear screen
0D	M	Return	Carriage return (enter current logical line)
0E	N		Append to end of line
0F	O		Ignored
10	P		Ignored
11	Q		Ignored
12	R	INS	Toggle Insert/typeover mode
13	S		Ignored

14	T		Ignored
15	U		Clear logical line
16	V		Ignored
17	W		Ignored
18	X	Select	Ignored
19	Y		Ignored
1A	Z		Ignored
1B	[	ESC	Ignored
1C	\	Right arrow	Cursor right
1D	]	Left arrow	Cursor left
1E	^	Up arrow	Cursor up
1F	_	Down arrow	Cursor down
7F	DEL	DEL	Delete character at cursor

PREVIOUS WORD

The cursor is moved left to the previous word. The previous word is defined as the next character to the left of the cursor in the sets A-Z, a-z, or 0-9.

BREAK

Returns to MSX BASIC direct mode, without saving changes that were made to the line currently being edited.

TRUNCATE

The cursor is moved to the end of the logical line. The characters it passes over are deleted. Characters typed from the new cursor position are appended to the line.

NEXT WORD

The cursor is moved right to the next word. The next word is defined as the next character to the right of the cursor in the sets A-Z, a-z, or 0-9.

BEEP

The beep sound will be produced.

BACKSPACE

Deletes the character to the left of the cursor. All characters to the right of the cursor are moved left one position. Subsequent characters and lines within the current logical line are moved up (wrapped).

TAB

TAB moves the cursor to the next tab stop overwriting blanks. Tab stops occur every 8 characters.

CURSOR HOME

Moves the cursor to the upper left corner of the screen. The screen is not blanked.

CLEAR SCREEN

Moves the cursor to home position and clears the entire screen, regardless of where the cursor is positioned when the key is entered.

CARRIAGE RETURN

A carriage return ends the logical line and sends it to MSX BASIC.

APPEND

Moves cursor to the end of the line, without deleting the characters passed over. All characters typed from the new position until a carriage return are appended to the logical line.

INSERT

Toggle switch for insert mode. When insert mode is on, the size of the cursor is reduced and characters are inserted at the current cursor position. Characters to the right of the cursor move right as new ones are inserted. Line wrap is observed.
When insert mode is off, the size of cursor returned to normal size and typed characters will replace existing characters on the line.

CLEAR LOGICAL LINE

When this key is entered anywhere in the line. the entire logical line is erased.

CURSOR RIGHT

Moves the cursor one position to the right. Line wrap is observed.

CURSOR LEFT

Move the cursor one position to the left. Line wrap is observed.

CURSOR UP

Moves the cursor up one physical line (at the current position).

CURSOR DOWN

Move the cursor down one physical line (at the current position).

Logical line Definition with INPUT

Normally, a logical line consists of all the characters on each of the physical lines that make up the logical line. During execution of an INPUT or LINE INPUT statement, however, this definition is modified slightly to allow for forms input. When either of these statements is executed, the logical line is restricted to characters actually typed or passed over by the cursor. Insert mode and the delete function only move characters which are within that logical line, and Delete will decrement the size of the line.

Insert mode increments the logical line except when the characters moved will write over non-blank characters that are on the same physical line but not part of the logical line. In this case, the non-blank characters not part of the logical line are preserved and the characters

at the end of the logical line are thrown out. This preserves labels that existed prior to the INPUT statement. If an incorrect character is entered as a line is being typed, it can be deleted with the <Back Space> key or with Control-H. and they backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

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

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

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

1.9 Special keys

MSX BASIC supports several special keys as follows.

1.9.1 Function Keys

MSX BASIC has 10 pre-defined function keys. The current contents of these keys are displayed on the last line on the screen and can be re-defined by program with KEY statement. The initial values for each keys are:

F1	color[b]	[b] = blank character
F2	auto[b]	[cr]= carriage return
F4	goto[b]	[u] = cursor up character
F5	list[b]	[cls]=clear screen character
F5	run[cr]	
F6	color 15,4,7[cr]	
F7	cload"	
F8	cont[cr]	
F9	list.[cr][u][u]	
F10	[cls]run[cr]	

Function keys are also used as event trap keys. See ON KEY GOSUB and KEY ON/OFF/STOP statement for details.

1.9.2 Stop key

When MSX BASIC is in command mode, the STOP key has no effects to the operation, MSX BASIC just ignores it.

When MSX BASIC is executing the program, pressing the STOP key causes suspension of the program execution, and MSX BASIC turn on the cursor display to indicate that the execution is suspended. Another STOP key input resumes the execution. If the STOP key and control key are pressed simultaneously, MSX BASIC terminates the execution and return

to command mode with following message.

Break in nnnn

where nnnn is the program line number where the execution stopped.

1.10 ERROR MESSAGES

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

CHAPTER 2

MSX BASIC COMMANDS, STATEMENTS AND FUNCTIONS

2.1 Commands, Statements, and Functions except I/O

2.1.1 Commands except I/O

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

To generate a line number automatically after every carriage return.

AUTO begins numbering at <line number> and increment each subsequent line number by <increment>. The default for both value 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 line number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

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

CONT

To continue program execution after BREAK or STOP in execution.

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

To delete program lines.

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

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

To list all or part of the program.

If both <line number> parameters are omitted, the program is listed beginning at the lowest line number.

If only the first <line number> is specified, that line is listed.

If the first <line number> and "-" are specified, that line and all higher-numbered lines are listed.

If "-" and the second <line number> are specified, all lines from the beginning of the program through that line are listed.

If both <line number> parameters are specified, the range from the first <line number> through the second <line number> is listed.

Listing is terminated by typing "CTRL" and "STOP" keys at same time. Listing is suspended by typing "STOP" key. and it is resumed by typing "STOP" key again.

LLIST [<line number>[-[<line number>]]]
To list all or part of the program on the printer. (See the LIST command for details of the parameters)

NEW
To delete entire program from working memory and reset all variables.

RENUM [[<new number>][,[<old number>][,<increment>]]]
To renumber program lines.

<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. ELSE. ON..GOTO, ON..GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statement, the error message 'Undefined line nnnn in mmmm' is printed. The incorrect line number reference(nnnn) is not changed by RENUM, but line number mmmm 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.

RUN [<line number>]
To execute a program.

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

TRON/TROFF

To trace the execution of program statements.

As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as 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).

CLEAR [<string space>[,<highest location>]]

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

<string space>
Space for string variables. Default size is 200 bytes.
<Highest location>
The highest memory location available for use by BASIC.

DATA <list of constants>

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

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 there in 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 comma, 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 read from the beginning or specified line by use of the RESTORE statement.

DIM <list of subscripted variables>

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

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.

