AVT

# PERSONAL COMPUTER

DPC-200

MSX

# PERSONAL COMPUTER
# DPC-200

# INDEX

# 1. System installation

- Unpacking
- Ports and Sockets
- Power cable connection
- Power on
- System installation
- How to connect with your TV
- How to connect with Data Recorder
- How to connect with Joystick

# 2. Key board

- Key board layout
- Control keys
- Editing keys
- Function keys
- Graphic keys layout
- Code keys layout

# 3. Editing

- Screen editor
- Insert mode
- Delete mode
- Copy mode

## 4. Numbers and variable

- Constants
- Numeric constants
- Variables
- Array Variables
- Type conversion
- Expressions and operators
- Arithmetic operators
- Interger division and modulus arithmetic
- Overflow and division by zero
- Relational operators
- Logical operators
- Functional operators
- String operators

## 5. Graphics

- Screen 0 mode
- Screen 1 mode
- Screen 2 mode
- Screen 3 mode
- Color
- Border, Background and Foreground color
- Circle
- Paint
- Line and box drawing
- Sprites

# 6. Sound

- Play
- "O" (Octave)
- "T" (Tempo)
- "L" (Length)
- "S" (Shape)
- "M" (Tone)
- "R" (Rest)
- "V" (Volume)
- Using a channels of sound
- Tone generator control
- Amplitude control
- Mixer control
- Register 7
- Envelope Period control register
- Shape register
- PSG block diagram

# 7. MSX-basic

# 8. Sample programs

# Appendices

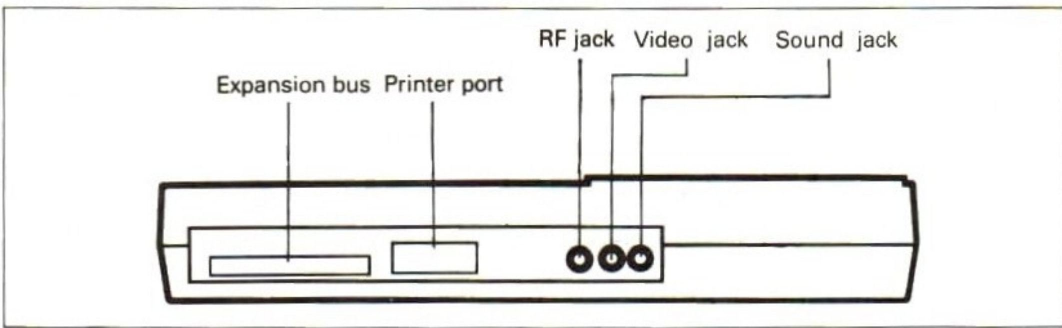
- Appendix A : Control code table
- Appendix B : Character code table
- Appendix C : Slot arrangement
- Appendix D : Memory map
- Appendix E : I/O map
- Appendix F : Pinouts for Input/output devices
- Appendix G : Error messages and error codes
- Appendix H : MSX-Basic reserved words
- Appendix I : Mathematical functions
- Appendix J : Trouble shooting chart

# System Installation

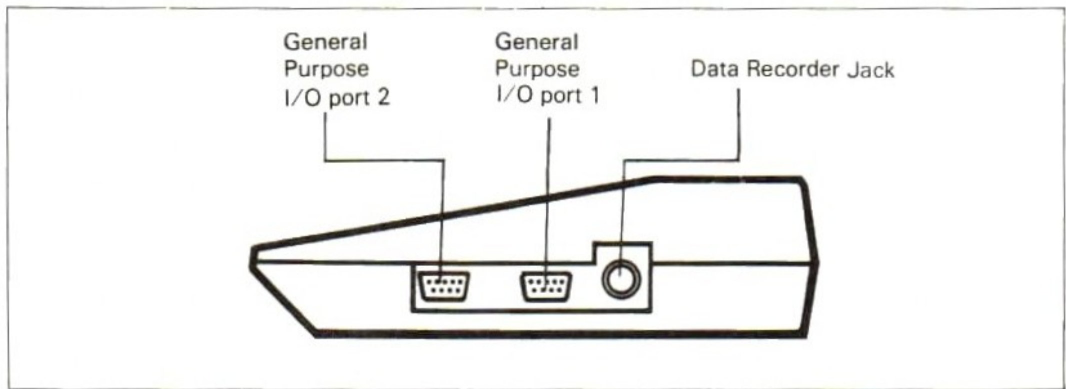
# Unpacking

The computer, video and sound cable, RF cable and data recorder cable are packed in a foam cushioned carton.
Please save all packaging materials in case you must ship the unit for maintenance of repairs.

# Ports and Sockets




Expansion bus : Connects the external bus to the computer.
Printer port : Connects the printer to the computer.
Video jack : This port connects your T.V. or monitor to the computer.
Sound jack : This port connects your T.V. or monitor to the computer.
RF jack : This port is connected to antenna jack of your T.V.



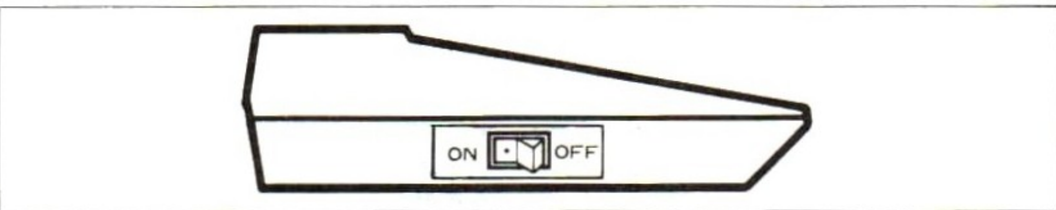

Data recorder jack : Connects the computer to the cassette recorder.
General purpose I/O Port : These ports connect optional joystick, graphic tablet or paddle to the computer.

# Power cable connection

Before connecting the power, please check and be sure the power switch on the left side of the units is off.
Connect the power cable to wall outlet.

# Power on

After you have connected the computer to power first turn on your T.V. or monitor then turn the computer power switch to the "ON" position.




Power Switch: Turns on the power to the Computer.

The power on indicator on the keyboard panel will light up. You should see the message displayed on the screen.

MSX System
Version 1.0
Copyright 1983 by Microsoft.

After a few second, you should see the next message displayed on the screen.

MSX BASIC Version 1.0
Copyright 1983 by Microsoft.
28815 Bytes free
OK

Now you can start your work with the computer. If the system still does not start up properly refer to the trouble shooting chart.
(Appendix J)

# System Installation

To connect the system, you will need 220V electrical outlets for your computer and monitor, or T.V. set.
Choose a comportable position for operation, away from any source of extreme heat (sunlight, heater, etc).
The computer is designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause interference to radio or television reception, which can be determined by turning equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient the receiving antenna
- Relocate the computer with respect to the receiver
- Move the computer away from the receiver
- Plug the computer into a different outlet so that computer and receiver are on different branch circuits.

If necessary, the user should consult the dealer or an experienced radio/T.V. technician for additional suggestions.

# How to connect your T.V.

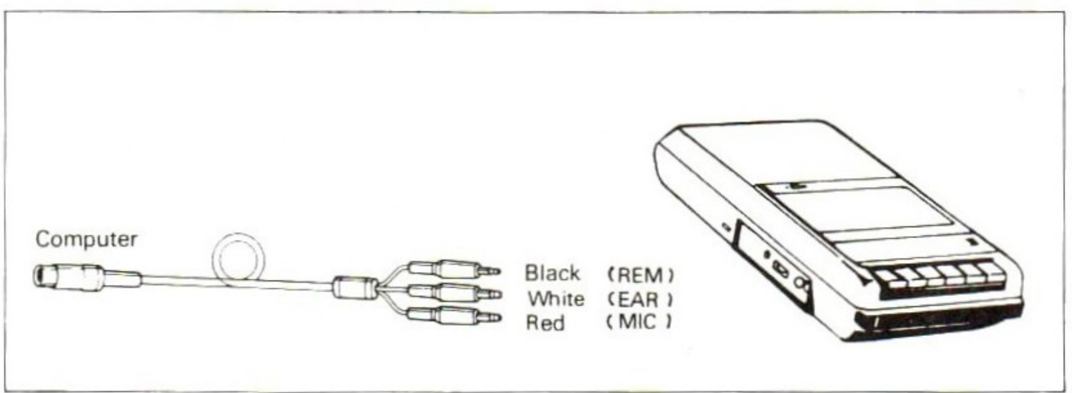
O. Connect one end of the video cable to the RF modulator.

O. Connect the other end of the video cable to the T.V set.

# How to connect with Data Recorder.

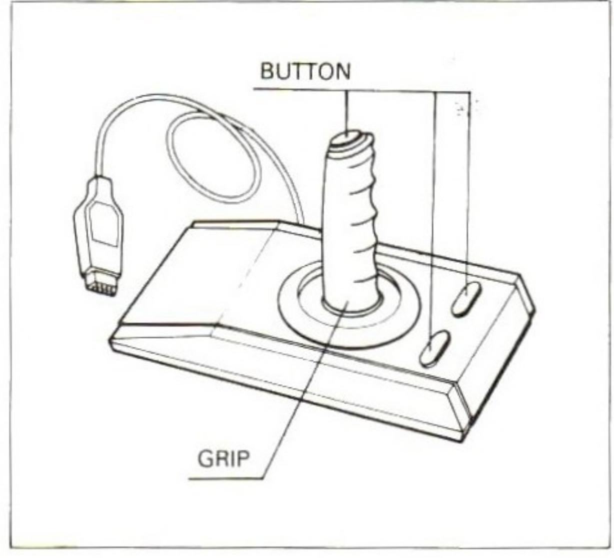
O. Attach one end of the data recorder cable to the 8 pin DIN jack at the side panel of the computer.

O. The other end consists of three cables.
In case,
Black cable connects to the REM input of the data recorder.
White cable connects to the EAR output of the data recorder.
Red cable connects to the MIC input of the data recorder.




# How to connect with Joystick

Joystick cable connects directly to the 9 pin Jack at the side panel of the computer.



# System Installation

To connect the system, you will need 220V electrical outlets for your computer and monitor, or T.V. set.
Choose a comportable position for operation, away from any source of extreme heat (sunlight, heater, etc).
The computer is designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause interference to radio or television reception, which can be determined by turning equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient the receiving antenna
- Relocate the computer with respect to the receiver
- Move the computer away from the receiver
- Plug the computer into a different outlet so that computer and receiver are on different branch circuits.

If necessary, the user should consult the dealer or an experienced radio/T.V. technician for additional suggestions.

# How to connect your T.V.

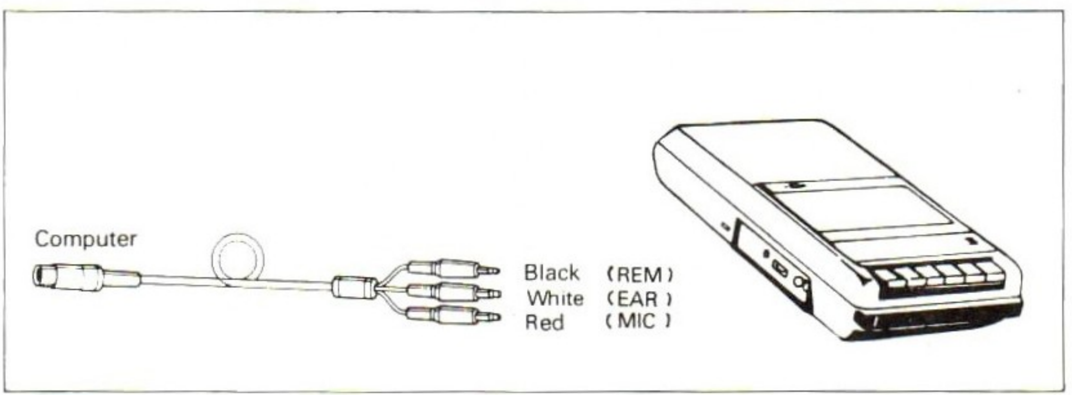
O. Connect one end of the video cable to the RF modulator.

O. Connect the other end of the video cable to the T.V set.

# How to connect with Data Recorder.

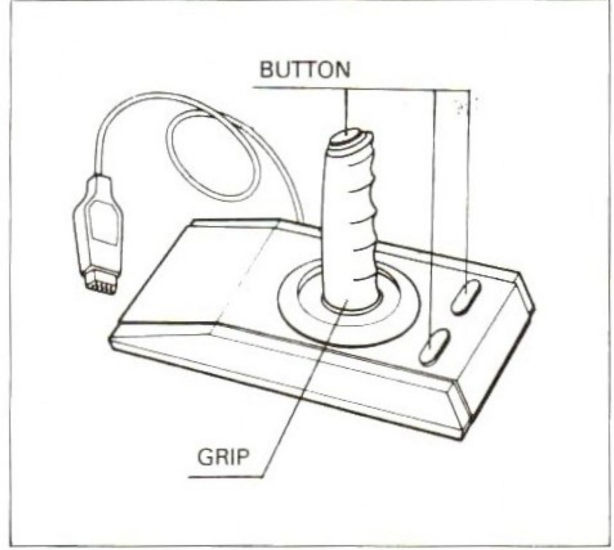
O. Attach one end of the data recorder cable to the 8 pin DIN jack at the side panel of the computer.

O. The other end consists of three cables.
In case,
Black cable connects to the REM input of the data recorder.
White cable connects to the EAR output of the data recorder.
Red cable connects to the MIC input of the data recorder.




# How to connect with Joystick

Joystick cable connects directly to the 9 pin Jack at the side panel of the computer.

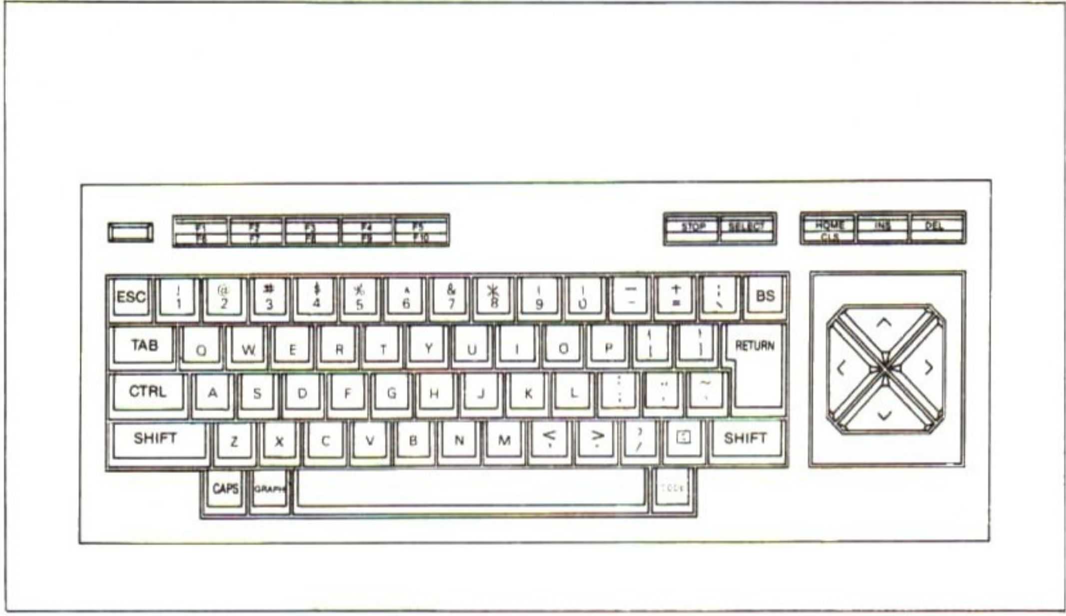


# KEY BOARD

Programming is generally done by sending instructions to the computer through the keyboard. Your instructions and the computer's responses are visible on a TV screen, which is connected to the computer.

The computer's keyboard should look somewhat familiar to you because it resembles that of a typewriter. However, the keyboard contains additional keys that are necessary to effectively communicate with the computer.

## Keyboard Layout




The keyboard of your computer consists of alphabetic keys, numeric keys, special character keys and special function keys.

# Control Keys

| | |
|---|---|
| STOP | Pressing this key will pause the program execution.<br>Pressing this key again will resume the execution. |
| CTRL | The CONTROL Key. Pressing this key and a character key simultaneously tells the computer to work a special function.<br>(See Appendix B)<br>Pressing this key and STOP key simultaneously will terminates the program execution and return to control back over to you. |
| RETURN | Press this key at the end of each instruction you type.<br>By pressing this key you are telling the computer to enter the instruction you just typed into its memory.<br>Note: RETURN indicates you to press "RETURN" key. |
| SELECT | This key has no function in BASIC programming and are only accessed from programs. Pressing this key will generate the control code 24. |
| ESC | This key has no function in BASIC programming and is only accessed from programs. Pressing this key will generate the control code 27 |

# Editing Keys

**INS** This key is used when you wish to insert characters within a line. The key acts like a 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 the cursor returns to the normal size.

**DEL** This key is used when you wish to delete character at current cursor position. All characters to the right of the cursor are moved left one position. Line wrap is observed.

**BS** BACK SPACE. Deletes the character to the left of the cursor. All characters to the right of the cursor are moved left one position. Line wrap is observed.

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

**HOME** Moves the cursor to the upper left corner (home position) of the screen. When this key and shift key are pressed simultaneously it moves the cursor to the home position and clears the entire screen.

**CAPS** Pressing this key will toggle the display alphabetic characters from lower case to upper case or upper case to lower case.

**SHIFT** In typing the alphabetic characters, this key acts like a CAPS key. Pressing this key and the numeric key or special character key will display upper printed symbol on the key-top.

**GRAPH** Pressing this key and the character key simultanously will display the Graphic symbol.
The detailed figure of each key will be covered in later section.

**CODE** Pressing this key and the character key simultaneously will display the special character symbols.
The detailed figure of each key will be covered in later section.

**´ `** DEAD key. Pressing this key and a vowel simultaneously, will be displayed a special character.

## CURSOR CONTROL KEY

The cursor control keys (up, down, left and right) control the movement of the cursor.
By pressing a combination of the up and left arrow keys, you will cause the cursor to move towards the upper left corner of the display screen.
Other combinations will work in the same fashion giving you 8 directions of cursor movement using these keys.

# Function Keys

The keys is located at the top row of keys on the keyboard, and each one is marked with the letter "F". They are a labor-saving devices because they allow you to instruct the computer to perform a frequently used function by pressing only one key instead of having to type many keys.

Here is a list of each key, the function it performs and a brief description of the function.

Function keys [F1] through [F5] are operated by pressing the appropriate key. Function keys [F6] through [F10] are operated by pressing the [SHIFT] key and holding it down while simultaneously pressing the appropriate key.

Here is the function keys meaning

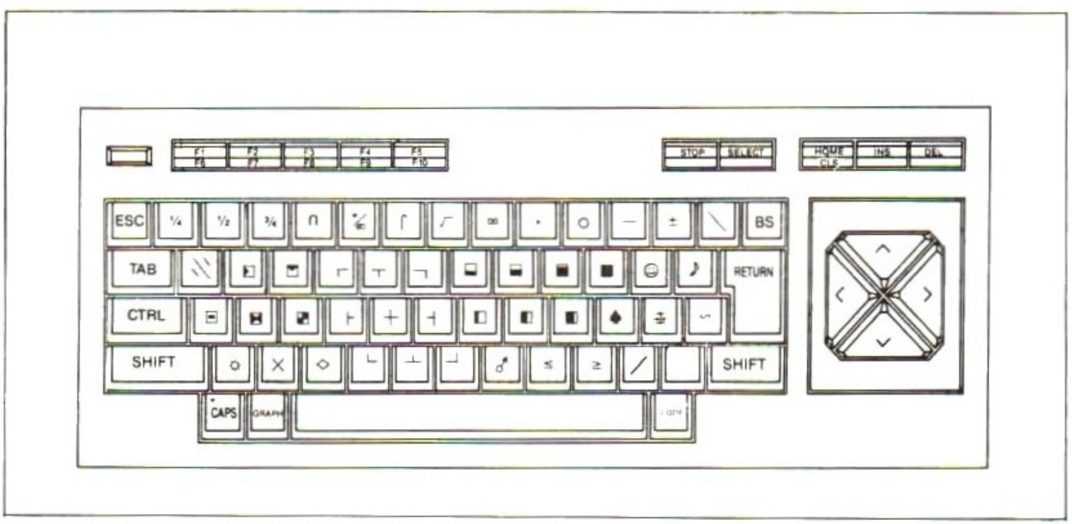
| Key | Function | Description |
|---|---|---|
| F 1 | color | The color command is used to change the text, background and border colors on screen. |
| F 2 | auto | The auto command is used to make the computer generate program line numbers automatically. This command is used very often, since all programming statements must be preceded by line numbers. |
| F 3 | goto | goto is a command which provides you with the ability to execute your program from any place (line number) you desire. |
| F 4 | list | This command instructs your computer to print all your immediately preceding program statements on the screen. |
| F 5 | run+return | run tells the computer to take the program you have written and perform the commands you have indicated. |
| F 6 | color 15, 4, 4 +return | This tells the computer to print white letters on a blue background with a light blue border. These colors are the colors of the screen when you turn the computer on. |
| F 7 | cload" | cload" instructs the computer to input (load) data from a cassette recorder (which can be easily connected to your computer) |

| | | |
|---|---|---|
| F 8 | cont+return | This command is used to tell the computer to "continue" program execution after the last executed line. |
| F 9 | list+return | With this LIST. (with a period next to it) only the last line you were working on (whether programming, editing, etc.) will be displayed on the screen. |
| F10 | CLS+run+ return | This command is similar to the standard RUN command. However this command also clears the screen before it "runs" your program. |

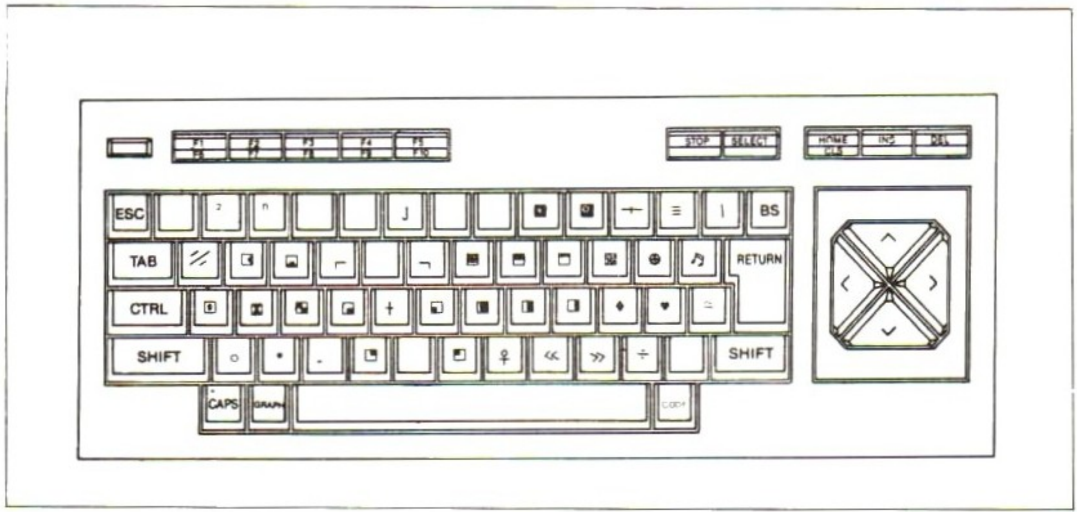
The computer will normally display the function of keys F1 through F5 and whenever you press the shift key it displays the function of keys F6 through F10

Any of these pre-defined functions can be quickly changed for your own convenience to a function that you frequently use. See the MSX Basic Command "KEY" for further details.

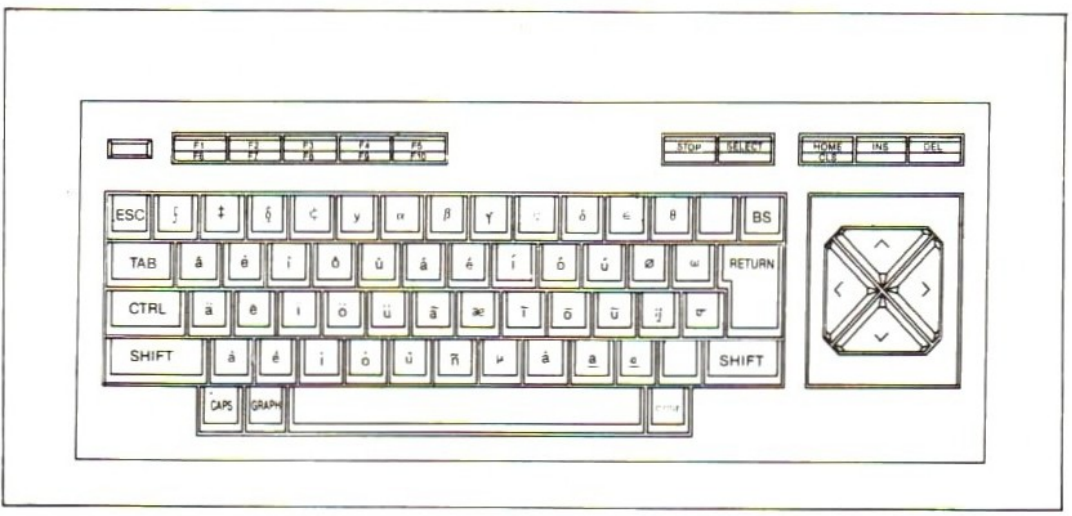
# Graphic Keys Layout



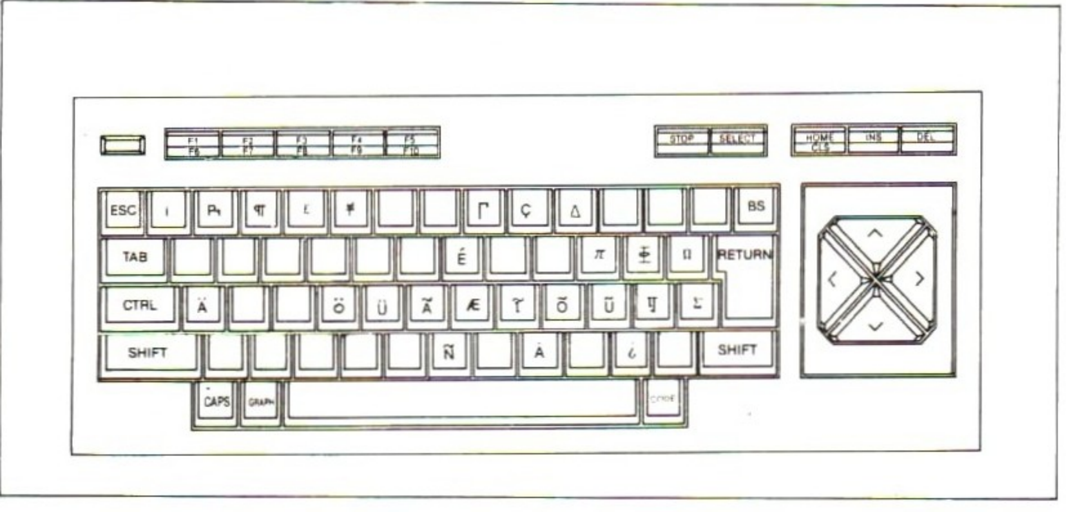

# Graphic + Shift Keys Layout



## Code Keys Layout




## Code + Shift Keys Layout



# EDITING

The MSX-BASIC Screen Editor lets you change a line any where on the screen. You can change only one line at a time. The Screen Editor can be used after an OK prompt appears and before a RUN command is issued. By using the cursor control keys and the editing keys, you can move quickly around the screen, making corrections where necessary.

## "SCREEN EDITOR"

Here is an example to show you how to use the screen editor. A more detailed description of the Screen Editor's syntax can be found in the MSX-Basic COMMAND.

Let's enter the following program.

```
10 PRINT "MERIO" [RETURN]
20 PRINT "MARIA" [RETURN]
30 END  RETURN
```

Remember that a program line must always begin with a line number.
If you make a mistake, just press RETURN and retype the line.

After you have finished, press the CLS/HOME key then type LIST or F4 and press RETURN you should see:

```
LIST

10 PRINT "MERIO"
10 PRINT "MARIA"
30 END
OK
```

Now let's correct the word MARIO in line 10. First use the cursor Key's up direction key to move to line 10 and then the RIGHT direction key to move the cursor to the top of the letter "E" of "MERIO"

Press the letter "A" to change the word to "MARIO", then press RETURN. The line will be stored now as:

```
10 PRINT "MARIO"
```

You have just replaced the character "E" with the character "A". To verify this, press CLS/HOME, F4 (list), RETURN. You should see:

```
LIST
    10 PRINT "MARIO"
    20 PRINT "MARIA"
    30 END
    OK
```

The next step is to insert the two words SHE IS into line 10. We do this by moving the cursor with the cursor control to "M" of "MARIO" in line 10.

## "INSERT" mode

Now press the INS, key and the cursor will become half as tall as before. This means you are in the "INSERT" mode. Type SHE IS and press RETURN.

You have just inserted the words "SHE IS". Follow the steps you used to verify line 10 and you will see:

LIST

```
10 PRINT "SHE IS MARIO"
20 PRINT "MARIA"
30 END
OK
```

Besides using the Screen Editor, you can also change a line by entering a new one with the same line number. BASIC will automatically replace that line.

## "DELETE" mode

The next step is to delete the one character S in line 10.
We do this by moving the CURSOR to "S" of "SHE" in line 10.
Now press the DEL key and press "RETURN".
You have just DELETE the character "S" of "SHE" in line 10.
Follow the steps you used to verify line 10 and you will see.

```
10 PRINT "HE IS MARIO"
20 PRINT "MARIA"
30 END
OK
```

## "COPY" mode

Type new, RETURN. Now all you have done is cleared.
Let's enter the following line.

10 PRINT "HE IS MARIO" RETURN.
Then move the cursor to up direction "1" of "10" in line 10.
Press the letter "2" and RETURN to change the line number 10 to 20.

You have just duplicated the line 10 to make line 20.
To verify this, press CLS/HOME, F4 (list), RETURN. You will see:

```
LIST
        10 PRINT "HE IS MARIO"
        20 PRINT "HE IS MARIO"
        OK
```

Now it is your turn to change the sentence "HE IS MARIO" to "SHE IS MARIA" in line 20. Try to do this with your MSX SCREEN EDITOR.

# NUMBERS AND VARIABLE

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.

## Constants

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

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

Exampless:

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

Numeric constants are positive or negative numbers. MSX-BASIC numeric constants cannot contain commas. There are six 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 ($10^{-64}$) to 10+63 ($10^{63}$).

   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. |
| | Examples: |
| | &O347 |
| 6. Binary constants | Binary numbers, denoted by the prefix &B. |
| | Examples: |
| | &B01110110<br>&B11100111 |

# 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-90
   3489.0#
   7654321.1234
   ```

# 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.

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.

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.

Examples:

3X, wrong : The first character is a number.
B/3, wrong : The special character "/" is not allowed.
COST, wrong : This variable name contains a reserved word "COS"

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 a 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.

## 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. But the maximum number of elements is determined by memory size.

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.

# 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
   ```

## 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

# 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<br>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.

# Integer Division and Modulus Arithmetic

Two additional operators are available in MSX-BASIC:

Integer division is denoted by the "\" 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.

## 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 termimates.

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

## 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
```

# Logical Operators

Logical operators perform tests on relations, multiple 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. |

## 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".

## String Operators

Strings may 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.

# GRAPHICS

There are four different kinds of screen display mode in this computer, two in Text mode and another two in Graphic mode.

## Screen 0 mode

Screen 0 mode is one of the Text modes and can display 24 lines with up to 40 characters on each line.
These characters are formed in a 5×7 dot matrix which is in a 6×8 dot font.
So, there is a one dot space on either side of the character and line.
In this mode, all the graphic commands and statements cannot be used.
This mode is a default mode after reset.

## Screen 1 mode

Screen 1 mode is another text mode and can display 24 lines with up to 32 characters per line.
These characters are formed in a 8×8 dot font.
All the graphic commands and statements are not allowed except the sprite handling statements.
We may as well use this mode about graphic and code character symbols.

## Screen 2 mode

This high-resolution graphic mode can display colored dots on a matrix 256 dots in horizontal and 192 dots in vertical.
These are sixteen colors available in this mode.

## Screen 3 mode

This graphic mode is called multi-color mode.
It has 64×48 colored blocks which consist of 4×4 dots.
Each block can be any of sixteen colors.

## Color

These are sixteen colors in computer.
Each color is signified with color code.
Now, the color codes are:

0 : Transparent
1 : Black
2 : Green
3 : Light Green
4 : Dark Blue
5 : Light Blue
6 : Dark Red
7 : Cyan
8 : Red
9 : Light Red
10 : Yellow
11 : Light Yellow
12 : Dark Green
13 : Magenta
14 : Gray
15 : White

# Border, Background and Foreground Color

We may think of the screen's display as of the three layers, one on top of the other.

At the bottom there is the Border, above it there is the Back ground (which in the Screen 0 mode covers the Border totally; and in the graphic screen (1 or 2) "grows down" in size and "exposes" the Border at the top and bottom of the screen.)

Above the Background comes the Foreground which might be described as a clear acetate layer that "carries" all the images that appear on the screen on screen 0 or 1 it's the text, and on Screen 2 or 3 it's the graphic image.

# Circle

To begin exploring the graphics capability of the computer type in the following lines, pressing RETURN after each is completed:

```
10 SCREEN 2
20 CIRCLE (128,80), 60, 11
30 PAINT (128,80), 11
40 GOTO 40
```

Now, RUN the program and you will see the yellow circle appear on the screen and then it will be filled in by the computer's yellow paintbrush. To understand how this happens, let's look at each line individually.