DEFINT <range(s) of letters>
DEFSNG <range(s) of letters>
DEFDBL <range(s) of letters>
DEFSTR <range(s) of letters>

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

DEFINT/SNG/DBL/STR statements declare 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 DEFxxx statement in the typing of a variables. (See the end of section 1.5.1, for details of declaration characters.)

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

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

<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 name 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.

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 DEFFN 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. DEFFN is illegal in the direct mode.

DEFUSR[<digit>]=<integer expression>

To specify the starting address of an assembly language subroutine.

<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, DEFUSR0 is assumed. The value of <integer expression> is the starting address of the USR routine

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

ERASE <list of array variables>
To eliminate arrays from a program

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 '"Redimensioned array' error occurs.

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

END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional.

ERROR <integer expression>
To simulate the occurrence of an error or to allow error codes to be defined by the user.

The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC, the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed.

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

Example:

```
10 ON ERROR GOTO 1000
.
.
120 IF A$="Y" THEN ERROR 250
.
.
1000 IF ERR=250 THEN PRINT "Sure?"
.
.
```

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

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

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

note: <Variable> can be integer,single-precision or double-precision. where x,y,z are numeric expressions.

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

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

If step is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is executed one time at least if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

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

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.

GOSUB <line number>

RETURN [<line number>]

To branch to subroutine beginning at <line number> and return from a subroutine.

<line number> is the first line of the subroutine. A subroutine may be called any number of times in a program, and a subroutine

may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine. Otherwise, a 'RETURN without GOSUB' error message is issued and execution is terminated.

GOTO <line number>

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

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

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

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

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.

Example:

```
A=1:B=2  -> A=B is zero (FALSE).
A=2:b=2  -> A=B is not zero (TRUE).
```

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example,

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

will not print "A<>C" when A<>B. It will print "A<>C" when A=B and B<>C.

If an IF...THEN statement is followed by a line number in the

direct mode, an 'Undefined line' error results unless a statement with the specified line number had previously been entered in the indirect mode.

INPUT ["<prompt string>";]<list of variables>
To allow input from the keyboard during program execution.

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 keyboard.

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 names in the <list of variables> 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 the wrong type of value (string instead of the numeric, etc.) causes the message "?Redo from start" to be printed. No assignment of input value is made until an acceptable response is given.

Example:
```
list
10 INPUT "A and B";A,B
20 PRINT A+B
Ok
run
A and B? 10,@0
?Redo from start
A and B? 10,20
 30
Ok
```

Responding to INPUT with too many items causes the message "?Extra ignored" to be printed and the next statement to be executed.

Example:
```
list
10 INPUT "A and B";A,B
20 PRINT A+B
Ok
run
A and B? 10,20.30
?Extra ignored
 30
Ok
```

Responding to INPUT with too few item causes two question marks to be printed and a wait for the next data item.

Example:

```
list
10 INPUT "A and B";A,B
20 PRINT A+B
Ok
run
A and B? 10 (The 10 was typed in by the user)
?? 20       (The 20 was typed in by the user)
 30
Ok
```

Escape INPUT by typing Control-C or the "CTRL" and "STOP" keys simultaneously. BASIC returns to command level and types "Ok". Typing CONT resumes execution at the INPUT statement.

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

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

The prompt string is a string literal that is printed at the console 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>.

Escape LINE INPUT by typing Control-C or the "CTRL" and "STOP" keys simultaneously. BASIC returns to command level and types "Ok". Typing CONT resumes execution at the LINE INPUT statement.

[LET] <variable>=<expression>

; To assign value of an expression to a variable.

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

LPRINT [<list of expressions>]
LPRINT USING <string expression>;<list of expressions>

; To print data at the line printer. (see PRINT and PRINT USING statements below for details.)

MID$(<string exp. 1>),n[,m])=<string exp.2)

; To replace a portion of one string with another string.

The character in <string exp.1>, beginning at position n, are replaced by the characters in <string exp.2>. The optional m refers to the number of characters from <string exp.2> that will be used in the replacement. If m is omitted or included, the replacement of characters never goes beyond the original length of <string exp.1>.

ON ERROR GOTO <line number>

; To enable error trapping and specify the first line of the error

handling subroutine.

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

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

ON <expression> GOTO <list of line number>
ON <expression> GOSUB <list of line number>
; To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated. 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 discarded.)

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

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

OUT <port number>,<integer expression>
; To send a byte to a machine output port.

<port number> and <integer expression> are in the range 0 to 255. <integer expression> is the data (byte) to be transmitted.

POKE <address of the memory>,<integer expression>
; To write a byte into a memory location.

<address of the memory> is the address of the memory location to be POKEd. The <integer expression> is the data (byte) to be POKEd. It must be in the range 0 to 255. And <address of the memory> must be in the range -32768 to 65535. If this value is negative, address of the memory location is computed as subtracting from 65536. For example, -1 is same as the 65535 (=65536-1). Otherwise, an 'Overflow' error occurs.

PRINT [<list of expressions>]
; To output data to the console.

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 console. An expression in the list may be a numeric and/or a string expression. (Strings must be enclosed in quotation marks.)

The position of each printed item is determined by the punctuation used to separate the items in the list. 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 console width, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign.

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

PRINT USING <string expression>;<list of expressions>
; To print strings or numerics using a specified format.

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

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.

Example:
```
A$="Japan"
Ok
PRINT USING "!";A$
```

```
J
Ok
```

"&n spaces&"

Specifies that 2+n characters from the string are to be printed. If the backslashs 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:

```
A$="Japan"
Ok
PRINT USING "&  &";A$
Japa
Ok
```

"@"

Specifies that the whole character in the given string is to be printed.

Example:

```
A$="Japan"
Ok
PRINT USING "I love @ very much.";A$
I love Japan very much.
Ok
```

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.

Example:

```
PRINT USING "###.##";10.2,2,3.456,.24
 10.20  2.00  3.46  0.24
Ok
```

"+"

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.

Example:
```
PRINT USING "+###.##";1.25,-1.25
  +1.25  -1.25
Ok
PRINT USING "###.##+";1.25,-1.25
  1.25+  1.25-
Ok
```

"-"

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

Example:
```
PRINT USING "###.##-";1.25,-1.25
  1.25   1.25-
Ok
```

"**"

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 or more digits.

Example:
```
PRINT USING "**#.##";1.25,-1.25
**1.25*-1.25
Ok
```

"¥¥"

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

Example:
```
PRINT USING "¥¥###.##";12.35,-12.35
  Y12.35 -¥12.35
Ok
PRINT USING "¥¥###.##-";12.35,-12.35
  ¥12.35   ¥12.35-
Ok
```

"**¥"

The **¥ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a yen sign will be printed before the number. **¥ specifies

three more digit positions, one of which is the yen sign.

Example:
```
PRINT USING "**¥#.##";12.35
*¥12.35
Ok
```

","

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.

Example:
```
PRINT USING "####,.##";1234.5
1,234.50
Ok
PRINT USING "####.##,";1234.5
1234.50,
Ok
```

"^^^^"

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

Example:
```
PRINT USING "##.##^^^^";234.56
 2.35E+02
Ok
PRINT USING "#.##^^^^-";-12.34
1.23E+01-
Ok
PRINT USING "+#.##^^^^";12.34,-12.34
+1.23E+01-1.23E+01
Ok
```

"%"

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

Example:
```
PRINT USING "##.##";123.45
%123.45
```

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

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

READ <list of variables>

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

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' error will result. 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.

REM <remark>

; To allow explanatory remarks to be inserted in a program.

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

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

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

Do not use this in a DATA statement as it would be considered legal data.

RESTORE [<line number>]

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

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement. If a nonexistent line number is specified, an 'Undefined Line number'

error will result.

RESUME
RESUME 0
RESUME NEXT
RESUME <line number>
; To continue program execution after an error recovery procedure has been performed.

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 subroutine causes a 'RESUME without' error.

STOP
; To terminate program execution and return to command level.

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

Break in nnnn (nnnn is a line number)

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

Execution is resumed by issuing a CONT command.

SWAP <variable>,<variable>
; To exchange the value of two variables.

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

WAIT <port number>,I[,J]
; To suspend program execution while monitoring the status of a machine input port.

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

the next statement. If J is omitted, it is assumed to be zero.

2.1.3 Functions, except I/O

ABS(X)
; Returns the absolute value of the expression X.

ASC(X$)
; Returns a numerical value that is the ASCII code of the first character of the string X$. If X$ is null, a 'Illegal function call' error is returned.

ATN(X)
; 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 double precision.

BIN$(n)
; Returns a string which represents the binary value of the decimal argument.

n is a numeric expression in the range -32768 to 65535. If n is negative, the two's complement from is used. That is, BIN$(-n) is the same as BIN$(65536-n).

CDBL(X)
; Converts X to a double precision number.

CHR$(I)
; Returns a string whose one element is the ASCII code for I. ASC$ is commonly used to send a special character to the console, etc.

CINT(X)
; Converts X to a integer number by truncating the fractional portion. If X isn't the range -32768 to 32767, an 'Overflow' error occurs.

COS(X)
; Returns the cosine of X in radians. COS(X) is calculated to double precision.

CSNG(X)
; Converts X to a single precision number.

CSRLIN
; Returns the vertical coordinate of the cursor.

EOF(<file number>)
; Return -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTing, to avoid 'Input past end' errors.

ERL/ERR
; When an error handling subroutine is entered, the variable ERR contains the error code for error. and the variable ERL contains

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

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

```
IF 65535=ERL THEN .....
```

Otherwise, use

```
IF ERL=<line number> THEN ....
IF ERR=<error code> THEN....
```

Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement.

EXP(X)
; Returns e to the power of X. X must be <=145.06286085862. If EXP overflows, the 'Overflow' error message is printed.

FIX(X)
; Returns the integer part of X (fraction truncated). 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.

FRE(0)
FRE("")
; Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by BASIC.

FRE(0) returns the number of bytes in memory which can be used for BASIC program, text file, machine language program file, etc. FRE("") returns the number of bytes in memory for string space.

HEX$(X)
; Returns a string which represents the hexadecimal value of the decimal argument.

n is a numeric expression in the range -32768 to 65535. If n is negative, the two's complement from is used. That is, HEX$(-n) is the same as HEX$(65536-n).

INKEY$
; Returns either a one-character string containing a character read from the keyboard or a null string if no key is pressed. No characters will be echoed and all characters are passed through to the program except for Control-C, which terminates the program.

INP(I)
; Returns the byte read from the port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement.

INPUT$(X)
; Returns a string of X characters, read from the keyboard. No character will be echoed and all characters are passed through except Control-C, terminates the execution of the INPUT$ function.

INSTR([I,]X$,Y$)
; 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 0 to 255. If I>LEN(X$) or if X$ is null or if Y$ cannot be found or if X$ and Y$ are null, INSTR returns 0. If only Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions, or string literals.

INT(X)
; Returns the largest integer <=X.

LEFT$(X$,I)
; Returns a string comprising the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) is returned. If I=0, a null string (length zero) is returned.

LEN(X$)
; Returns the number of characters in X$. Nonprinting characters and blanks are counted.

LOG(X)
; Returns the natural logarithm of X. X must be greater than zero.

LPOS(X)
; 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.

MID$(X$,I[,J])
; 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>LEN(X$), MID$ returns a null string.

OCT$(n)
; Returns a string which represents the octal value of the decimal argument.

n is a numeric expression in the range -32768 to 65535. If n is negative, the two's complement from is used. That is, OCT$(-n) is the same as OCT$(65536-n).

PEEK(I)
; Returns the byte (decimal integer in the range 0 to 255) read from memory location I. I must be in the range -32768 to 65535.

PEEK is the complementary function to the POKE statement.

POS(I)
; Returns the current cursor position. The leftmost position is 0. I is a dummy argument.

RIGHT$(X$,I)
; Returns the rightmost I characters of string X$. If I=LEN(X$), return X$. If I=0, a null string (length zero) is returned.

RND(X)
; Returns a random number between 0 and 1. The same sequence of random number is generated each time the program is RUN. If X<0, the random generator is reseeded for any given X. X=0 repeats the last number generated. X>0 generates the next random number in the sequence.

SGN(X)
; Returns 1 (for X>0), 0 (for X=0), -1 (for X<0).

SIN(X)
; Returns the sine of X in radians. SIN(X) is calculated to double precision.

SPACE$(X)
; Returns the string of spaces of length X. The expression X discards the fractional portion and must be range 0 to 255.

SPC(I)
; Prints I blanks on the screen. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255.

SQR(X)
; Returns the square root of X. X must be >=0.

STR$(X)
; Returns a string representation of the value of X.

STRING$(I,J)
STRING$(I,X$)
; Returns a string of length I whose characters all have ASCII code J or the first character of the string X$.

TAB(I)
; Spaces to position I on the console. If the current print position is already beyond space I, TAB does nothing. Space 0 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 0 to 255. TAB may only be used with PRINT and LPRINT statements.

TAN(X)
; Returns the tangent of X in radians. TAN(X) is calculated to double precision. If TAN overflows, an 'Overflow' error will occur.

USR[<digit>](X)
; 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 DEFUSR statement for that routine. If <digit> is omitted, USR0 is assumed.

VAL(X$)
; Returns the numerical value of the string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example

```
PRINT VAL("   -7")
-7
Ok
```

VARPTR(<variable name>)
VARPTR(#<file number>)
; 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 -32768 to 32767. 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 machine language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-address element of the array is returned.