```
10 SCREEN 2
```

This line causes the computer to display its graphics screen

```
20 CIRCLE (128,80), 60, 11
```

Here, you are telling the computer to draw a circle around a center point that is 128 columns from the left side of the screen, 80 rows down from the top of the screen, with a radius (distance from the center of the circle) of 60 points and using the yellow (the number 11) outline.

# Paint

```
30 PAINT (128,80), 11
```

This line introduces you to the PAINT command. This command tells the computer to use its "paint brush" to fill certain areas. In line 30 the computer is instructed to fill the circle you have just outlined in line 20. In order to paint (fill) an object you must give the computer the coordinates numbers (in parentheses) which designate any point inside the object (As we just did-giving the coordinates of the center point of our circle). If you had used coordinates which designate a point outside the object, the computer would have painted the whole background but not fill the object itself.

However, the fill color MUST be the same as the outline color in our case the number 11 is the same yellow color used for the circle outline from line 20. The PAINT "recognizes" outlines of objects as borders only if their color matches the PAINT color. A different PAINT color will "ignore" the outlines and will paint (fill) the whole screen-covering the object.

```
40 GOTO 40
```

The last line of this program causes the computer to repeat line 40 so the circle will not disappear. To stop the program press the CTRL-STOP key combination.

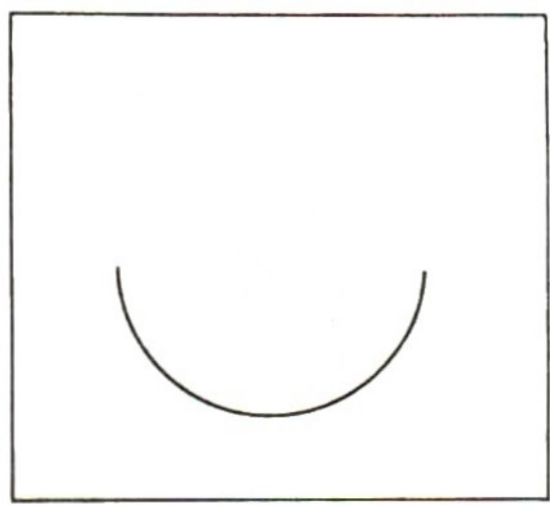
You can experiment with the numbers in this program to vary the location, size, or color of the circle being painted.

You can create a vast array of different sized circles and geometric shapes by adding a few more instructions to the CIRCLE command. We will give you another example of using the circle command and for additional hints, you should refer to the BASIC.

Change the program to read:

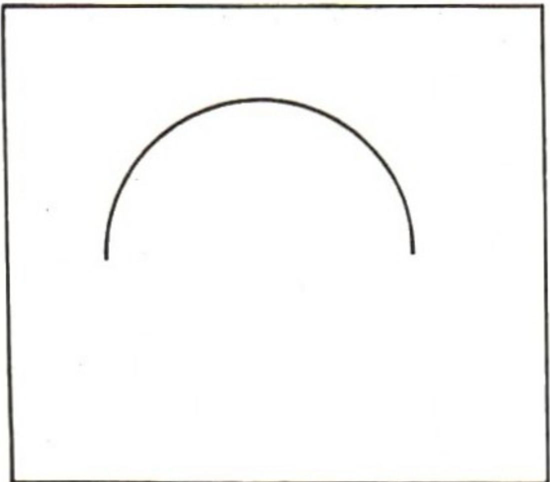
```
10 SCREEN 2
20 CIRCLE (128,96), 80, 1, 3.14, 6.28
30 GOTO 30
```

RUNning the program will give you the following result:



You should see the bottom half of the circle. Should you change line 20 to be:

```
20 CIRCLE (128,96), 80, 13, 6.28, 3.14
```



You will see that the top half of the circle is drawn. Another way of constructing a whole circle is with the following changes in line 20:

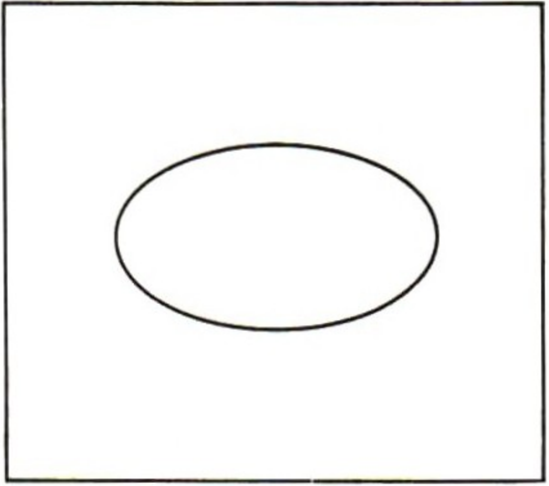
```
20 CIRCLE (128,96), 80, 13, 0, 6.28
```

Those of you who remember your geometry will recall that 3.14 is pi and 6.28 is 2pi (approximate). The two pi numbers which follow the color number (#13) on line 20 tell the computer where you would like the computer to begin and end the circle (how much of the cirlce you want drawn).

Those of you who are not adept at using variants of pi can just overlook this business and consult the BASIC.

You can also specify the kind of shape drawn. For example, you can draw an ellipse (a distorted circle for those of you unfamiliar with geometry) with the following added feature on line 20.

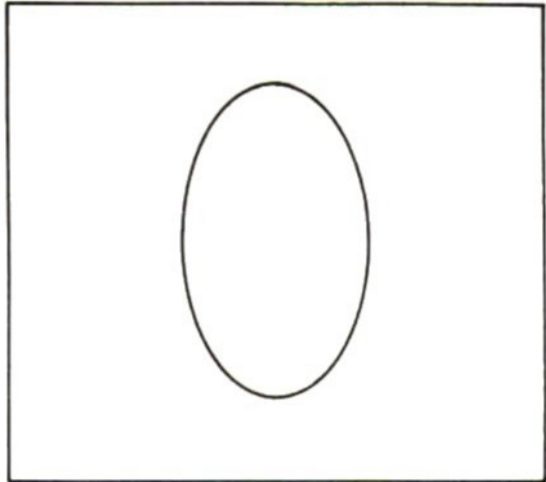
```
20 CIRCLE (128,96), 80, 13,,,1/4
```



How did we get an ellipse? Well, the three commas after the number 13 are necessary to inform the computer that we will not be specifying the starting and ending points of the shape and are therefore leaving them blank. The computer knows what to do when we leave it blank. It assumes that we want the whole shape drawn. The 1/4 at the end of line 20 tells the computer the height/-width ratio that we want.

Generally, the width of the circle is the same as the radius you specify. However, if the ratio number at the end of the CIRCLE command line is less than one (1), the circle will be wider than it is high, as in the example above where the ratio is "1/4". If the ratio is greater than one (1), the circle will be higher than it is wide, as in the following example:

```
20 CIRCLE (128,96), 80, 13,,,2
```



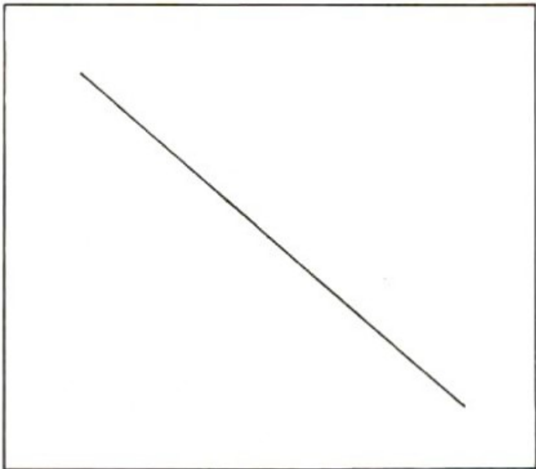
For further information on the CIRCLE command, consult the BASIC.

# Line and Box Drawing

## LINE

Now that you have seen what your computer can do with circles and its paintbrush, we'll take a look at lines and boxes. The computer has the same simple method for drawing them as it does for circles. First, type NEW to clear the memory of the program we were using before. Now, enter the following

```
10 SCREEN 2
20 LINE (50, 40)-(200, 150), 8
30 GOTO 30
```



When you run this program, you will see that a line has been drawn from high on the left side of the screen to a low point on the right side of the screen. The line that causes this to happen is line 20:

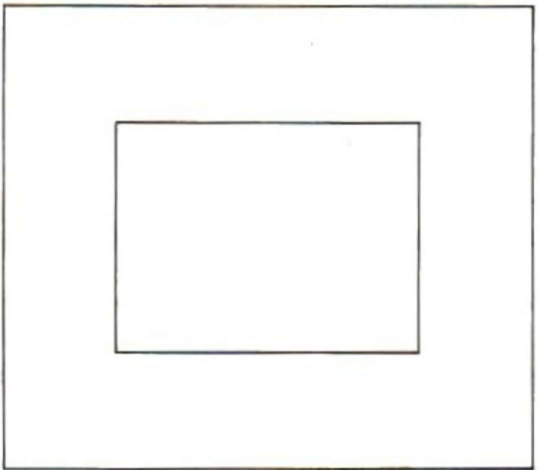
```
20 LINE (50, 40)–(200, 150), 8
```

This line tells the computer to draw a line from a position 50 points from the left margin on the screen and 40 points down from the top over to a position that is 200 points from the left margin and 150 points down from the top. The number 8 designates the color of the line to be red.

**BOX (B)**

By simply adding the letter B following a comma to the end of line 20 you will convert this line into a Box.

```
20 LINE (50, 40)-(200, 150), 8, B
```



RUNning the program now, will show a box (outlined rectangle) on the screen. The "B" tells the computer to draw the box at the same coordinates as the line.

**BOX FILL (BF)**

To tell the computer to use the paintbrush (fill the box) simply, add the letter "F" immediately to the right of the "B" in line 20.

```
20 LINE (50, 40)-(200, 150), 8,BF
```

Now, you will see that the program draws the same box and paints the inside with the same color as the outline.

## DRAW

The DRAW command is actually the door to an actual mini-language within BASIC called "Graphic Macro Language (GML)" start by clearing the computer's memory.

(type: NEW then press RETURN) Then type the following lines:

```
10 SCREEN 2
20 PSET (50, 60), 1
30 DRAW "D50 R50 U50 L50"
40 GOTO 40
```

Line 20 positions your graphic cursor at the X,Y coordinates (50, 60), and designates the color to be Black (,1)

Line 30 starts the line drawing at the point specified in the PSET command. It then moves relative to the point, according to the distance and directional commands specified in the DRAW statement.

Example: DRAW "U50 R50"

is: Draw fifty units UP then fifty units to the RIGHT.

The quotation marks around the instructions are in quotes because the DRAW command acts on a character string. Remember, a character string is a variable that holds characters. Therefore, we could have written the DRAW example used above in the following manner.

```
30 T$= "U50 R50 D50 L50"
40 DRAW T$
50 GOTO 50
```

This second way of DRAWing first defines the object we wish to DRAW, and then places it in T$ and then DRAWs T$.

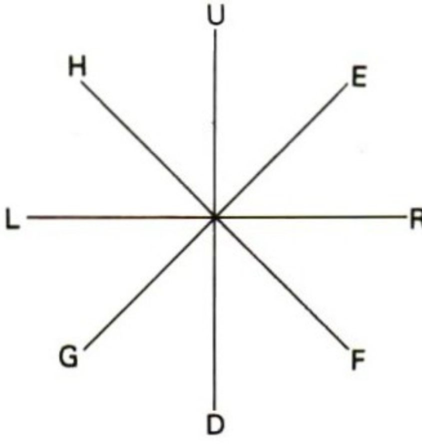
Please note you can draw diagonally using

E = diagonally up and right

F = diagonally down and right

G = diagonally down and left

H = diagonally up and left



# Sprites

Now that we have learned how to deal with the simpler shapes, we will examine the more complex type of graphics generation called SPRITE generation. The best way to understand a sprite is to imagine it as a magic genie you can create and easily control.

Unlike the graphic commands we have encountered up till now- which can only create one type of an object, like a line or circle– the manipulation of sprites is a lot more flexible.

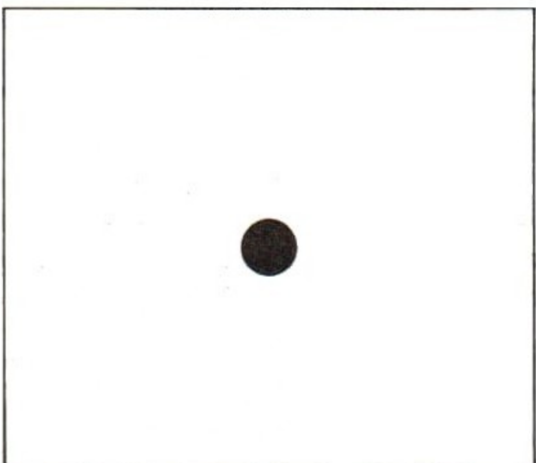
In order to see a sprite on the screen you must do the following:

STEP 1: Pick one genie "to talk to." (There are 32 different genies available to do your bidding).
STEP 2: You must tell the genie "what you want it to wear." (In other words, what shape you want it to assume).
STEP 3: You must tell it what color to make the shape that it will wear.
STEP 4: You must tell it where to appear on the screen.

The importance of this genie metaphor cannot be overstated. Whenever you do not get the results you expected when commanding a genie it is probably because you did not provide all four pieces of information necessary to make the genie appear.

We will now demonstrate how to instruct a genie. Enter the following program and RUN it.

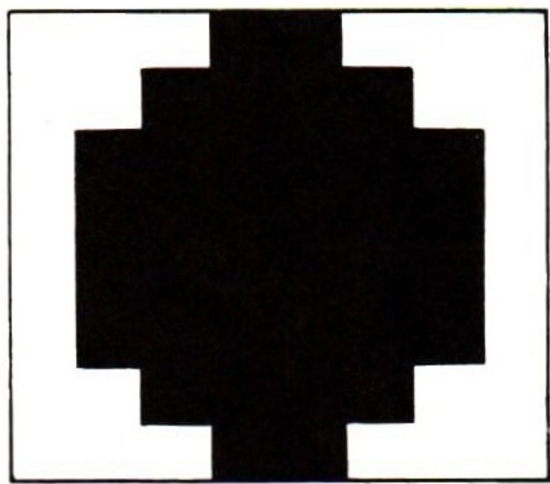
```
 10 SCREEN 2
 20 FOR T = 1 TO 8
 30 READ A$
 40 S$ =S$ + CHR$ (VAL ("&B" + A$) )
 50 NEXT T
 60 SPRITE$ (1) = S$
 70 PUT SPRITE 0, (128, 96), 8, 1
 80 GOTO 80
100 DATA 00011000
110 DATA 00111100
120 DATA 01111110
130 DATA 01111110
140 DATA 01111110
150 DATA 01111110
160 DATA 00111100
170 DATA 00011000
```



You should see a red ball appear at the center of the screen. This ball is the Sprite that we created in the above program. Here is how it works.

Lines 100-170: These lines design the clothes that the genie will wear (or in straighter language, they contain the design of the sprite's shape). Each line of the data statement has eight characters on it. They represent the size of the sprite. The zeros are to make the display transparent at that point of the shape while the ones are the points of the display that are lit.

If we took a grid 8 by 8 boxes, the shape would look like this:



Lines 20-50: These lines complete the design of the shape of the clothes. They set up a loop that will read the set data lines, convert them into binary strings, append each one to the one that follows and then store this one shape unit in S$.

Line 60: This line picks the sprite that we will command (#1) and tells it to carry the shape contained in S$.

Line 70: This line tells the sprite what color to make the shape, and where to appear on the screen. This line:

```
70 PUT SPRITE 0, (128, 96), 8, 1
```

is read like this (It's a long sentence, but you'll be able to follow it):

Put the SPRITE that is specified at the end of line, which is #1, on plane (surface) 0, at position (128, 96) which is the center of the screen, using color #8. The sprite to use is #1.

(Note: In the command, PUT SPRITE (sprite plane), (X,Y), (Color #), (Sprite pattern #) the use of different plane numbers allows the user to place more than one sprite on the screen at once.)

The way we create sprites is very logical. If you are familiar with any other computer's BASIC, you will immediately notice how much easier the sprite manipulation is on the computer. That's because other systems force you to go PEEKING and POKING around in their computer's memory.

Sprites are not limited to only 8 by 8 pixels. Sprites can also be placed within a 16 by 16 box.

When SCREEN sizes 0 to 1 are selected (screen 1,1), the sprite size is limited to 8 by 8; however, if sprite size 2 is selected, then the use of a 16 by 16 box is allowed. But, the computer fills the 16 by 16 box differently than it fills the 8 by 8 box. The following program should illustrate how this works:

```
 10 screen 1,2
 20 for x = 1 to 32
 30 read a$
 40 restore
 50 s$ = s$ + chr$ (val ("& b" + a$) )
 60 sprite$ = s$
 70 put sprite 0, (128, 96), 1, 0
 80 next x
 90 goto 90
100 data 11111111
```

Notice that the computer first fills a box 8 by 16, and then fills another 8 by 16 box alongside this one to make 16 by 16.

Therefore, when making a16 by 16 sprite, careful manipulation of data statements (32 of them) is required.

# SOUND

There is also a very powerful music synthesizer built-in to the computer that is easily used by simple BASIC commands to produce music. In addition to the power of this synthesizer, it is most important to realize that it can do its work independently of the main microprocessor. What this means is that you can program the synthesizer to do one thing while the screen, printer, modem or other peripheral is doing something else.

The following figure represents the available musical scale that can be accessed by the sound synthesizer.

# PLAY

The command that opens the door to this synthesizer is the BASIC keyword PLAY. For example, Typing the BASIC statement:

```
PLAY "CEG"
```

followed by pressing the return key will produce musical tones of do-mi-sol from your computer through the speaker on your television or monitor. You could achieve the same results by writing a BASIC program with the following lines:

```
10 PLAY "CEG"
20 END
```

The "PLAY" command has the some Music Macro Languages.
This Music Macro Languages help the "PLAY" command to generate the various Music sound easily. The Music Macro Languages are:

## "O" (OCTAVE)

First, change line 10 to read:

```
10 PLAY "o1CEG"
```

Now, when you run the program, you will hear that the sounds produced are at a very low pitch when compared with the first ones you made. This is because you have set the OCTAVE by adding the "o1" before the "CEG". This is the command that allows you to access 8 octaves with the synthesizer. Now add this line:

```
15 PLAY "o4CEG"
```

When the program is run, you will hear three low notes (i.e., line 10) followed bv three higher notes (i.e., line 15). The octaves you can access using the "o" command can range from 1 (lowest) to 8 (highest).

## "T" TEMPO

Now, change line 10 to read:

```
10 PLAY "T32o1CEG"
```

The program will now play the same note you heard before but at a much slower rate. What you did by typing the "T32" before the "o1CEG" was to set the TEMPO or speed of the music. The values for "T" can range from 32 (slowest) to 255 (fastest).

You will also notice that the notes in line 15 also play at the slower pace. This is because the synthesizer will play at whatever tempo you set until you tell it to play at a different tempo. To see this in action, change line 15 to read:

```
15 PALY "T255o4CEG"
```

Now, as you can hear, the notes from line 15 play at a much faster pace than those in line 10.

## "L" (LENGTH)

You can also control the length of each note individually. To see this, change line 10 to read:

```
10 PLAY "T255o1CEL1G"
```

This changes the "G" note to a much longer duration than "C" or "E" and also causes the notes in line 15 to play for a longer time. This length command can be placed in front of any note to control the length of the note. The lengths of the notes can be varied from 1 (longest) to 255 (shortest).

Two other BASIC commands that can be applied to sound are the "S" command and the "M" command. These two commands determine the tonal qualities of the note being played. These are more commonly referred to as the "ENVELOPE" characteristics of a note.
Everything that creates a sound has unique characteristics.

For example: the same note played on a piano and a trumpet may be at the same pitch but will have two distinctly different sounds. These two commands allow you to shape the notes you are creating in the same way.

## "S" (SHAPE)

The "S" command controls the shape of the note. As an illustration of this, change line 10 to read:

```
10 PLAY "S1o4CEG"
```

and eliminate line 15.

Now, run the program to see the differences in the sounds you hear. These shape commands can be considered the voices of the synthesizer. There are 14 of them built in to the computer. This means that the numbers used to set the "S" command can range from 1 to 14.

## "M" (TONE)

The "M" command controls the tone period or to be more specific, the amount of time that you will hear each note based on its tonal qualities. To see how this works, change line 10 to read:

```
10 PLAY "S10M5000o4CEG"
```

As you will hear, this changes the sound dramatically. The values used to set "M" can range from 1 to 65535.

## "R" (REST)

You can also insert pauses between notes by using the "R" command. Change line 10 to read:

```
10 PLAY "o4CR1ER10G"
```

This causes the "C" note to play and then silence is heard for a while then the "E" note plays, then a shorter period of silence, then the "G" note followed immediately by the "C" note again.

## "V" (VOLUME)

The final command we will examine in this section is the "V" command. This command is used to set the volume of the sound being produced. Change line 10 to read:

```
10 PLAY "o4V5CV1oEV15G"
```

You will now hear that each note gets louder than the one before it. You can set the volume from 0 to 15.

## Using a channels of SOUND

So far, we have only used one note at a time to demonstrate the use of the synthesizer. However, the computer has three separate channels of sound that can be programmed individually to play together to create chords.
Change line 10 to read:

```
10 PLAY "o1CEG", "o3EFC", "o5GAB"
```

What you hear now is three notes being played in combination to create a chord. You can also have each channel play something entirely different from the others to create a melody and harmony part in the music you create.

Other than the PLAY statement which allows you to create musical notes, you can use the SOUND statement to directly cotrol the various capabilities of the Programmable Sound Generator which we will refer to as the PSG.

A PSG SOUND statement take the form of:

SOUND < register of psg >, < value >

Where < register of PSG > is one of the 13 available registers the PSG uses, and < value > is a number between 1 to 255 The function of creating a specific sound or sound effect logically follows the control sequence listed below:

| OPERATIONS | REGISTERS | FUNCTION |
|---|---|---|
| Tone generator control | R0-R5 | Programmable tone periods. |
| Noise generator control | R6 | Programmable noise period. |
| Mixer control | R7 | Enable tone and or noise on selected channels |
| Amplitude control | R8-R10 | Select "fixed" or "envelope variable" amplitudes. |
| Envelope generator control | R11-R13 | Programmable envelope period and select envelope pattern. |

# Tone generator control

The PSG has 3 tone channels A,B and C. The frequency for each channel is obtained by counting down the input clock by 16 times the value of the frequency wanted.

For example:

Desired Value = 1789773/(16*frequency)
Low register = Remainder of < Desired Value > /256
High register = Integer of < Desired Value > /256

The high and low registers correspond to the register pair used by each control.

| Channel | High Register | Low Register |
|---|---|---|
| A | 1 | 0 |
| B | 3 | 2 |
| C | 5 | 4 |

Program example follows:

```
10 Input "frequency" ; A
20 F = 1789773/(16*A)
30 H = F/256
40 L = F Mod 256
50 SOUND 0, L
60 SOUND 1, H
70 SOUND 8, 15
80 SOUND 7, 254
90 END
```

## Amplitude control

In the previous example program, you should notice we used register 8 to enable to volume of channel A. The PSG has three separate register to control the amplitude of the different channels.

| Channel | Register |
|---|---|
| A | 8 |
| B | 9 |
| C | 10 |

Each channel can have a volume from 9 to 15 with 15 being the loudest.

The Amplitude control register can also be used to direct the envelope period of each channel, by setting the Amplitude channel to a value of 16, the amplitude of the corresponding channel would be controlled by reg 11, 12, and 13. For more information on this, refer to Envelope Period Control Registers.

## Mixer control

The MIXER register, register number 7, controls the three Noise / Tone channels. The Mixers, as previously described, combine the noise and tone frequencies for each of the three channels. The determination of combining neither, either or both noise and tone frequencies on each channel is made by the state of Bit 0-Bit 5 of reg. #7. Bit 6 and 7 are for I/O ports connected through the PSG, and these are ignored by BASIC.

## Register 7

| B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
|---|---|---|---|---|---|---|---|
| Not used | | Noise channel | | | Tone channel | | |
| / / / / / / / | | C | B | A | C | B | A |

Bits logical value
1 if channel is disabled
0 if channel is enabled

For example:

SOUND 7, &B 11111110

will turn on tone channel A.

SOUND 7, & B 11110110

will enable both noise and tone channel A.

## Envelope period control register

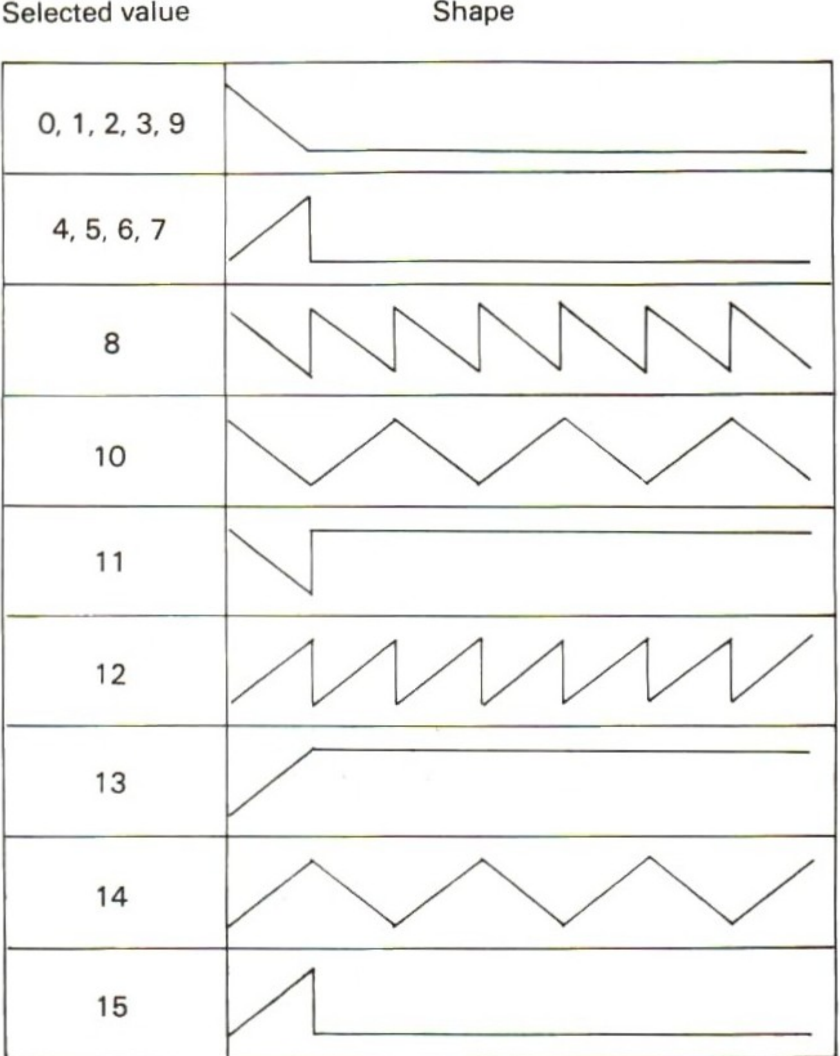
The generation of fairly complex envelope patterns can be accomplished two different ways in BASIC. First, it is possible to vary the frequency of the envelope using register 11 and 12 as a 16 bit register; and second, the relative shape and cycle of the envelope can be varied by using register 13.

For example:

< Desired envelope freq > = 1789773/ (256 * freq)

# Shape register

You can select 9 different shapes for the envelope period output, by programming the shape register 13.




For example:

SOUND 13, 14

will create a tone modulating up and down according to the envelope period set in register 11 and 12, when the enable bit 4 of register 8 (SOUND 8, 16) is set.

# PSG block diagram

| REGISTER | BIT | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
|---|---|---|---|---|---|---|---|---|---|
| R0 | Channel A Tone Period | 8 BIT Fine Tune A | | | | | | | |
| R1 | | | | | | 4 BIT Coarse Tune A | | | |
| R2 | Channel B Tone Period | 8 BIT Fine Tune B | | | | | | | |
| R3 | | | | | | 4 BIT Coarse Tune B | | | |
| R4 | Channel C Tone Period | 8 BIT Fine Tune C | | | | | | | |
| R5 | | | | | | 4 BIT Coarse Tune C | | | |
| R6 | Noise Period | | | | 5 BIT Period Control | | | | |
| R7 | $\overline{\text{Enable}}$ | $\overline{\text{IN}}$/OUT | | $\overline{\text{Noise}}$ | | | $\overline{\text{Tone}}$ | | |
| | | 10B | 10A | C | B | A | C | B | A |
| R8 | Channel A Amplitude | | | | M | L3 | L2 | L1 | L0 |
| R9 | Channel B Amplitude | | | | M | L3 | L2 | L1 | L0 |
| R10 | Channel C Amplitude | | | | M | L3 | L2 | L1 | L0 |
| R11 | Envelope Period | 8 BIT Fine Tune E | | | | | | | |
| R12 | | 8 BIT Coarse Tune E | | | | | | | |
| R13 | Envelope Shape/Cycle | | | | | CONT. | ATT. | ALT. | HOLD |

REGISTERED ARRAY
(14 READ/WRITE CONTROL REGISTERS)

# 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.

## Notation

The following notation is used throughout this book in the descriptions of computer prompts and your response:

- When inputting, capital and small letters do not have to be differentiated except for those inside quotes (") (file names e.t.c), which must be differentiated.
- The categories inside angle brackets "<" ">" are decided by the user.
- The categories inside square brackets "[" "]" can be omitted or anyone can be chosen.
- When omitting parameters seperated by comma (,), the procedure is represented as follows.

  EX) CLEAR [ <SIZE OF STRING AREA> ] [, <STATUS OF MEMORY> ]

  In the above case, the parameters enclosed in large parameters can be omitted.
- All commas (,), semi colons (;), parenthesis ( ( ) ), slash marks and equal signs must be entered exactly.
- A "+" is used to indicate that the two keys are to be pressed simultaneously.

Thus CTRL + STOP means: hold down the CTRL and press STOP.

## ABS (X)

Returns the absolute value of the expression X, without its sign (+ or −). The answer is always positive double precision number.

Ex)
```
10 FOR I =-2 TO 2
20 PRINT "ABSOLUTE VALUE OF";
I; "IS"; ABS (I)
30 NEXT I
40 END
RUN
ABSOLUTE VALUE OF -2 IS 2
ABSOLUTE VALUE OF -1 IS 1
ABSOLUTE VALUE OF  0 IS 0
ABSOLUTE VALUE OF  1 IS 1
ABSOLUTE VALUE OF  2 IS 2
```

## 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.

Ex)
```
10 A$ = "ABCdef123@"
20 FOR I = 1 TO 10
30 B$ = MID$ (A$, I, 1)
40 PRINT "ASCII CODE OF "; B$;"
   IS" ;ASC (B$)
50 NEXT : END
RUN
ASCII CODE OF A IS  65
ASCII CODE OF B IS  66
ASCII CODE OF C IS  67
ASCII CODE OF d IS  100
ASCII CODE OF e IS  101
ASCII CODE OF f IS  102
ASCII CODE OF 1 IS  49
ASCII CODE OF 2 IS  50
ASCII CODE OF 3 IS  51
ASCII CODE OF @IS  64
```

※ ASCII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE)

# 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.

Ex)
```
10 P = 3.1415926535#
20 PRINT "X (DEG.)     ATN (X)"
30 FOR I = 0 TO P STEP P/5
40 PRINT USING"# # # #          "; INT(I/P*180);
50 PRINT ATN(I)
60 NEXT I: END
RUN
X (DEG.)          ATN (X)
  0               0
 36               .56098211609574
 72               .89863709304242
108               1.0830346193243
144               1.1921125187293
180               1.2626272556706
```

# AUTO

**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.

Ex) AUTO 20
20
30
[CTRL] [STOP]
AUTO, 5
0
5
10

# BASE (<n>)

Current base address in decimal number for each table of VDP (Video Display Processor). 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.

Ex)
```
10 FOR I = 5 TO 8
20 PRINT "BASE (";I;")"; TAB (12)
;
30 PRINT RIGHT$ (
"OOO" + HEX$ (BA
 SE (I)), 4)
 40 NEXT I
 50 END
 RUN
 BASE ( 5 )      1800
 BASE ( 6 )      2000
 BASE ( 7 )      0000
 BASE ( 8 )      1B00
```

# BEEP

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

Ex)
```
10 FOR I = 0 TO 5
20 BEEP
30 NEXT I
40 FOR I = 0 TO 100 : NEXT I
50 FOR I = 0 TO 5
60 PRINT CHR$ (7)
70 NEXT I
RUN
OK
```

# BIN$ (n)

Returns a string which represents the binary value of the decimal argument.
n is a numeric expression in the range -32768 to 32767. If n is negative, the two's complement form is used. That is, BIN$ (-n) is the same as BIN$ (65536 -n).
The answer becomes to be zero suppressed number.

Ex)
```
10 A = 123 : B = 234
20 PRINT "A = "; BIN$ (A);
30 PRINT "B = "; BIN$ (B)
40 PRINT "A AND B = "; BIN$ (A AND B)
50 PRINT "A OR B = "; BIN$ (A OR B)
60 PRINT "A XOR B = "; BIN$ (A XOR B)
70 PRINT "A EQV B = "; BIN$ (A EQV B)
80 PRINT "A IMP B = "; BIN$ (A IMP B)
90 END
run
A = 1111011 B = 11101010
A AND B = 1101010
A OR B = 11111011
A XOR B = 10010001
A EQV B = 1111111101101110
A IMP B = 1111111111101110
```

# BLOAD

**BLOAD "< device > : [ <file name>]" [,R], [, < offset >]**

To load a machine language program from the specified device. If option is specified, after the loading, program begins execution automatically from the address which is specified as 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.

Ex) BLOAD "CAS: LPN", R, 5
FOUND: LPN
OK

# BSAVE

**BSAVE "< device > [: < file name > ]", < top address >, < end address > [, <execution address>]**

To save a memory image at the specified memory location to the device.

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

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

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

# CALL

**CALL <statement> [(argument, argument, ......)]**

To invoke an expanded statement supplied by ROM cartridge.

'_' is an abbreviation for 'CALL'.

Ex) CALL TALK ("A", "B", "C")
_TALK ("A", "B", "C")

## CDBL (X)

Converts X to a double precision number.

But precision of the answer is same as before conversion.

Ex)
```
10 B ! = 2.333333#
20 A# = CDBL (B !/1.7)
30 PRINT A#
run
 1.3725470588235
OK
```

## CHR$ (X)

Returns a string character whose ASCII code is X. CHR$ is commonly used to send a special character to the console, etc. This is the opposite of ASC (X$).
If X is control code, the character correspond to that code is not displayed but the function of that code is executed.
If X isn't the range 0 to 255, an Illegal function call error occurs.

See Appendix B about the CHARACTER CODE TABLE.

Ex)
```
10 PRINT "chr$ (32) – chr$ (37)"
20 PRINT
30 FOR I = 32 TO 37
40 PRING CHR$ (I)
50 NEXT
RUN
chr$ (32) - chr$ (37)