All simple variables should be assigned before calling VARPTR for an array because the address of the arrays change whenever a new simple variable is assigned. If #<file number> is specified, VARPTR returns the starting address of the file control block.

2.2 Device specific statements and functions.

--- Expanded statements and functions for MSX ---

2.2.1 Statements

SCREEN [<mode>][,<sprite size>][,<key click switch>]
[,<cassette baud rate>][,<printer option>]

; To assign the screen mode, sprite size, key click, cassette baud rate and printer option.

<mode> should be set to 0 to select 40x24 text mode, 1 to select 32x24 text mode, 2 to select high resolution mode, 3 to select multi color (low-resolution mode).

0:40x24 text mode
1:32x24 text mode
2:high resolution mode
3:multi color mode

<sprite size> determines the size of sprite. Should be set to 0 to select 8x8 unmagnified sprites, 1 to select 8x8 magnified sprites, 2 to select 16x16 unmagnified sprites, 3 to select 16x16 magnified sprites. NOTE: If <sprite size> is specified, the contents of SPRITE$ will be cleared.

0:8x8 unmagnified
1:8x8 magnified
2:16x16 unmagnified
3:16x16 magnified

<key click switch> determines whether to enable or disable the key click. Should be set to 0 to disable it.

0:disable the key click
1:enable the key click

Note that in text mode, all graphics statements except 'PUT SPRITE' generate an 'Illegal function call' error. Note also that the mode is forced to text mode when an 'INPUT' statement is encountered or BASIC returns to command level.

<cassette baud rate> determines the default baud rate for succeeding write operations. 1 for 1200 baud, and 2 for 2400 baud. Baud rate can also be determined using CSAVE command with baud rate option.

Note that when reading cassette, baud rate is automatically determined, so the user don't have to know in what baud rate the cassette is written. <printer option> determines if the printer in operation is 'MSX printer' (which has 'graphics symbol' and 'HIRAGANA' capability) or not. Should be non-0 if the printer does not have such capability. In this case, graphics symbols

are converted to spaces, and HIRAGANA characters are converted to equivalent KATAKANA characters.

WIDTH <width of screen in text mode>
; To Set the width of display during text mode. Legal value is 1..40 in 40x24 text mode, 1..32 in 32x24 text mode.

CLS
; To clear the screen. Valid in all screen modes.

LOCATE [<x>][,<y>][,<cursor display switch>]
; To locate character position for PRINT. <cursor display switch> can be specified only in text mode.

0:disable the cursor display
1:enable the cursor display

COLOR [<foreground color>][,<background color>][,<border color>]
; To define the color. Defaults to 15,4,7. The argument can be in the range of 0..15. Actual color corresponding to each value is as follows.

0 transparent
1 black
2 medium green
3 light green
4 dark blue
5 light blue
6 dark red
7 cyan
8 medium red
9 light red
10 dark yellow
11 light yellow
12 dark green
13 magenta
14 gray
15 white

PUT SPRITE <sprite plane number>[,<coordinate specifier>][,<color>]
[,<pattern number>]
; To set up sprite attributes.

<sprite plane number> may range from 0 to 31.

<coordinates specifier> always can come in one of two forms:

STEP (x offset, y offset) or
(absolute x, absolute y)

The first form is a point relative to the most recent point referenced. The second form is more common and directly refers to a point without regard to the last point referenced. Examples are:

```
(10,10)       absolute form
STEP (10,0)   offset 10 in x and 0 in y
(0,0)         origin
```

Note that when Basic scans coordinate values it will allow them to be beyond the edge of the screen, however values outside the integer range (-32768 to 32767) will cause an overflow error.

Note that (0,0) is always the upper left hand corner. It may seem strange to start numbering y at the top so the bottom left corner is (0,191) in both high-resolution and medium resolution, but this is the standard.

Above description can be applied wherever graphic coordinate is used.

X coordinate <x> may range from -32 to 255. Y coordinates <y> may range from -32 to 191. If 208 (&HD0) is given to <y>, all sprite planes behind disappears until a value other than 208 is given to that plane. If 209 (&HD1) is specified to <y>, then that sprite disappears from the screen. (Refer to VDP manual for further details.)

When a field is omitted, the current value is used. At start up, color defaults to the current foreground color.

<pattern number> specifies the pattern of sprite, and must be less than 256 when size of sprites if 0 or 1, and must be less than 64 when size of sprites is 2 or 3. <pattern number> defaults to the <sprite plane number>. (See also SCREEN statement and SPRITE$ variable)

CIRCLE <coordinate specifier>,<radius>[,<color>]
[,<start angle>][,<end angle>][,<aspect ratio>]
; To draw an ellipse with a center and radius as indicated by the first of its arguments.

<coordinate specifier> specifies the coordinate of the center of the circle on the screen. For the detail of <coordinate specifier>, see the description at PUT SPRITE statement.

The <color> defaults to foreground color.

The <start angle> and <end angle> parameters are radian arguments between 0 and 2*PI which allow you to specify where drawing of the ellipse will begin and end. If the start or end angle is negative, the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive (Note that this is different than adding 2*PI).

The <aspect ratio> is for horizontal and vertical ratio of the ellipse.

DRAW <string expression>
; To draw figure according to the graphic macro language.

The graphic macro language commands are contained in the string expression string. The string defines an object, which is drawn when BASIC executes the DRAW statement. During execution, BASIC examines the value of string and interprets single letter commands from the contents of the string. These commands are detailed below:

The following movement commands begin movement from the last point referenced. After each command, last point referenced is the last point the command draws.

```
U n     ;Moves up
D n     ;Moves down
L n     ;Moves left
R n     ;Moves right
E n     ;Moves diagonally up und right
F n     ;Moves diagonally down and right
G n     ;Moves diagonally down and left
H n     ;Moves diagonally up and left
```

n in each of the preceding commands indicating the distance to move. The number of points moved is n times the scaling factor (set by the S command).

```
M x,y   ;Moves absolute or relative. If x has a plus
         sign(+) or a minus sign(-) in front of it, it
         is relative. Otherwise, it is absolute.
```

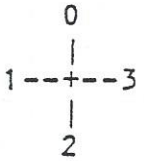
The aspect ratio of the screen is 1. So 8 horizontal points are equal in length to 8 vertical points.

The following two prefix commands may precede any of the above movement commands.

```
B       ;Moves, but doesn't plot any points.
N       ;Moves, but returns to the original position
         when finished.
```

The following commands are also available:

```
A n     ;Sets angle n. n may range from 0 to 3, where
         0 is 0 degree, 1 is 90, 2 is 180, 3 is 270.
```




```
C n     ;Sets color n. n may range 0 to 15.

S n     ;Sets scale factor. n may range from 0 to 255.
         n divided by 4 is the scale factor. For example,
         If n=1, then the scale factor is 1/4. The scale
```

factor multiplied by the distance given with the U,D,L,R,E,F,G,H, and relative M commands gives the actual distance moved. The default value is 0, which means 'no-scaling' (i.e., same as S4)

X<string variable>;
;Executes substring. This allows you to execute a second string from within a string.