!
"
#
$
%
OK
```

# CINT (X)

Converts X to a integer number by truncating the fractional portion.
If X isn't in the range −32768 to 32767, an 'Overflow' error occurs.

Ex)

```
10 B# = 1.2345678912345#
20 A% = CINT (B # * 3)
30 PRINT A%,
40 C% = CINT (2.3333#)
50 PRINT C%
run
 3             2
OK
```

# CIRCLE

**CIRCLE <(X,Y) or STEP (X,Y)>, <radius> [,<color>] [,<start angle> [,<end angle>] [,<ratio>]**

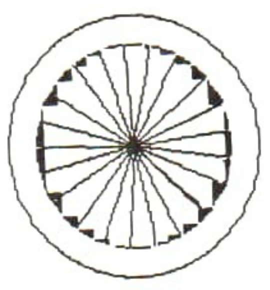
To draw an ellipse with a center and radius as indicated by the first of its arguments.
(X,Y) or STEP (X,Y) specifies the coordinate of the center of the circle on the screen.

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

Ex)
```
10 COLOR 10, 15
20 SCREEN 2
30 CIRCLE (128, 96), 60, 10,,, 1.15
40 PAINT (128, 96), 10
50 FOR I = -6.28# TO -0 STEP .3
60 CIRCLE (128, 96), 50, 15, I, I +. 3, 1.15
70 NEXT
90 COLOR 15, 4, 7
100 GOTO 100
run
```



# CLEAR

**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.

Ex)
```
10 A = 10: B$ = "TEST"
20 PRINT A, B$
30 CLEAR
40 PRINT A, B$
50 END
RUN
 10          TEST
 0
OK
```

# CLOAD

**CLOAD ["<file name>"]**

To load a BASIC program file from the cassette.

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.

Ex)
```
CLOAD "SAMPLE"
FOUND : SAMPLE
OK
```

# CLOAD?

**CLOAD? ["<file name>"]**

To verify a BASIC program on cassette with one in memory. CLOAD? is generally used right after CSAVE command to confirm tht the program in memory has stored to tape surely.

EX)
```
CLOAD? "SAMPLE"
FOUND: SAMPEL
OK
```

# CLOSE

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

To close the channel and release the buffer associated with it. If no <file number> 's are specified, all open channels are closed.

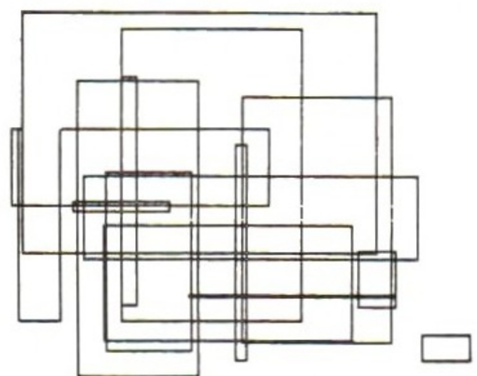
Ex)
```
10 SCREEN 2
20 LINE (50, 50)-(100, 100), 10, B
30 OPEN "GRP:" FOR OUTPUT AS #
1
40 PRESET (75, 35)
50 PRINT # 1, "BOX"
60 CLOSE
70 GOTO 70
RUN
```

BOX

# CLS

To clear the screen. Valid in all screen modes.

Ex)
```
10 SCREEN 2: CLS
20 FOR I = 0 TO 15
30 X1 = INT (RND (1)*226)
40 Y1 = INT (RND (1)*192)
50 X2 = INT (RND (1)*226)
60 Y2 = INT (RND (1)*192)
70 COLOR I
80 LINE (X1, Y1)-(X2, Y2),, B
90 NEXT I
100 FOR I = 0 TO 200 : NEXT I
110 CLS : GOTO 20
RUN
```



# COLOR

**COLOR [<foreground color>] [, <background color>] [, <border color>]**

To define the color screen. The argument can be in the range of 0 to 15. Actual color corresponding to each value is as follows.

Omitting all three color codes is not allowed.

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

Ex)
```
10 CLS
20 FOR I = 0 TO 15
30 FOR J = 0 TO 15
40 PRINT "COLOR"; I; ",";J
50 COLOR I, J
60 FOR K = 0 TO 300 : NEXT K
70 NEXT J, I
80 FOR I = 0 TO 15
90 COLOR,, I
100 FOR K = 0 TO 300 : NEXT K
110 NEXT I
120 COLOR 15, 4, 7
```

## CONT

To continue program execution after BREAK or STOP in execution.

Ex)
```
10 PRINT "    *** TEST ***"
20 STOP
30 PRINT "  PROGRAM"
RUN
    *** TEST ***
BREAK IN 20
OK
CONT
    PROGRAM
```

## COS (X)

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

Ex)
```
10 P = 3.1415926536#
20 FOR I = 0 TO P STEP P/6
25 C ! = COS (I)
30 PRINT "  COS"; INT (I/P*180);
 TAB (9); " = "; C!
40 NEXT I : END
run
  COS 0   = 1
  COS 29  = .866025
  COS 59  = .5
  COS 90  =-5.15221E-12
  COS 120 =-.5
  COS 150 =-.866025
  COS 180 =-1
OK
```

# CSAVE

**CSAVE "<file name>" [, <baud rate>]**

To save a BASIC program file to the cassette tape.

BASIC saves the file in a compressed binary 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 CSAVE command.

<File name> can not be omitted and can be determind up to 6 characters.

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

Ex)
```
CSAVE "SAMPLE"
OK
```

# CSNG (X)

Converts X to a single precision number.

Ex)
```
10 I = COS (3.14/4)
20 A = CSNG (I)
30 PRINT I
40 PRINT A
50 END
run
 .70738826916719
 .707388
OK
```

# CSRLIN

Returns the current vertical coordinate of the cursor.

About the horizontal coordinate of the cursor, see to POS statement.

About the horizontal and vertical coordinate of the cursor, see to LOCATE statement.

Ex) print csrlin → first line
1
OK
print csrlin → fourth line
4
OK
print csrlin → seventh line
7
OK

# DATA

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

To store the numeric and string constants that are accessed by 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 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.

Ex)

```
10 FOR I = 0 TO 4
20 READ A$
30 PRINT TAB (5);A$
40 NEXT I
50 READ A, B, C, D
55 PRINT A, B, C, D
60 DATA I, WANT TO
70 DATA HAVE
80 DATA "YOU, COMPUTER"
90 DATA FROM  AVT..!
100 DATA 10, &h10, &O10, &B10
110 END
run
     I
     WANT TO
     HAVE
     YOU, COMPUTER
     FROM  AVT ..!
 10               16
 8                2
OK
```

# DEF FN

**DEF FN <name> [<parameter>, <parameter>...] = <function definition>**

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

< name > must be a legal variable name. This name, proceded by FN, becomes the name of the function. Comprised of the 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 DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an 'Undefined user function' error occurs. DEF FN is illegal in the direct mode.

Ex)
```
10 DEF FNLN (X) = LOG (X)/LOG (10)
20 PRINT "  X     LN(X)     LOG(X)"
30 FOR I = .4 TO 1.4 STEP .2
40 PRINT USING " #.#    ##.#####    ##.#####";I;LOG(I):FNLN(I)
50 NEXT I:END
run
  X       LN (X)      LOG (X)
 0.4    -0.91629    -0.39794
 0 6    -0.51083    -0.22185
 0.8    -0.22314    -0.09691
 1.0     0.00000     0.00000
 1.2     0.18232     0.07918
 1.4     0.33647     0.14613
OK
```

# DEFINT/DEFSNG/DEFDBL/DEFSTR

**DEFINT/DEFSNG/DEFDBL/DEFSTR**

**< ranges of characters >**

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 character(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.

Ex)
```
10 DEFINT I
20 DEFSNG J
30 DEFDBL K
40 DEFSTR L
50 I = 1.6:PRINT I
60 J=I/3:PRINT J
70 K = 1/3:PRINT K
80 L = "ABC":PRINT L
90 CLEAR
100 PRINT I;J;K;L:END
run
 1
 .333333
 .33333333333333
ABC
 0  0  0  0
OK
```

# DEFUSR

**DEFUSR [< digit >]= <start address>**

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, DEFUSR 0 is assumed. The value of < start address> 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.

Ex)
```
10 CLEAR 200, &HDFFF
20 DEFINT A-Z
30AD= &HE000:DEFUSR=AD
40 FOR I=0 TO 3
50 READ A$
60 POKE AD+I,VAL("&h"+A$)
70 NEXT I
80 DATA 23,23,34,C9
90 INPUT "b=";B
100 A=USR (B)
110 PRINT"b + 1 = ";A
120 END
run
b = ? 66
b+1 = 67
OK
```

# DELETE

**DELETE [< start line number>] [-<end 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.

Ex)

```
10 A = 1
20 B = 2
30 C = A+B
40 D = A*B
50 E = A/B
60 F = A-B
70 PRINT A,B,C,D,E,F
80 END
DELETE 30-60
OK
list
10 A=1
20 B=2
70 PRINT A,B,C,D,E,F
80 END
OK
```

# DIM

**DIM < variable name >(maximum values of subscript,......)**

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.

Ex)
```
10 DIMA$ (5,3)
20 FOR I = 0 TO 4
30 FOR J = 0 TO 2
40 READ A$ (I,J)
50 PRINT A$(I,J);
55 NEXT J,I
60 DATA 1, 3, 5, 2, q, w, e, r, t, y
70 DATA !, @, #, $, %, &, ', (, ), ~
run
1352qwerty ! @ # $ %
```

# DRAW

**DRAW < string expression >**

To draw figure according to the graphic macro language.

The graphic macro language commands are contained in the string expression. 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.

Un ; Moves up
Dn ; Moves down
Ln ; Moves lift
Rn ; Moves right
En ; Moves diagonally up and right
Fn ; Moves diagonally down and right
Gn ; Moves diagonally down and left
Hn ; 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).

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

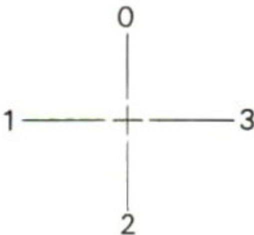
The aspect ratio of the screen is 1. So 8 horizontal points are equal in length to 8 vertical points.

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:

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




Cn ; Sets color n. n may range 0 to 15.

Sn ; 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 U, which means 'no-scaling' (i.e., same as S4)

X <string variable>;

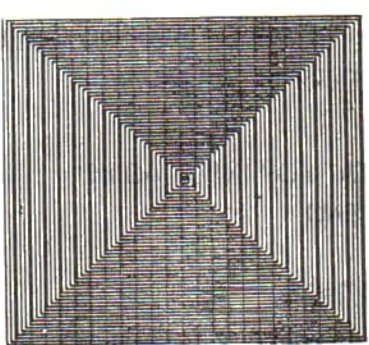
; Executes substring. This allows you to execute a second string from within a string
Example A$="U80R80D80L80":DRAW "XA$;"

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. For 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.

Ex)
```
10 SCREEN 2
20 PSET (220, 191), 10
30 DRAW "U190"
40 FOR I = 189 TO 1 STEP -4
50 A$ = "L" + STR$(I)+"D"+STR$(I-1)+"R"+STR$(I-2)+ "U"+STR$(I-3)
60 DRAW "XA$;"
70 NEXT I
80 GOTO 80
RUN
```



# 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 but in this case can not close files.

Ex)
```
10 PRINT "  *** PROGRAM IS ENDED ***"
20 END
30 PRINT "  NOT ANY MORE..."
RUN
   ***  PROGRAM IS ENDED ***
OK
```

# EOF (< n >)

Return -1 (true) if the end of a sequential file has been reached.
Otherwise, returns 0. <n> is the file number. Use EOF to test for end-or-file while INPUTing, to avoid 'Input past end' errors.

Ex)
```
10 OPEN "CAS:DATA" FOR INPUT AS #1
20 IF EOF (1) THEN END
30 INPUT # 1, N, R
40 PRINT "SQR(";N;")=";R
50 GOTO 20
```

# ERASE

**ERASE < variable name >, < variable name >....**

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.

Ex)
```
10 DIM A(5)
20 FOR I = 0 TO 5
30 READ A(I):PRINT A(I);
40 NEXT I:PRINT
50 ERASE A
60 FOR I = 0 TO 5
70 PRINT A(I);
80 NEXT I:END
90 DATA 1, 2, 3, 4, 5, 6
RUN
  1   2   3   4   5   6
  0   0   0   0   0   0
 OK
```

# 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.

```
Ex)  10 'GOTO BY ERROR
     20 ON ERROR GOTO 80
     30 INPUT "* IF NOT 1^999";A
     40 PRINT A
     50 IF A < 1 OR A > 1000 THEN ERROR 26
     60 GOTO 30
     70 'ERROR-GOTO
     80 IF ERR <> 26 THEN 120
     90 PRINT " * UNPRINTABLE ERROR"
     100 PRINT " * IN -"; ERL
     110 RESUME NEXT
     120 ON ERROR GOTO 0
     130 END
     run
     * IF NOT 1^999? 500
        500
     * IF NOT 1^999? 2000
        2000
      * UNPRINTABLE ERROR
      * IN - 50
     * IF NOT 1^999?-3
     -3
      * UNPRINTABLE ERROR
      * IN - 50
     * IF NOT 1^999?
```

# ERROR < error code >

To simulate the occurrence of an error or to allow error codes to be defined by the user.

The value of < error code > must be greater than 0 and less than 255. If the value of < error code > 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 G 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.

Ex)

```
10 INPUT "ERROR NO. =";A
20 ERROR A
30 END
run
 ERROR NO. =? 2
 Syntax error in 20
OK
run
 ERROR No. =? 10
 Redimensioned array in 20
OK
```

# EXP (X)

Returns 'e' to the power of X. 'e' means the base of natural logarism. X must be < = 145.06286085862. If EXP overflows, the 'Overflow' error message is printed.

Ex)
```
10 PRINT "  X    EXP(X)    LOG(EXP(X))"
20 FOR I = .4 TO 1.4 STEP .2
30 PRINT USING "#.# ###.###"; I; EXP (I);
40 PRINT LOG(EXP(I))
50 NEXT I: END
run
 X     EXP(X)   LOG (EXP(X))
0.4    1.492    .39999999999985
0.6    1.822    .59999999999982
0.8    2.226    .79999999999987
1.0    2.718    .99999999999986
1.2    3.320    1.1999999999999
1.4    4.055    1.4
OK
```

# 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.

Ex)
```
10 FOR A =-1.2345 TO 1.2345 STEP .5
20 PRINT USING "##.####"; A;
30 B = FIX (A)
40 PRINT USING "##"; B;
50 B = INT(A)
60 PRINT USING "##";B
70 NEXT A:END
run
-1.2345-1-2
-0.7345 0-1
-0.2345 0-1
 0.2655 0 0
 0.7655 0 0
OK
```

# FOR ~ NEXT

**FOR <variable >= X TO Y [STEP < Z >]**
**~**
**NEXT [ [< variable>] [, <variable >] ]**

< variable > can be integer, single-precision or double-precision.

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.

Ex)
```
10 FOR I = 0 TO 100 STEP 10
20 PRINT I;
30 NEXT I:PRINT
40 FOR I = 0 TO 10 STEP 5
50 FOR J = I TO I+4
60 PRINT USING "###"; J;
70 NEXT J:PRINT
80 NEXT I:END
run
 0    10    20   40    50    60    70    80
            90   100
  0    1     2    3    4
  5    6     7    8    9
 10   11    12   13   14
OK
```

# FRE (X)/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. For more details, see to Appendix D.

Ex)
```
20 PRINT " * FREE BYTES ARE"; FRE(0); "AT NOW"
30 FOR I = 200 TO 1000 STEP 200
40 DIM A(I)
50 PRINT " * FREE BYTES ARE"; FRE(0); "AT DIM A(";I;")"
60 ERASE A
70 NEXT I:END
run
 * FREE BYTES ARE 28679 AT NOW
 * FREE BYTES ARE 27027 AT DIM A(200)
 * FREE BYTES ARE 25427 AT DIM A(400)
 * FREE BYTES ARE 23827 AT DIM A(600)
 * FREE BYTES ARE 22227 AT DIM A(800)
 * FREE BYTES ARE 20627 AT DIM A(1000)
OK
```

# GOSUB

**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.

Ex)
```
10 GOSUB 50
20 GOSUB 50
30 I = 1:GOSUB 50
40 I = 2:GOSUB 50
50 'subroutine
60 PRINT "subroutine";
70 PRINT "I = ";I
80 IF I = 2 THEN RETURN 100
90 RETURN
100 END
run
subroutine I =  0
subroutine I =  0
subroutine I =  1
subroutine I =  2
OK
```

# GOTO

**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>. Only one space is allowed between GO and TO.

Ex)
```
10 GOTO 30
20 PRINT "YOU";: GOTO 70
30 PRINT "I";
40 PRINT "LOVE";: GOTO 20
50 PRINT "AVT MSX";
60 PRINT "COMPUTER":END
70 PRINT ",";:GOTO 50
run
ILOVEYOU, AVT MSX COMPUTER
OK
```

# HEX$ (n)

Returns a string which represents the hexadecimal value of the decimal argument.

'n' is a numeric expression in the range -32768 to 32767. If "n" is negative, the two's complement from is used. That is, HEX$ (-n) is the same as HEX$ (65536 -n). The answer becomes to be suppressed leading zeros.

Ex)
```
10 PRINT "HEX         DEC"
20 FOR I = 0 TO 16 STEP 4
30 PRINT RIGHT$ ("0" + HEX$(I),2);"         ";
40 PRINT USING "##";I
50 NEXT I:END
run
HEX         DEC
00            0
04            4
08            8
0C           12
10           16
OK
```

# IF~THEN~ELSE
# IF~GOTO~ELSE

**IF <expression> THEN <statements> ELSE <statements>**

**IF <expression> GOTO <statements> ELSE <statements>**

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.

Ex) 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.

Ex)
```
10 INPUT "INPUT No."; N: PRINT"          ";
20 IF N = 1 THEN PRINT "ABC"
30 IF N = 2 THEN PRINT "DEF"
40 IF N = 3 THEN PRINT "GHI"
50 IF N = 4 THEN GOTO 70
60 PRINT:GOTO 10
70 END
run
INPUT No. ?  1
        ABC

INPUT No. ?  3
        GHI

INPUT No. ?  4

OK
```

# 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.

Ex)
```
10 PRINT "Hit any key or space bar to end"
20 I$ = INKEY$:IF I$ = "" THEN 20
30 IF I$ = " " THEN PRINT "ENDED !":END
40 PRINT I$:" 'S ASCII CODE IS ";ASC(I$)
50 GOTO 10
run
Hit any key or space bar to end
1'S ASCII CODE IS 49
Hit any key or space bar to end
D'S ASCII CODE IS 68
Hit any key or space bar to end
*'S ASCII CODE IS 42
Hit any key or space bar to end
ENDED !
Ok
```

# INP (n)

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

In above statements and functions, port number (n) is handled with 16bit number to support Z80's capability that accesses I/O port with [BC] register pair. However, standard MSX system does not support those extended I/O address space, port number larger than 255 is meaningless.

Ex)
```
10 CLS
20 PRINT "PUSH SPACE, HOME, INS, DEL, CURSOR KEYS !"
30 OUT 170, (INP (170) AND &HF0) OR 8
40 LOCATE 10, 10:PRINT RIGHT$ ("0000000"+BIN$ (INP(169)), 8):G
OTO 30
run
PSH SPACE, HOME, INS, DEL, CURSOR KEYS !
```

# INPUT

**INPUT ["< prompt string >";] <variable>, <variable>,....**

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 order of variables. The number of data items supplied must be the same as the number of variables. Data items are separated by commas.

The variable name 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.

Ex)

```
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.

Ex)

```
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.

Ex)

```
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.

# INPUT #

**INPUT # <file number>, <variable> [, <variable>.....]**

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 < variables >. 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 return, 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.

Ex)
```
10 OPEN "CAS:DATA" FOR INPUT AS #1
20 INPUT #1, A, B
30 PRINT "DATA A = ";A
40 PRINT "DATA B = ";B
50 END
```

# INPUT $

**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.

Ex)
```
10 'PASSWORD : "MSX"+'ESC'KEY
20 P$ = "MSX"+CHR$ (27)
30 PRINT "Password ";
40 A$ = INPUT$(4):PRINT A$
50 IF A$ < > P$ THEN PRINT" You are NOT allowed !":GOTO 30
60 PRINT:BEEP:PRINT" Welcome to MSX world! "
70 END
run
Password ABCD
You are NOT allowed!
Password MSX

Welcome to MSX world!
OK
```

# INSTR

**INSTR ([N], X$, Y$)**

Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset N sets the position for starting the search. N 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 N or 1. X$ and Y$ may be string variables, string expressions, or string literals.

Ex)
```
10 ' *** PIANO ***
20 PRINT "C = Do"
30 TB$ = "AZSXCFVGBNJMK,L./,Q2WE4R5T6YU8I9 0p-[=]"
40 PLAY"L8"
50 IN$ = INKEY$:IF IN$ = " " THEN GOTO 50
60 I = INSTR (TB$, IN$)
70 IF I > 0 THEN I = I+32
80 PLAY "N = I;"
90 GOTO 50
run
C = Do
```

# INT (X)

Returns the largest integer < = X.

Compare to FIX statement.

Ex)
```
10 PRINT "X", "INT(X)"
20 FOR I =-2.4 TO 2.4
30 PRINTI, INT (I)
40 NEXT I:END
run
X         INT(X)
-2.4      -3
-1.4      -2
-4        -1
 6         0
 1.6       1
OK
```

# INTERVAL ON/OFF/STOP

To activate/deactivate trapping of time interval in a BASIC program.

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

If an 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.

Ex)
```
10 ON INTERVAL = 100 GOSUB 80
20 N = 20:CLS:COLOR 2
30 LOCATE 5, 5:PRINT"TIME"
40 INTERVAL ON
50 LOCATE 5, 7:PRINT SPACE$ (128)
60 LOCATE 5, 7:PRINT STRING$(N,"*")
70 GOTO 60
80 '
90 N= N-1
100 IF N < 0 THEN END
110 IF N > 10 THEN COLOR 10
120 IF N < 5 THEN COLOR 8
130 RETURN 50
```

```
TIME
* * * * * * * * * * * * * * * * * * * * *
```

# KEY

**KEY < function key #>"< string characters> "**

To set a string to specified function key < function key # > must be in the range 1 to 10. < string characters > must be within 15 characters.

Ex)

| | |
|---|---|
| color auto goto list run | key 1, "LOAD"<br><br>LOAD auto goto list run |

# KEY LIST

To list the contents of all function keys.

Ex)
```
KEY LIST
color
auto
goto
list
run
color 15, 4, 4
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.

# KEY (n) 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.

Ex)
```
10 ON KEY GOSUB 80, 90, 100, 110, 120, 130, 140, 150, 160, 170
20 KEY(1) ON:KEY(2) ON
30 KEY(3) ON:KEY(4) ON
40 KEY(5) ON:KEY(6) ON
50 KEY(7) ON:KEY(8) ON
60 KEY(9) ON:KEY(10) ON
70 GOTO 70
80 N = 1:GOTO 180
90 N = 2:GOTO 180
100 N = 3:GOTO 180
110 N = 4:GOTO 180
120 N = 5:GOTO 180
130 N = 6:GOTO 180
140 N = 7:GOTO 180
150 N = 8:GOTO 180
160 N = 9:GOTO 180
170 N = 10
180 PRINT USING " F - ##  KEY IS DEPRESSED";N
190 RETURN
run
F- 1 KEY IS DEPRESSED
F- 3 KEY IS DEPRESSED
F-10 KEY IS DEPRESSED
```

# KEY ON/OFF

To turn on/off function key display on 24th line of text screen.

Ex)
```
KEY OFF
OK

KEY ON
OK
color auto goto list run
```

# LEFT $

**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.

Ex)
```
10 INPUT A$
20 FOR I = 1 TO LEN(A$)
30 PRINT LEFT$(A$, I)
40 NEXT I
50 END
run
 ? AVT
A
AV
AVT
OK
```

# LEN (X$)

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

Ex)
```
10 A$ = "daewoo computer"
20 PRINT LEN(A$)
run
 15
```

# LET

**[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.

Ex)
```
10 LET A = 10:PRINT A
20 A = 5:PRINT A
30 LET A = 1:PRINT A
40 END
run
 10
 5
 1
OK
```

# LINE

```
LINE [|(X1, Y1)       |] - |(X2, Y2)       | [, < color code > ] [, | B |]
     |STEP (X1, Y1)   |    |STEP (X2, Y2)  |                       | 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.

Ex)
```
10 SCREEN 2
15 CLS
20 FOR I = 0 TO 95 STEP 3
30 LINE (131-I, 95-I)-(131+I, 95+I), 10, B
40 NEXT I
50 GOTO 50
```

# LINE INPUT

**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.

Ex)
```
10 PRINT " * YOU CAN PRINT" CHR$(34); "&";CHR$(44)
20 PRINT " * BY 'LINE INPUT' "
30 LINE INPUT A$
40 PRINT "A$ = "; A$
50 END
run
   * YOU CAN PRINT "&,
   * BY 'LINE INPUT'
"AVT", MSX
A$ = "AVT", MSX
OK
```

# LINE INPUT #

**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.

Ex)
```
10 OPEN "CAS:DATA" FOR INPUT AS #1
20 LINE INPUT #1, A$
30 PRINT A$
40 CLOSE:END
```

# LIST

**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 the same time.
Listing is suspended by typing "STOP" key, and it is resumed by typing "STOP" key again.

Ex)
```
10 A = 3
20 B = 6
30 C = A+B
40 PRINT A, B, C
50 END
list
10 A = 3
20 B = 6
30 C = A+B
40 PRINT A, B, C
50 END
OK
LLIST
OK
```

# LIST

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

To list all or part of the program on the printer. (See the LIST command for details of the parameters)

# LOAD

**LOAD "<file name>" [,R]**

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.

Ex) LOAD "CAS:TEST"
    Found: TEST

# LOCATE

**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

Ex)
```
10 A$ = "ABCDEFGHIJKLMN"
20 CLS: FOR I = 1 TO 15
30 LOCATE I, I-1:PRINT LEFT$(A$, I)
40 NEXT I:END
run
 A
  AB
   ABC
    ABCD
     ABCDE
      ABCDEF
       ABCDEFG
        ABCDEFGH
         ABCDEFGHI
          ABCDEFGHIJ
           ABCDEFGHIJK
            ABCDEFGHIJKL
             ABCDEFGHIJKLM
              ABCDEFGHIJKLMN
               ABCDEFGHIJKLMN
OK
```

# LOG (X)

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

Ex)
```
10 FOR I = 10 TO 50 STEP 10
20 PRINT " LOG(";I;") = ";LOG(I)
30 NEXT I:END
run
LOG (  10  ) = 2.302585092994
LOG (  20  ) = 2.995732273554
LOG (  30  ) = 3.4011973816622
LOG (  40  ) = 3.6888794541139
LOG (  50  ) = 3.912023005428
OK
```

# 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.

Ex)
```
10 PRINT LPOS(X)
20 LPRINT "ABCDEFG";
30 PRINT "AT 30:";LPOS(X)
40 LPRINT "HIJKLNOP";
50 PRINT "AT 50:";LPOS(X)
60 LPRINT "QRS"
70 PRINT "AT 70:";LPOS(X)
80 END
run
 0
AT 30:7
AT 50:15
AT 70:0
Ok
```

# LPRINT/LPRINT USING

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

Ex)
```
10 PRINT "PRINT ANY SENTENCE"
20 LINE INPUT A$
30 PRINT A$
40 LPRINT  A$
50 END
run
 PRINT ANY SENTENCE
"AVT", "MSX"
"AVT", "MSX"
OK
```

# MAXFILES

**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.

Ex)
```
10 MAXFILES = 3
20 OPEN "CAS:DATA 1" FOR OUTPUT AS #1
30 OPEN "CRT:DATA 2" FOR OUTPUT AS #2
40 OPEN "LPT:DATA 3" FOR OUTPUT AS #3
50 ...
```

# MERGE

**MERGE "<device>[<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 the BASIC returns to command level.

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

Ex)
```
MERGE "CAS:SAMPLE"
FOUND:SAMPLE
OK
RUN
```

# MID$

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

Returns a string of length J character 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 character to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), returns a null string.

Ex)
```
10 A$ = "ABC12345+*?="
20 PRINT MID$(A$, 4, 7)
30 END
run
12345+*
Ok
```

# MID $

**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>.

Ex)
```
10 A$ = "abcdefg"
20 B$ = "h swearty"
30 PRINT A$;" ";B$
40 PRINT
50 FOR I = 1 TO 7
60 C$ = A$
70 MID$(C$, I, 3) = B$
80 PRINT  C$;" ";I
90 NEXT
run
abcdefg h swearty

h sdefg 1
ah sefg 2
abh sfg 3
abch sg 4
abcdh s 5
abcdeh  6
abcdefh 7
```

# MOTOR

**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.

Ex)
```
MOTOR ON
Ok
MOTOR OFF
Ok
```

# NEW

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

Ex)
```
list
10 A$ = "ABC12345+*?="
20 PRINT MID$(A$, 4, 7)
30 END
OK
NEW
OK
list
OK
```

# 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 form is used. That is, OCT$ (-n) is the same as OCT$ (65536-n).

Ex)
```
10 PRINT " X  OCT(X)"
20 FOR I = 1 TO 16 STEP 2
30 PRINT USING "##   &&"; I;OCT$ (I)
40 NEXT I:END
run
 X   OCT (X)
 1    1
 3    3
 5    5
 7    7
 9    11
11    13
13    15
15    17
OK
```

# ON ERROR GOTO

**ON ERROR GOTO <line number >**

To enable error trapping and specify the first line of the error handling sub-routine.

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 subroutines 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.

Ex) ON ERROR GOTO 100

# ON GOSUB/ON GOTO

**ON <expression> GOSUB <line number>....**
**GOTO**

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.

Ex)
```
10 INPUT "INPUT NO. (1~3)";A
20 ON A GOTO 40, 50, 60
30 GOTO 10
40 PRINT "A = 1":GOTO 10
50 PRINT "A = 2":GOTO 10
60 PRINT "A = 3":GOTO 10
70 END
run
INPUT NO. (1~3) ? 2
A = 2
INPUT NO. (1~3) ? 3
A = 3
INPUT NO. (1~3) ? 6
INPUT NO. (1~3) ? 1
A = 1
INPUT NO. (1~3) ?
```

# ON INTERVAL GOSUB

**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).

# ON KEY GOSUB

**ON KEY GOSUB <list of line numbers>**

To set up a line numbers for BASIC to trap to when the function key 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 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).

Ex)
```
10 ON KEY GOSUB 50, 60, 70, 80
20 KEY(1) ON:KEY(2) ON
30 KEY(3) ON:KEY(4) ON
40 GOTO 40
50 N = 1:GOTO 90
60 N = 2:GOTO 90
70 N = 3:GOTO 90
80 N = 4:GOTO 90
90 PRINT USING " F- ## key is depressed";N
100 RETURN
RUN
    F-1 key is depressed
    F-2 key is depressed
    F-3 key is depressed
    F-4 key is depressed
```

# ON SPRITE GOSUB

**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 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).

Ex)

```
10 A$ = "- ~ < < < < ♥ <"
20 B$ = "< ♥ ~ < < < ~ - "
30 SCREEN 2
40 ON SPRITE GOSUB 140
50 SPRITE$(0) = A$
60 SPRITE$ (1) = B$
70 SPRITE ON
80 XA = RND(1) * 100
90 XB = RND(1) * 100
100 FOR Y = 0 TO 191
110 PUT SPRITE 0, (50+XA, Y), 6
120 PUT SPRITE 1, (50+XB, 191-Y), 3
130 NEXT Y:GOTO 70
140 SPRITE OFF
150 PLAY "L4CEDFDECREFGAGFER"
160 IF PLAY (0) THEN 160
170 PUT SPRITE 0, (0, 208)
180 PUT SPRITE 1, (0, 208)
190 Y = 191:RETURN
```

# ON STOP GOSUB

**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 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).

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

Ex)
```
10 ON STOP GOSUB 60
20 STOP ON
30 INPUT A$:PRINT A$
40 IF A$ = "END" THEN STOP OFF: END
50 GOTO 30
60 PRINT "PLEASE, PRINT 'END' "
70 RETURN
run
? BCD
BCD
? [CTRL] + [STOP]
PLEASE, PRINT 'END'
BCD
? END
END
OK
```

# ON STRIG GOSUB

**ON STRIG GOSUB <list of line numbers>**

To set up a line numbers for BASIC to trap to when the trigger button is pressed.

Ex) 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).

Ex)
```
10 CLS
20 ON STRIG GOSUB 80
30 STRIG (0) ON
40 PRINT "IF YOU DEPRESS SPACE BAR"
50 GOTO 50
80 COLOR 15, 1, 1
90 PLAY "T255CEG"
100 LOCATE 8, 12:PRINT "PUSHED ! ! !"
110 FOR I = 0 TO 100:NEXT I
120 LOCATE 8, 12:PRINT SPC(10)
130 COLOR 15, 4, 7
140 RETURN
```

# OPEN

**OPEN "<device descripter> [:] [<file name>]" [FOR <mode>] As [#] <filenumber>**

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

Ex)
```
10 OPEN "CAS:DATA" FOR INPUT AS#1
20 LINE INPUT #1, A$
30 PRINT A$
40 CLOSE:END
```

# OUT

**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.

Ex)
```
10 OUT&HA0, 7:OUT&HA1, 7
20 OUT&HA0, 8:OUT&HA1, 8
30 FOR I = 0 TO 31
40 OUT&HA0, 6:OUT&HA1, I
50 FOR J = 1 TO 50:NEXT J
60 NEXT I
70 PRINT "hit the < CTRL > + < STOP > key"
80 END
run
hit the < CTRL > + < STOP > key
```

# PAD

**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).

Ex)
```
10 SCREEN 2
20 IF PAD(0) = 0 THEN SW = 0
30 X = PAD(1):Y=PAD(2)
40 IF SW = 0 THEN PSET(X, Y) ELSE LINE-(X, Y)
50 SW = 1
60 GOTO 20
```

# PAINT

**PAINT (X, Y)**
**STEP (X, Y) [, < 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. PAINT does not allow <coordinate specifier> to be out of the screen.