Example A$="U80R80D80L80":DRAW "XA$;"
->Draws a square

In all of these commands, the n,x, or y argument can be a constant like 123 or it can be '=<variable>;' where <variable> is the name of a numeric variable. The semicolon (;) is required when you use a variable this way, or in the X command. Otherwise, a semicolon is optional between commands. Spaces are ignored in string. Foe example, you could use variables in a move command this way:

```
X1=40:X2=50
DRAW "M+=x1;,-=X2"
```

The X command can be a very useful part of DRAW, because you can define a part of an object separate from the entire object and also can use X to draw a string of commands more than 255 characters long.

LINE [<coordinate specifier>]-<coordinate specifier>[,<color>] [,<B¶BF>]
; To draw line connecting the two specified coordinate. For the detail of the <coordinate specifier>, see description at PUT SPRITE statement.

If 'B'is specified, draws rectangle. If 'BF' is specified, fills rectangle.

PAINT <coordinate specifier>[,<paint color>][,<color regarded as border>]
; To fill in an arbitrary graphics figure of the specified fill color starting at <coordinate specifier>. For the detail of the <coordinate specifier>, see the description at PUT SPRITE statement.

Note that PAINT must not have border for high resolution graphics, border can be specified only in multicolor mode. In high resolution graphics mode, paint color is regarded as border color.

PSET<coordinate specifier>[,<color>]
PRESET<coordinate specifier>[,<color>]
; To set/reset the specified coordinate. For the detail of the <coordinate specifier>, see the description at PUT SPRITE statement

The only difference between PSET and PRESET is that if no <color> is given in PRESET statement, the background color is selected. When a <color> argument is given, PRESET is identical to PSET.

KEY <function key #>,<string expression>
; To set a string to specified function key. <function key #> must be in the range 1 to 8. <string expression> must be within 15 characters.

Example:

```
KEY 1,"PRINT TIME$"+CHR$(13)
A$="Japan"
KEY 2,A$
```

KEY LIST
; To list the contents of all function keys.

Example:

```
KEY LIST
color
auto
goto
list
run
color 15,7,7
cload"
cont
list .
 run
Ok
```

"color" aligns with key "f1", "auto" with "f2", "goto" with "f3", and so on. Position in the list reflects the key assignments. Note that control characters assigned to a function key is converted to spaces.

ON KEY GOSUB <list of line numbers>
; To set up a line numbers for BASIC to trap to when the function keys is pressed.

example

```
ON KEY GOSUB 100,200,,400,,500
```

When a trap occurs, an automatic KEY(n)STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a KEY(n)ON unless an explicit KEY(n)OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not execution a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all trapping (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

KEY (<function key #>) ON/OFF/STOP
; To activate/deactivate trapping of the specified function key

in a BASIC program.

A KEY(n)ON statement must be executed to activate trapping of function key. After KEY(n)ON statement, if a line number is specified in the ON KEY GOSUB statement then every time BASIC starts a new statement it will check to see if the specified key was pressed. If so it will perform a GOSUB to the line number specified in the ON KEY GOSUB statement.

If a KEY(n)OFF statement has been executed, no trapping takes place and the event is not remembered even if it does take place.

If a KEY(n)STOP statement has been executed, no trapping will take place, but if the specified key is pressed this is remembered so an immediate trap will take place when KEY(n)ON is executed.

KEY(n)ON has no effect on whether the function key value are displayed at the bottom of the console.

ON STRIG GOSUB <list of line numbers>
; To set up a line numbers for BASIC to trap to when the trigger button is pressed.

Example:

```
ON STRIG GOSUB ,200,,400
```

When the trap occurs an automatic STRIG(n)STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a STRIG(n)ON unless an explicit STRIG(n)OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all trapping (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

STRIG (<n>) ON/OFF/STOP
; To activate/deactivate trapping of trigger buttons of joy sticks in a BASIC program.

<n> can be in the range of 0..4. If <n>=0, the space bar is used for a trigger button. If <n> is either 1 or 3, the trigger of a joy-stick 1 is used. When <n> is either 2 or 4, joy-stick 2.

A STRIG(n)ON statement must be executed to activate trapping of trigger button. After STRIG(n)ON statement, if a line number is specified in the ON STRIG GOSUB statement then every time BASIC starts a new statement it will check to see if the trigger button was pressed. If so it will perform a GOSUB to the line number specified in the ON STRIG GOSUB statement.

If a STRIG(n)OFF statement has been executed, no trapping takes place and the event is not remembered even if it does take place.

If a STRIG(n)STOP statement has been executed, no trapping will take place, but if the trigger button is pressed this is remembered so an immediate trap will take place when STRIG(n)ON is executed.

ON STOP GOSUB <line number>
; To set up a line numbers for BASIC to trap to when the Control-STOP key is pressed.

When the trap occurs an automatic STOP STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a STOP ON unless an explicit STOP OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not execution a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all trapping (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

The user must be VERY careful when using this statement. For example, following program cannot be aborted. The only way left is to reset the system!

```
example: 10 ON STOP GOSUB 40
         20 STOP ON
         30 GOTO 30
         40 RETURN
```

STOP ON/OFF/STOP
; To activate/deactivate trapping of a control-STOP.

A STOP ON statement must be executed to activate trapping of a control-STOP. After STOP ON statement, if a line number is specified in the ON STOP GOSUB statement then every time BASIC starts a new statement it will check to see if a control-STOP was pressed. If so, it will perform a GOSUB to the line number specified in the ON STOP GOSUB statement.

If a STOP OFF statement has been executed, no trapping takes place and the event is not remembered even if it does take place.

If a STOP STOP statement has been executed, no trapping will take place, but if a control-STOP is pressed this is remembered so an immediate trap will take place when STOP ON is executed.

ON SPRITE GOSUB <line number>
; To set up a line number for BASIC to trap to when the sprites coincide.

When the trap occurs an automatic SPRITE STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a SPRITE ON unless an explicit SPRITE OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not execution

a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all trapping (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

SPRITE ON/OFF/STOP
; To activate/deactivate trapping of sprite in a BASIC program.

A SPRITE ON statement must be executed to activate trapping of sprite. After SPRITE ON statement, if a line number is specified in the ON SPRITE GOSUB statement then every time BASIC starts a new statement it will check to see if the sprites coincide. If so it will perform a GOSUB to the line number specified in the ON SPRITE GOSUB statement.

If a SPRITE OFF statement has been executed, no trapping takes place and the event is not remembered even if it does take place.

If a SPRITE STOP statement has been executed, no trapping will take place, but if the sprites coincide this is remembered so an immediate trap will take place when SPRITE ON is executed.

ON INTERVAL=<time interval> GOSUB <line number>
; To set up a line number for BASIC to trap to time interval.

Generates a timer interrupt at every <time interval>/60 second.

When the trap occurs an automatic INTERVAL STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a INTERVAL ON unless an explicit INTERVAL OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all traps (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

INTERVAL ON/OFF/STOP
; To activate/deactivate trapping of time interval in a BASIC program.

A INTERVAL ON statement must be executed to activate trapping of time interval. After INTERVAL ON statement, if a line number is specified in the ON INTERVAL GOSUB statement then every time BASIC starts a new statement it will check the time interval. If so it will perform a GOSUB to the line number specified in the ON INTERVAL GOSUB statement.

If a INTERVAL OFF statement has been executed, no trapping takes place and the event is not remembered even if it does take place.

If a INTERVAL STOP statement has been executed, no trapping will take place, but if the timer interrupt occur, this is remembered so an immediate trap will take place when INTERVAL ON is executed.

VPOKE <address of VRAM>,<value to be written>

; To poke a value to specified location of VRAM. <address of VRAM> can be in the range of 0..16383. <value to be written> should be a byte value.

BEEP

; To generate a beep sound. Exactly the same with outputting CHR$(7).

MOTOR [<ON¶OFF>]

; To change the status of cassette motor switch. When no argument is given, flips the motor switch. Otherwise, enables/disables motor of cassette.

SOUND <register of PSG>,<value to be written>

; To write value directly to the <register of PSG>.

PLAY <string exp for voice 1>[,<string exp for voice 2> [,<string exp for voice 3>]]

; To play music according to music macro language.

PLAY implements a concept similar to DRAW by embedding a "music macro language" into a character string. <string exp for voice n> is a string expression consisting of single character music commands. When a null string is specified, the voice channel remains silent. The single character commands in PLAY are:

A to G with optional #,+,or -
;Plays the indicated note in the current octave. A number sign(#) or plus sign(+) afterwards indicates a sharp, a minus sign(-) indicates a flat. The #,+, or - is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

O n ;Octave. Sets the current octave for the following notes. There are 8 octaves, numbered 1 to 8. Each octave goes from C to B. Octave 4 is the default octave.

N n ;Plays note n. n may range from 0 to 96. n=0 means rest. This is an alternative way of selecting notes besides specifying the octave(O n) and the note name (A-G). (The C of octave 4 is 36.)

L n ;Sets the length of the following notes. The actual note length is 1/n. n may range from 1 to 64. The following table may help explain this:

Length	Equivalent
L1	whole note

L2	half note
L3	one of a triplet of three half notes (1/3 of a 4 beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter note triplet
.	
.	
L64	sixty-forth note

The length may also follow the note when you want to change the length only for the note. For example, A16 is equivalent to L16A.

R n ;Pause(rest). n may range from 1 to 64, and figures the length of the pause in the same way as L(length).

. ;(Dot or period) After a note, causes the note to be played as a dotted note. That is, its length is multiplied by 3/2. More than one dot may appear after the note, and the length is adjusted accordingly. For example, "A..." will play 27/8 as long, etc. Dots may also appear after the pause(P) to scale the pause length in the same way.

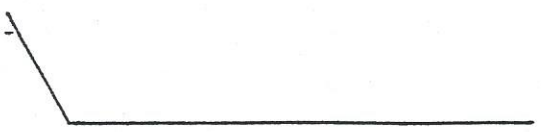
T n ;Tempo. Sets the number of quarter notes in a minute. n may range from 32 to 255. The default is 120.

V n ;Volume. Sets the volume of output. n may range from 0 to 15.

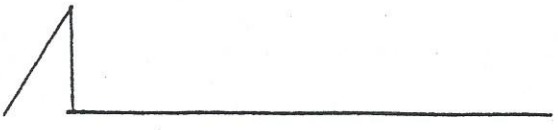
M n ;Modulation. Sets period of envelope. n may range from 1 to 65535.

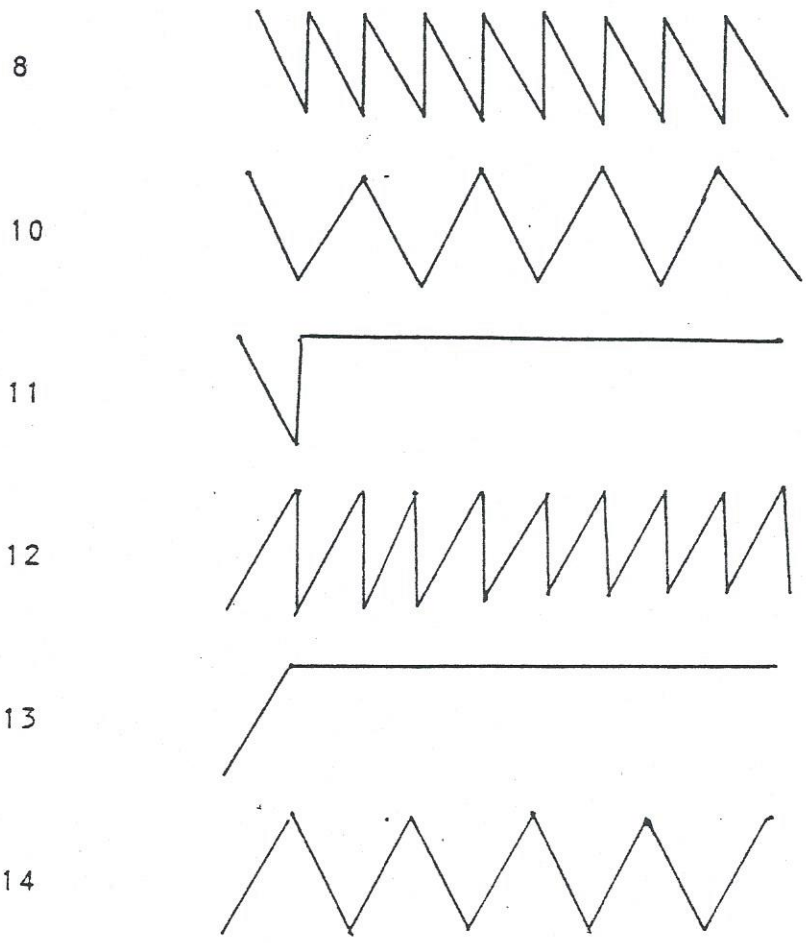
S n ;Shape. Sets shape of envelope. n may range from 1 to 15. The pattern set by this command are as follows:

0,1,2,3,9



4,5,6,7,15






```
X<variable>;
        ;Executes specified string.
```

In all of these commands the n argument can be a constant like 12 or it can be "=<variable>;" where variable is the name of a variable. The semicolon(;) is required when you use a variable in this way, and when you use the X command. Otherwise, a semicolon is optional between commands.

```
MAXFILES=<expression>
    ; To specify the maximum number of files opened at a time.
      <expression> can be in the range of 0..15. When 'MAXFILES=0'
      is executed, only SAVE and LOAD can be performed.
      The default value assigned is 1.

OPEN "<device_descriptor>[<file name>]" [FOR <mode>]
            AS [#]<file number>
    ; To allocate a buffer for I/O and set the mode that will be used
      with the buffer.
```

This statement opens a device for further processing. Currently, following devices are supported.

```
CAS:    cassette
CRT:    CRT screen
GRP:    Graphic screen
```

LPT: line printer

Device descriptors can be added using the ROM cartridge. See SLOT.MEM for further details.