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.

Ex)
```
10 SCREEN 2
20 FOR C = 1 TO 10
30 X1 = RND(1)*256
40 X2 = RND(1)*256
50 Y1 = RND(1)*192
60 Y2 = RND(1)*192
70 LINE (X1, 0)-(X2, 191), 10
80 LINE (0, Y1)- (255, Y2), 10
90 NEXT C
100 FOR C = 1 TO 20
110 X3 = RND(1)*256
120 Y3 = RND(1)*192
130 PAINT(X3, Y3), 10
140 NEXT C
150 GOTO 150
```

# PDL

**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.

Ex) PDL (2)

# 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.

Ex)
```
10 '*** MEMORY DUMP ***
20 INPUT "ADDRESS = "; A
30 IF A < 0 THE A = A+65536 !
40 FOR I = A TO A+64 STEP 8
50 PRINT " "; RIGHT$("000"+HEX$ (I), 4);
60 PRINT USING "(#####)";I;
70 FOR J = I TO I+7
80 PRINT RIGHT$("0"+HEX$(PEEK (J)), 2);
90 NEXT J:PRINT
100 NEXT I:END
run
ADDRESS = ?-100
 FF9C(65436)C3D702BF1B9898
 FFA4(65444)C3D702BF1B9898
 FFAC(65452)C3D702BF1B9898
 FFB4(65460)C3D702BF1B9898
 FFBC(65468)C3D702BF1B9898
 FFC4(65476)C3D702BF1B9898
 FFCC(65484)C3D702BF1B9898
 FFD4(65492)C3D702BF1B9898
 FFDC(65500)C3D702BF1B9898
OK
```

# PLAY

**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.

On ; 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.

Nn ; 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 (On) and the note name (A-G). (The C of octave 4 is 36.)

Ln ; 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. The default is 4.

Rn ; Pause(rest). n may range from 1 to 64, and figures the length of the pause in the same way as L(length). The default is 4.

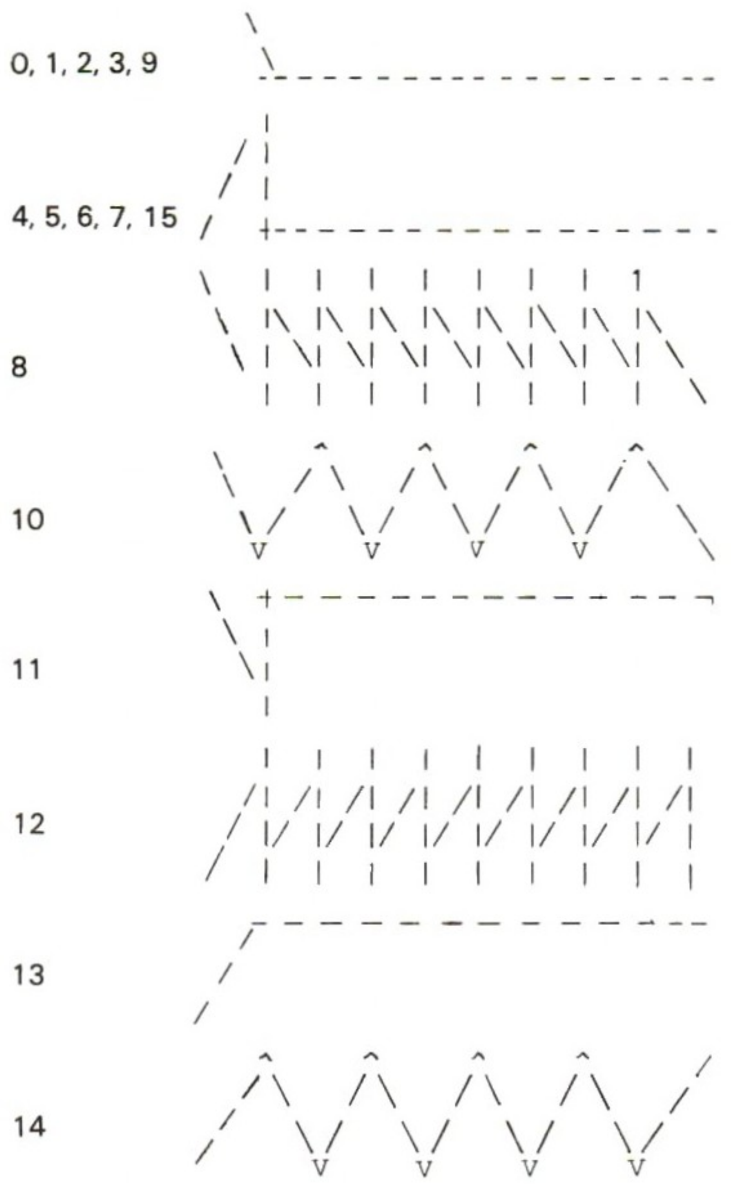
; (Dot or period) After a note, causes the note to be played as a dotted note. That is, its length is multiplied by 3/2. More than one dot may appear after the note, and the length is adjusted accordingly. For example, "A..." will play 27/8 as long, etc. Dots may also appear after the pause(P) to scale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

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

Vn ; Volume. Sets the volume of output. n may range from 0 to 15. The default is 8.

Mn ; Modulation. Sets period of envelope. n may range from 1 to 65535. The default is 255.

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




**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.
Note that values specified with above commands will be reset to the system default when beep sound is generated.

## PLAY <n>

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.

Note that immediate after the PLAY statement is issued, the PLAY function returns -1 regardless to the actual status of the music queue.

Ex)
```
10 'BACH
20 PLAY "T240L6V12", "T240L2V9"
30 PLAY "R8O6GABO7DCCED", "O4GO5GE"
40 PLAY "DGF#GDO6BGAB", "O4BO5EO4E"
50 PLAY "O7CDEDCO6BGABG", "O4ABO5C"
60 PLAY "F#GADF#AO7CO6BA", "O5DF#D"
70 PLAY "BGA8O7DCCED", "GGC"
80 PLAY "DGF#GDO6BGAB", "O4BO5ED"
90 PLAY "EO7DCO6BAGDGF#G2", "CC#DG"
100 IF PLAY (0) THEN 100
110 PRINT " * * * THE END * * *"
run
 * * * THE END * * *
OK
```

# POINT

**POINT (X,Y)**

Returns color of a specified pixel .

Ex)
```
10 SCREEN 2
20 OPEN "GRP:"FOR OUTPUT AS #1
30 FOR I = 1 TO 20
40 FOR J =1  TO 8
50 PRESET(J * 24, I * 8)
60 C = INT (RND(1) *13)+2
70 COLOR C:PRINT#1, " ♥ "
80 NEXT J, I
90 COLOR 15
100 PRESET (100, 170)
110 PRINT #1, "Color No."
120 FOR I = 1 TO 20
130 FOR J = 1 TO 8
140 K = POINT(J*24+4, I*8+4)
150 PRESET(J*24+8, I*8)
160 PRINT #1, USING "##"; K
170 NEXT J, I
180 GOTO 180
RUN
```

| ♥ 9 | ♥ 3 | ♥ 11 | ♥ 9 | ♥ 11 | 4 | ♥ 6 | ♥ 14 | ♥ 10 |
|---|---|---|---|---|---|---|---|---|
| ♥ 8 | ♥ 12 | ♥ 8 | 4 | ♥ 3 | ♥ 8 | ♥ 12 | ♥ 2 | ♥ 8 |
| ♥ 10 | ♥ 7 | ♥ 12 | ♥ 12 | ♥ 9 | ♥ 2 | ♥ 5 | ♥ 6 | 4 |
| 4 | ♥ 12 | ♥ 3 | ♥ 5 | ♥ 7 | ♥ 7 | ♥ 8 | ♥ 14 | ♥ 10 |
| ♥ 3 | ♥ 3 | ♥ 2 | ♥ 3 | ♥ 3 | ♥ 6 | 12 | ♥ 9 | ♥ 14 |
| ♥ 6 | 4 | ♥ 14 | ♥ 8 | ♥ 3 | ♥ 8 | ♥ 10 | ♥ 14 | 4 |
| ♥ 13 | ♥ 10 | ♥ 11 | 4 | ♥ 11 | ♥ 12 | ♥ 4 | ♥ 5 | ♥ 9 |
| ♥ 5 | ♥ 7 | ♥ 10 | ♥ 9 | ♥ 5 | ♥ 14 | ♥ 12 | ♥ 3 | ♥ 12 |
| ♥ 5 | ♥ 14 | ♥ 2 | ♥ 3 | ♥ 12 | ♥ 2 | ♥ 8 | 4 | ♥ 9 |
| ♥ 11 | ♥ 3 | ♥ 8 | ♥ 14 | 4 | ♥ 14 | 6 | ♥ 13 | ♥ 8 |
| 4 | ♥ 9 | ♥ 11 | ♥ 7 | ♥ 8 | ♥ 10 | ♥ 13 | ♥ 14 | ♥ 6 |
| ♥ 9 | ♥ 12 | ♥ 10 | ♥ 13 | ♥ 2 | ♥ 3 | ♥ 4 | ♥ 2 | ♥ 6 |
| ♥ 11 | ♥ 9 | ♥ 2 | ♥ 8 | ♥ 9 | ♥ 7 | ♥ 12 | ♥ 12 | ♥ 2 |
| ♥ 5 | ♥ 13 | ♥ 11 | ♥ 12 | ♥ 13 | ♥ 9 | ♥ 7 | ♥ 3 | ♥ 7 |
| ♥ 12 | ♥ 14 | ♥ 11 | ♥ 7 | ♥ 6 | ♥ 13 | ♥ 3 | ♥ 11 | ♥ 14 |
| ♥ 3 | ♥ 9 | ♥ 11 | ♥ 5 | ♥ 9 | ♥ 6 | ♥ 7 | ♥ 12 | ♥ 5 |
| 4 | ♥ 10 | ♥ 5 | ♥ 12 | ♥ 2 | ♥ 8 | ♥ 13 | ♥ 5 | 4 |
| 4 | 4 | ♥ 6 | ♥ 12 | ♥ 9 | ♥ 7 | ♥ 11 | ♥ 11 | ♥ 5 |
| ♥ 2 | 4 | ♥ 14 | ♥ 2 | ♥ 12 | ♥ 8 | ♥ 7 | ♥ 10 | ♥ 2 |
| ♥ 2 | ♥ 10 | ♥ 9 | ♥ 6 | ♥ 6 | ♥ 11 | 11 | ♥ 5 | ♥ 9 |
| ♥ 9 | ♥ 14 | ♥ 13 | ♥ 2 | ♥ 9 | ♥ 11 | 10 | ♥ 11 | ♥ 5 |

Color No.

# POKE

**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.

Ex)
```
10 CLEAR 256, &HE000
20 'IT POKES AT &HE001 ~ E010
30 FOR I = &HE001 TO &HE010
40 POKEI, 256+I MOD 256
50 NEXT I
60 'IT PEEKS AT &HE000 ~ E020
70 FOR I = &HE000 TO &HE020 STEP 8
80 FOR J = I TO I+7
90 PRINT USING "& & "; HEX$ (PEEK (J));
100 NEXT J:PRINT
110 NEXT I
120 END
run
FF 1  2  3  4  5  6  7
8  9  A  B  C  D  E  F
10 3  FC 0  FF 3  FC 0
FF 3  FC 0  FF 3  FC 0
FF 3  FC 0  FF 3  FC 0
OK
```

# POS (I)

Returns the current cursor positon. The leftmost position is 0. I is a dummy argument.

Ex)
```
10 CLS
20 FOR I = 0 TO 13
30 LOCATE I*2, I:PRINT POS(X)
40 NEXT I
50 END
RUN
0
  2
    4
      6
        8
          10
            12
              14
                16
                  18
                    20
                      22
                        24
                          26
OK
```

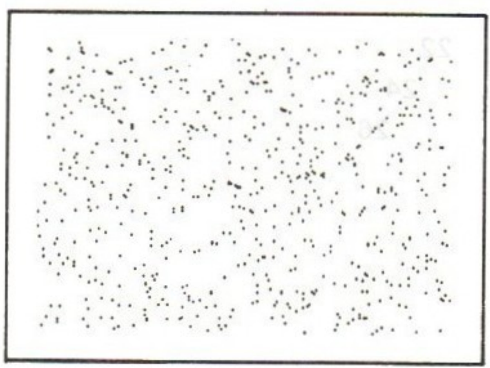
# PRESET

**PRESET <coordinate specifier> [, <color> ]**

To 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.

Ex)
```
10 SCREEN 2
20 LINE (10, 10)-(245, 180), 15, BF
30 FOR I = 0 TO 700
40 X = INT(RND(1) * 233)+11
50 Y = INT(RND(1) * 168)+11
60 PRESET (X, Y)
70 NEXT I
80 GOTO 80
RUN
```



# PRINT

**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.

Ex)
```
PRINT "AVT MSX"
AVT MSX
OK
? "COMPUTER !"
COMPUTER !
OK
10 PRINT "My Friend"
run
My Friend
OK
```

# PRINT USING

**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$ = "WORLD"
Ok
PRINT USING "!" ;A$
W
Ok
```

" \ n spaces \ "

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

If the '\' signs 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$="WORLD"
Ok
PRINT USING "\  \" ; A$
WORL
Ok
```

"&"

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

Example:
A$="You"
Ok
PRINT USING "I love & very much.";A$
I love you 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 dollar sign causes dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

Example:

```
PRINT USING "$$ ###.##";12.35,-12.35
$ 12.35-$12.35
Ok
PRINT USING "$$###.##-";12.35,-12.35
$ 12.35  $12.35-
Ok
```

"**$"

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

```
PRING USING "##.##^^^^"; 234.56
 2.35E+02
Ok
PRINT USING "#.##^^^^";-12.34
-.12E+02
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 filed, 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.

Ex)
```
10 PRINT USING "######"; 123456 !
20 PRINT USING "#. ####"; 12.345
30 PRINT USING "####.#"; 1234.56
40 PRINT USING "$$####" ; 12345 !
50 PRINT USING "####, #"; 12345 !
60 PRINT USING "** ###"; 12 !
70 PRINT USING "##### +"; 12345
80 PRINT USING "######  ^ ^ ^ ^"; 1234567890#
90 END
run
123456
%12.3450
1234.6
$ 12345
12,345
*** 12
12345+
 12346E+05
OK
```

# PRINT #/PRINT # USING

**PRINT # <file number>, [<expression> ......]**
**PRINT # <file number> USING <expression>; [<expression> ......]**

To write data to the specified channel. (See PRINT/PRINT USING statements for details.)

Ex)
```
10 CLS
20 A$ = "AVT MSX"
30 OPEN "CRT:" FOR OUTPUT AS #1
40 LOCATE 10, 10
50 PRINT #1, A$
60 CLOSE
RUN




    AVT MSX
OK
```

# PUT SPRITE

**PUT SPRITE <sprite plane number> [,<coordinates specifier>] [,<color>] [,<pattern number>]**

To set up sprite attributed.

<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.
And the values outside of the screen will be substituted with the nearest possi ble value. For example, 0 for any negative coor dinate specification.

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.

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 is 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)

Ex)
```
10 DEFINT A-Z
20 DIM X(10), Y(10)
30 SCREEN 2, 3:COLOR, 1, 1:CLS: I=RND ( - TIME)
40 FOR I=1 TO 10:X(I)=96 Y(I)=I*15:NEXT
50 FOR I=0 TO 31:READ A$:B$=B$+CHR$(VAL("&H"+A$)):NEXT
60 SPRITE$(0)=B$
70 FOR I=1 TO 10
80 PUT SPRITE I, (X(I), Y(I)), I+4, 0
90 NEXT
100 FOR I=1 TO 10
110 X(I)=(X(I)+(RND (1) * 21-10)) MOD 256
120 Y(I)=(Y (I)+(RND (1) * 21-10))MOD 192
130 NEXT
140 GOTO 70
150 DATA 0C, 06, 62, F2, FA, DD, CF, C7
160 DATA FF, 7F, 3F, 1B, 37, 3E, 1C, 00
170 DATA 30, 30, 46, 4F, 5F, BB, FB, F3, E3
180 DATA FF, FE, FC, D8, EC, 7C, 38, 00
```

# READ

**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.

Ex)
```
10 FOR I=1 TO 3
20 READ A$, B, C
30 PRINT TAB(3);A$
40 D=B+C
50 PRINT TAB(7) USING "######";B;C;D
60 NEXT I
70 DATA "AVT", 50, 60
80 DATA "MSX", 30, 40
90 DATA "COMPUTER", 1, 2
RUN
   AVT
            50    60    110
   MSX
            30    40     70
   COMPUTER
             1     2      3
OK
```

# REM

**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.

Ex)
```
10 REM statement of read, data
20 FOR I=1 TO 3
30 READ A$
40 PRINT A$ ; SPC(2)
50 NEXT
60 END
70 DATA avt, msx, computer
run
avt msx computer
OK
```

# RENUM

**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.

Ex)
```
10 REM RENUM
20 '
25 '
55 '
70 '
RUN
OK
RENUM 20, 10, 5
OK
list
20 REM RENUM
25 '
30 '
35 '
40 '
```

# RESTORE

**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.

Ex)
```
10 RESTORE:READ A, B, C, D, E, F, G
20 PRINT A, D, F
30 RESTORE 100
40 READ A$, B$, C$, D$, E$, F$, G$
50 PRINT A$; B$; C$; D$; E$; F$; G$
60 END
70 DATA 1, 4, 6
80 DATA 2, 5, 3
90 DATA 7, 8, 9, 0
100 DATA D, P, C,-
110 DATA 2, 0, 0
120 DATA A, B, C, D, E, F, G
```

# RESUME

```
         |[o]            |
RESUME   |NEXT           |
         |<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.