<mode> is one of the following:

OUTPUT : Specifies sequential output mode
INPUT : Specifies sequential input mode
APPEND : Specifies sequential append mode

<file number> is an integer expression whose value is between one and the maximum number of files specified in a MAXFILES= statement.

<file number> is the number that is associated with the file for as long as it is OPEN and is used by other I/O statements to refer to the file.

An OPEN must be executed before any I/O may be done to the file using any of the following statements, or any statement or function requiring a file number:

PRINT #, PRINT # USING
INPUT #, LINE INPUT #
INPUT$, GET, PUT

PRINT #<file number>,<exp>
PRINT #<file number>,USING <string expression>;<list of expression>
; To write data to the specified channel. (See PRINT/PRINT USING statements for details.)

INPUT #<file number>,<variable list>
; To read data items from the specified channel and assign them to program variables.

The type of data in the file must match the type specified by the <variable list>. Unlike the INPUT statement, no question mark is printed with INPUT# statement.

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 start of a number. The number terminates on a space, carriage return, line feed, or comma.

Also, if the BASIC is scanning the data for a string item, 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 string item. If this first character is a double-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 return, 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.

LINE INPUT #<file number>,<string variable>
; To read an entire line (up to 254 characters), without delimiters, from a sequential file to a string variable.

<file number> is the number which the file was OPENed.

<string variable> is the name of a string variable 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. That is, the line feed/carriage return characters are returned as part of the string.)

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

INPUT$(n,[#]<file number>)
; To Return a string of n characters, read from the file. <file number> is the number which the file was OPENed.

CLOSE [[#]<file number>[,<file number>]]
; To close the channel and releases the buffer associated with it. If no <file number>'s are specified, all open channels are closed.

SAVE "<device descriptor>[<file name>]"
; To save a BASIC program file to the device. Control-Z is treated as end-of-file.

LOAD "<device_descriptor>[<file name>]"
; To load a BASIC program file from the device.

LOAD closes all open files and deletes the current program from memory. However, with the "R" option, all data files remain OPEN and execute the loaded program.

If the <file name> is omitted, the next program, which should be an ASCII file, encountered on the tape is loaded. Control-Z is treated as end-of-file.

MERGE "<device descriptor>[<file name>]"
; To merge the lines from an ASCII program file into the program

currently in memory.

If any lines in the file being merged have the same line number as lines in the program in memory, the lines from the file will replace the corresponding lines in memory.

After the MERGE command, the MERGEd program resides in memory, and BASIC returns to command level.

If the <file name> is omitted, the next program files, it should be ASCII file, file encountered on the tape is MERGEd. Control-Z is treated as end-of-file.

BSAVE "<device descriptor>[<file name>]",<top adrs>,<end adrs>
[,<execution adrs>]
; To save a memory image at the specified memory location to the device. (Currently, only CAS: is supported.)

<top adrs> and <end adrs> are the top address and the end address of the area to be saved.

If <execution adrs> is omitted, <top adrs> is regarded as <execution adrs>.

Example:

```
BSAVE "CAS:TEST",&HA000,&HAFFF
BSAVE "CAS:GAME",&HE000,&HE0FF,&HE020
```

BLOAD "<device_descriptor>[<file name>]"[,R][,<offset>]
; To load a machine language program from the specified device. (Currently only CAS: is supported.)

If R option is specified, after the loading, program begins execution automatically from the address which is specified at BSAVE.

The loaded machine language program will be stored at the memory location which is specified at BSAVE. If <offset> is specified, all addresses which are specified at BSAVE are offset by that value.

If the <file name> is omitted, the next machine language program file encountered is loaded.

CSAVE "<file name>"[,<baud rate>]
; To save a BASIC program file to the cassette tape.

BASIC saves the file in a compressed binary (tokenized) format. ASCII files take up more space, but some types of access require that files be in ASCII format. For example, a file intended to be MERGEd must be saved in ASCII format. Programs saved in ASCII may be read as BASIC data files and text files. In that case, use the SAVE command.

<baud rate> is a parameter from 1 to 2, which determines the default baud rate for every cassette write operations. 1 for 1200 baud, 2 for 2400 baud. The default baud rate can also be set with SCREEN statement.

CLOAD ["<file name>"]
; To load a BASIC program file from the CMT.

CLOAD closes all open files and deletes the current program from memory. If the <file name> is omitted, the next program file encountered on the tape is loaded. For all cassette read operations, baud rate is determined automatically.

CLOAD? ["<file name>"]
; To verify a BASIC program on CMT with one in memory.

CALL <name of expanded statement>[(<argument list>)]
; To invoke an expanded statement supplied by ROM cartridge. See SLOT.MEM for further details. '_' is an abbreviation for 'CALL', so the next 2 statements have the same meaning.

```
CALL TALK("Yamashita","Hayashi","Suzuki GSX400FW")
_TALK("Yamashita","Hayashi","Suzuki GSX400FW")
```

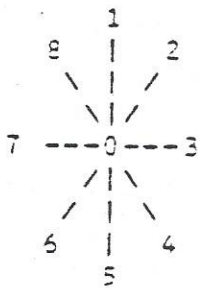
2.2.2 Functions

BIN$(<expression>)
; Returns a binary equivalent string of <expression>. Leading 0's are suppressed.

POINT(<X coordinate>,<Y coordinate>)
; Returns color of a specified pixel.

VPEEK(<address of VRAM>)
; Returns a value of VRAM specified. <address of VRAM> can be in the range of 0..16383.

STICK(<n>)
; Returns the direction of a joy-stick. <n> can be in the range of 0..2. If <n>=0, the cursor key is used as a joy-stick. If <n> is either 1 or 2, the joy-stick connected to proper port is used. When neutral, 0 is returned. Otherwise, value corresponding to direction is returned.



STRIG(<n>)
; Returns the status of a trigger button of a joy-stick. <n> can be in the range of 0..4. If <n>=0, the space bar is used for a trigger button. If <n> is either 1 or 3, the trigger of a joy-stick 1 is used. When <n> is either 2 or 4, joy-stick 2. 0 is returned if the trigger is not being pressed, -1 is returned otherwise.

PDL(<n>)
; Returns the value of a paddle. <n> can be in the range of 1..12. When <n> is either 1, 3, 5, 7, 9 or 11, the paddle connected to port 1 is used. When 2, 4, 6, 8, 10 or 12, the paddle connected to port 2 is used.

PAD(<n>)
; Returns various status of touch pad. <n> can be in the range of 0..7.

When 0..3 is specified, touch pad connected to joy stick port 1 is selected, when 4..7, port 2.

When <n>=0 or 4, the status of touch pad is returned, -1 when touched, 0 when released.

When <n>=1 or 5, the X-coordinate is returned, when <n>=2 or 6, Y-coordinate is returned.

When <n>=3 or 7, the status of switch on the pad is returned, -1 when being pushed, 0 otherwise.

Note that coordinates are valid only when PAD(0) (or PAD(4)) is evaluated. When PAD(0) is evaluated, PAD(5) and PAD(6) are both affected, and when PAD(4), PAD(1) and PAD(2).

PLAY(<play channel>)
; Returns the status of a music queue. <n> can be in the range of 0..3. If <n>=0, all 3 status are ORed and returned. If <n> is either 1,2 or 3, -1 is returned if the queue is still in operation, i.e., still playing. 0 is returned otherwise.

EOF(<file_number>)
; Returns -1 if end-of-file is encountered from input device. Otherwise, returns 0

2.2.3 Special variables

Following are the special variables for MSX. When assigned, the content is changed, when evaluated, the current value is returned.

TIME (type: unsigned integer)
; The system internal timer. TIME is automatically incremented by 1 everytime VDP generates interrupt (60 times per second), thus, when an interrupt is disabled (for example, when manipulating cassette), it retains the old value.

SPRITE$(<pattern number>) (type: string)
; The pattern of sprite.

<pattern number> must be less than 256 when size of sprites is 0 or 1, less than 64 when size of sprites is 2 or 3.

The length of this variable is fixed to 32 (bytes). So, if assign the string that is shorter than 32 character, the chr$(0)s are added.

Example

```
list
100 SCREEN 3,3
110 A$=CHR$(1)+CHR$(3)+CHR$(7)+CHR$(&HF)+CHR$(&H1F)
+CHR$(&H3F)+CHR$(&H7F)+CHR$(&HFF)
120 SPRITE$(1)=A$
130 SPRITE$(2)=A$+A$
140 SPRITE$(3)=A$+A$+A$
150 SPRITE$(4)=A$+A$+A$+A$
160 PUT SPRITE 1,(20,20),15
170 PUT SPRITE 2,(60,20),15
180 PUT SPRITE 3,(100,20),15
190 PUT SPRITE 4,(140,20),15
200 GOTO 200
Ok
run
```

```
**********************************************************************
*
*  Note: Following two are system variables which can be evaluated
*        or assigned like other ordinary variables. Prepared for
*        advanced programmers only. If you don't know the meaning,
*        never use.
*
**********************************************************************
```

VDP(<n>) (type: unsigned byte)
; If <n> is in the range of 0..7, specifies the current value of VDP's write only register. If <n> is 8, specifies the status register of VDP. VDP(8) is read only.

BASE(<n>) (type: integer)
; Current base address for each table. The description of <n> follows next.

0 - base of name table for text mode.
1 - meaningless
2 - base of pattern generator table for text mode.
3 - meaningless
4 - meaningless

5 - base of name table for text mode.
6 - base of color table for text mode.
7 - base of pattern generator table for text mode.
8 - base of sprite attribute table for text mode.
9 - base of sprite pattern table for text mode.

10 - base of name table for high-resolution mode.
11 - base of color table for high-resolution mode.
12 - base of pattern generator table for high-resolution mode.
13 - base of sprite attribute table for high-resolution mode.
14 - base of sprite pattern table for high-resolution mode.

15 - base of name table for multi-color mode.
16 - meaningless
17 - base of pattern generator table for multi-color mode.
18 - base of sprite attribute table for multi-color mode.
19 - base of sprite pattern table for multi-color mode.

CHAPTER 3

APPENDIX

A. Summary of error codes and error messages

code message

1 NEXT without FOR

A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.

2 Syntax error

A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.)

3 RETURN without GOSUB

A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.

4 Out of DATA

A READ statement is executed when there are no DATA statement with unread data remaining in the program.

5 Illegal function call

A parameter that is out of the 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. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR$ or ON...GOTO.

6 Overflow

The result of a calculation is too large to be represented in BASIC's number format.

7 Out of memory

A program is too large, has too many files, has too many FOR loops or GOSUBs, too many variables, or expressions that are too

complicated.

8 Undefined line number
A line reference in a GOTO, GOSUB, IF...THEN...ELSE is to a nonexistent line.

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.

10 Redimensioned array
Two DIM statements are given for the same array, or DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power.

12 Illegal direct
A statement that is illegal in direct mode is entered as a direct mode command.

13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

14 Out of string space
String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.

15 String too long
An attempt is made to create a string more than 255 character long.

16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 Can't continue
An attempt is made to continue a program that:

1. has halted due to an error,
2. has been modified during a break in execution, or

3. does not exist.

18 Undefined user function
FN function is called before defining it with the DEF FN statement.

19 Device I/O error
An I/O error occurred on a cassette, printer, or CRT operation. It is a fatal error; i.e., BASIC cannot recover from the error.

20 Verify error
The current program is different from the program saved on the cassette.

21 No RESUME
An error trapping routine is entered but contains no RESUME statement.

22 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

23 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.

24 Missing operand
An expression contained an operator with no operand following it.

25 Line buffer overflow
An entered line has too many characters.

26 Unprintable errors
.
49 These codes have no definitions. Should be reserved for future expansion in BASIC.

50 FIELD overflow
A FIELD statement is attempting allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or, the end of the FIELD buffer is encountered while doing sequential I/O(PRINT#,INPUT#) to a random file.

51 Internal error
An internal malfunction has occurred. Report to Microsoft the conditions under which the message appeared.

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 by MAXFILE statement.

53 File not found
A LOAD, KILL, or OPEN statement references a file that does not exist in the memory.

54 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.

55 Input past end
An INPUT statement is executed after all the data in the file has been INPUT, or for null (empty) file. To avoid this error, use the EOF function to detect the end of file.

56 Bad file name
An illegal form is used for the file name with LOAD, SAVE, KILL, NAME, etc.

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

58 Sequential I/O only
A statement to random access is issued for a sequential file.

59 File not OPEN
The file specified in a PRINT#, INPUT#, etc. hasn't been OPENed.

60 Unprintable error
. These codes have no definitions. Users may
. place their own error code definitions at the
255 high end of this range.