Ex)
```
10 ON ERROR GOTO 100
20 INPUT "X="; X
30 PRINT SQR(X);
40 IF F=1 THEN PRINT "i"ELSE PRINT
50 F=0:GOTO 20
100 X=- X:F=1
110 RESUME
```

# RIGHT $

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.

Ex)
```
10 CLS
20 INPUT "DATA="; A$
30 L=LEN (A$)
40 FOR I=1 TO L
50 B$=RIGHT$(A$, I)
60 B$=RIGHT$(SPACE$(50)+B$, L)
70 PRINT B$
80 NEXT
90 END
RUN
DATA=? AVT
          T
         VT
        AVT
```

# RND

**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.

Ex)
```
10 FOR I=1 TO 5
20 A=RND(1)
30 PRINT A;TAB(18);INT(A * 10)
40 NEXT I
50 END
RUN
 .59521943994623    5
 .10658628050158    1
 .76597651772823    7
 .57756392935958    5
 .73474759503023    7
```

# RUN

**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.

Ex)
```
10 PRINT "LOVE"
20 PRINT "COMPUTER"
RUN 20
COMPUTER
OK
```

# SAVE

SAVE "[<device descripter>] <file name> "

To save a BASIC program file to the device. Control + Z is treated as end-of-file.

Ex) SAVE "CAS:SAMPLE"
OK

# SCREEN

**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 40×24 text mode, 1 to select 32×24 text mode, 2 to select high resolution mode, 3 to select multi color (low-resolution mode).

0:40×24 text mode
1:32×24 text mode
2: high resolution mode
3: multi color mode

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

0:8×8 unmagnified
1:8×8 magnified
2: 16×16 unmagnified
3: 16×16 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
non zero: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 that default baud rate for succeeding write operation. 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 operations is 'MSX printer' (which has 'graphics symbol') or not. Should be non-0 if the printer does not have such capability. In this case, graphics symbols are converted to spaces.

Ex) SCREEN 1, 2

# SGN (X)

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

Ex)
```
10 INPUT "SIGN"; I
20 J=SGN(I)
30 J=J+2
40 ON J GOSUB 90, 100, 110
50 PRINT "THIS IS ";
60 PRINT A$
70 PRINT
80 GOTO 10
90 A$="MINUS":RETURN
100 A$="ZERO":RETURN
110 A$="PLUS":RETURN
RUN
SIGN ? 7
THIS IS PLUS

SIGN ? 0
THIS IS ZERO

SIGN ? -1
THIS IS MINUS

SIGN ?
```

# SIN (X)

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

Ex)
```
10 FOR I=1 TO 3
20 A=SIN(I)
30 PRINT I;A
40 NEXT
50 END
RUN
  1 .84147098480792
  2 .90929742682566
  3 .14112000805978
```

# SOUND

**SOUND <register of PSG>, <DATA>**

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

Ex)
```
SOUND 7,254
```

# SPACE $

**SPACE $ (X)**

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

Ex)
```
10 OPEN "CRT:" FOR OUTPUT AS #1
20 CLS: FOR I=0 TO 10
30 S=INT(RND(1) *17)
40 PRINT #1, " * ";
50 PRINT #1, SPACE$(S);" * ";
60 PRINT #1, SPACE$(17-S); "SPACE"; S
70 NEXT
80 CLOSE:END
RUN
*          *       SPACE 10
**                 SPACE 1
*             *    SPACE 13
*         *        SPACE 9
*            *     SPACE 12
*   *              SPACE 3
*      *           SPACE 6
*                * SPACE 16
*          *       SPACE 10
*       *          SPACE 7
*              *   SPACE 14
OK
```

# 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.

Ex)
```
10 CLS:FOR I = 0 TO 10
20 S=INT(RND(1) * 17)
30 PRINT "*";
40 PRINT SPC(S); "*";
50 PRINT SPC(17-S); "SPACE";S
60 NEXT:END
RUN
*          *      SPACE 10
**                SPACE 1
*             *   SPACE 13
*         *       SPACE 9
*            *    SPACE 12
*   *             SPACE 3
*      *          SPACE 6
*                *SPACE 16
*          *      SPACE 10
*       *         SPACE 7
*              *  SPACE 14
OK
```

# 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.

Ex) SPRITE ON

# SPRITE $

**SPRITE $ ( < pattern number > )**

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.

Ex)
```
10 SCREEN 1, 3
20 PRINT "*** MOUSE ***"
30 FOR I=1 TO 16
40 READ D$
50 A$=A$+CHR$(VAL("&B"+LEFT$(D$, 8)))
60 B$=B$+CHR$(VAL("&B"+RIGHT$(D$, 8)))
70 NEXT I
80 SPRITE$(0)=A$+B$
90 PUT SPRITE 0, (50, 70), 15, 0
100 PUT SPRITE 1, (90, 70), 14, 0
110 PUT SPRITE 2, (130, 70), 1, 0
120 PUT SPRITE 3, (170, 70), 13, 0
130 PRINT "PUT SPRITE 0, (0, 208)"
140 DATA 0000000000011110
150 DATA 0000100000101001
160 DATA 0001011111101101
170 DATA 0000100000101001
180 DATA 0011111011111111
190 DATA 0001111111111000
200 DATA 0000001111111000
210 DATA 0000011111110000
220 DATA 0000001111100010
230 DATA 0000000111100100
240 DATA 1100001111100100
250 DATA 0011111111110010
260 DATA 0000001111110010
270 DATA 0000001111110010
280 DATA 0000000111111100
290 DATA 0000011111110000
```

# SQR (X)

Returns the square root of X. X must be > = 0

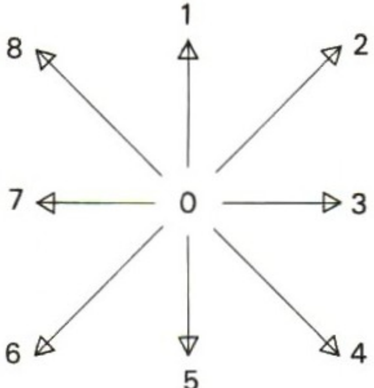
Ex)
```
10 INPUT "I=";I
20 PRINT "SQR(i)=";
30 PRINT SQR(I)
40 PRINT "I ^ .5=";
50 PRINT I ^ .5
60 PRINT
70 GOTO 10
RUN
I=? 23
SQR(i)=4.7958315233127
I ^.5=4.7958315233124

I=? 55
SQR(i)=7.4161984870955
I ^ .5=7.4161984870947
I=?
```

# STICK

**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.




Ex)
```
10 PRINT STICK (0);
20 PRINT STICK (1);
30 PRINT STICK (2)
40 GOTO 10
run
 1    0    0
 1    0    0
 0    0    0
 7    0    0
 7    0    0
 7    0    0
 0    0    0
 0    0    0
 0    0    0
 5    0    0
 5    0    0
```

# 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 nnn (nnn is a line number)

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

Execution is resumed by issuing a CONT command.

Ex)
```
10 INPUT"input no. (1 OR 2)";N
20 IF N=1 THEN PRINT "END":END
30 IF N=2 THEN PRINT "STOP":STOP
40 GOTO 10
run
input no. (1 OR 2) ? 2
STOP
Break in 30
OK
```

# 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.

Ex)
```
STOP ON
```

# STRIG

**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.

Ex)
```
10 PRINT STRIG (0);
20 PRINT STRIG (1);
30 PRINT STRIG (2)
40 GOTO 10
run
 0    0   0
 0    0   0
-1   -1   0
-1    0   0
 0   -1   0
-1    0   0
-1   -1   0
 0    0   0
 0    0   0
```

# STRIG ON/OFF/STOP

**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.

Ex) STRIG (0) ON

# STR $ (X)

Returns a string representation of the value of X.

Ex)
```
10 A=1957:B=3: C=6
20 PRINT STR$(A)+", "; STR$(B)+ ","+STR$(C)
30 END
run
 1957, 3, 6
OK
```

# STRING $

**STRING $ (< I >, < X$ >)**

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

Ex)
```
10 CLS
20 FOR I=3 TO 13
30 L=INT(RND(1)*20)
40 LOCATE 3, I
50 PRINT USING "##"; L
60 LOCATE 6, I
70 PRINT STRING$(L, "*")
80 NEXT
90 END
run
   11  * * * * * * * * * * *
    2  * *
   15  * * * * * * * * * * * * * * *
   11  * * * * * * * * * * *
   14  * * * * * * * * * * * * * *
    3  * * *
    7  * * * * * * *
   18  * * * * * * * * * * * * * * * * * *
   12  * * * * * * * * * * * *
    9  * * * * * * * * *
   16  * * * * * * * * * * * * * * * *
OK
```

# SWAP

**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.

Ex)
```
10 FOR I=0 TO 3
20 X(I)=INT(RND(1)*99)
30 Y(I)=INT(RND(1)*99)
40 NEXT:GOSUB 110
50 PRINT
60 FOR I=0 TO 3
70 SWAP X(I), Y(I)
80 NEXT
90 GOSUB 110
100 END
110 PRINT " I X(I) Y(I)"
120 PRINT
130 FOR I=0 TO 3
140 PRINT USING "#      ##      ##";I;X(I);Y(I)
150 NEXT:RETURN
run
 I   X(I)   Y(I)

 0   58     10
 1   75     57
 2   72     18
 3   36     94
 I   X(I)   Y(I)

 0   10     58
 1   57     75
 2   18     72
 3   94     36
OK
```

# 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.

Ex)
```
10 PRINT
20 PRINT "*", TAB (9) "*"
30 PRINT "*", "*"
40 PRINT "*" TAB (9) "*"
50 PRINT "*"; TAB (9); "*"
60 END
OK
run

*              *
*              *
*        *
*        *
OK
```

# TAN (X)

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

Ex)
```
10 FOR I=3.14 TO 6.28
20 A=TAN(I*3.14)
30 PRINT A
40 NEXT
run
 .46447027876367
 .46253546743286
 .46060350459682
 .45867437419176
```

# TIME

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.

Ex)
```
20 CLS
30 LOCATE 10, 8
40 PRINT "start !"
50 TIME=0
60 LOCATE 10, 10
70 T=TIME
80 H=INT(T/3600)
90 M=INT(T/60)
100 S=T MOD 60
110 PRINT USING "##   :  ## '   ## "; H;M;S
120 GOTO 60
run
          start !
           0 : 3' 45
```

# 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).

Ex)
```
10 PRINT "avt ";
20 PRINT "msx ";
30 PRINT "computer"
run
avt msx computer
OK
tron
OK
run
[10] avt [20] msx [30] computer
OK
troff
OK
run
avt msx computer
```

# USR

**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.

Ex)
```
10 CLEAR 200, &HEFFF
20 AD=&HF000
30 FOR I=AD TO AD+9
40 READ A$:A=VAL ("&h"+A$)
50 POKE I, A
60 NEXT I
70 DEF USR1=&HF000
80 INPUT A%
90 PRINT "A%=";A%
100 I=USR1(A%)
110 PRINT "i=";I
120 DATA 23, 23, 4e, 23, 46, 03, 27, 2b, 71, c9
run
? 1
A%= 1
i= 2
OK
run
? 9
A%=9
i=10
OK
run
? 99
A%=9
i=100
OK
```

# VAL (X$)

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

Ex)
```
10 FOR I=0 TO 30 STEP 5
20 PRINT "HEX"; HEX$(I);TAB (6)
"IS  ";VAL("&H"+HEX$(I))
30 NEXT I
40 END
RUN
HEX0    IS   0
HEX5    IS   5
HEXA    IS   10
HEXF    IS   15
HEX14   IS   20
HEX19   IS   25
HEX1E   IS   30
```

# VARPTR

**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.

Ex) PRINT HEX$ (VARPTR(A))

# VDP

**VDP (< n >)**

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.

Ex) A=VDP (8)

# VPEEK

**VPEEK (<address of VRAM>)**

Returns a value of VRAM specified. < address of VRAM > can be in the range of 0~16383.

Ex) A%=VPEEK(1)

# VPOKE

**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.

Ex) VPOKE 1%, & H F F

# WAIT

**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.

Ex) WAIT 1, & H 22, & H 22

# WIDTH

To set the width of display during text mode. Legal value is 1~40 in 40×24 text mode, 1~32 in 32×24 text mode.

Ex)
```
10 FOR I=5TO 30 STEP 5
20 WIDTH I
30 PRINT STRING$(I, "*")+"WIDTH";I
40 FOR J=0 TO 500:NEXT J
50 NEXT I
RUN
```

```
*****
WIDTH5
**********
WIDTH10
***************
WIDTH15
********************
WIDTH20
*************************
WIDTH25
******************************
WIDTH30
```

# SAMPLE PROGRAMS

## Addition

```
10    A=3
20    B=5
30    C=A+B
40    PRINT C
50    END
```

## The area of circle

```
10    INPUT R
20    M=3.14159*R*R
30    PRINT M
40    END
```

## Drawing of box

```
10    SCREEN 2
20    LINE (0,0)-(140, 100), 1, B
30    LINE (100, 80)-(255, 191), 2,BF
40    GO TO 40
```

## Subtraction from 100 to 1 step 3

```
10      FOR J=100 TO 1 STEP-3
20      PRINT J;
30      NEXT J
40      END
```

## Multiplication table

```
10      FOR K=1 TO 9
20      FOR J=1 TO 9
30      PRINT J*K;
40      NEXT J
50      PRINT
60      NEXT K
70      END
```

## Sounds of helicopter

```
10      FOR I=0 TO 13
20      READ A:SOUND I, A
30      NEXT
40      DATA 20, 0, 30, 0, 0, 9, 0
50      DATA 48, 16, 4, 6, 100, 2, 12
```

## Array

```
10    DIM X(10)
20    FOR J=1 TO 10
30    X(J) =J*2
40    NEXT J
50    FOR K=1 TO 10
60    PRINT X(K)
70    NEXT K
80    END
```

## Drawing of circle

```
10    SCREEN 2
20    P=3.14159
30    FOR T=0 TO 2*P STEP 0.1
40    X=30*COS (T)+40
50    Y=30*SIN (T)+48
60    LINE (X, Y)-(40, 48), 11
70    X1=X+80;Y1=Y
80    PSET (X1, Y1), 8
90    NEXT T
100   CIRCLE (40, 144), 30, 3
110   CIRCLE (120, 144), 30, 8
120   PAINT (40, 144), 3
130   GO TO 130
```

## LEFT$, RIGHT$

```
10    READ A$
20    PRINT LEFT$ (A$, 7)
30    PRINT RIGHT$ (A$, 4)
40    DATA "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
50    END
```

## MID$

```
10    READ A$, B$
20    PRINT MID$ (A$, 8, 7)
30    A$=LEFT$ (A$, 7) + B$
40    PRINT A$
50    DATA "I AM A TEACHER", "PIONEER"
60    END
```

## STRING$ (LEFT$, RIGHT$, MID$)

```
10    A$="THREE!&#$/STRING!&#$/FUNCTIONS"
20    L$=LEFT$ (A$, 5)
30    M$=MID$ (A$, 11, 6)
40    R$=RIGHT$ (A$, 9)
50    PRINT L$; " "; M$; " "; R$
60    END
```

# DRAW

```
10      SCREEN 2
20      PSET (220, 191), 10
30      DRAW "U190"
40      FOR I=189 TO 1 STEP-4
50      A$="L"+STR$(I)+"D"+STR$(I-1)+"R"+STR$(I-2)+"U"+STR$(I-3)
60      DRAW "XA$;"
70      NEXT I
80      GO TO 80
```

# PLAY

```
10    REM BACH
20    PLAY "T240L6V12", "T240L2V9"
30    PLAY "R8O6GABO7DCCED", "O4GO5GE"
40    PLAY "DGF#GD06BGAB", "O4BO5EO4E"
50    PLAY "O7CDEDCO6BGABG", "O4ABO5C"
60    PLAY "F#GADF#AO7CO6BA", "O5DF#D"
70    PLAY "BGA8O7DCCED", "CGC"
80    PLAY "DGF#GDO6BGAB", "O4BO5ED"
90    PLAY "EO7DCO6BAGDGF#G2", "CC#DG"
100   IF PLAY (0) THEN 100
110   PRINT "*** THE END ***"
```

# STRING$

```
10    CLS
20    FOR I=3 TO 13
30    L=INT (RND (1)*20)
40    LOCATE 3, I
50    PRINT USING "##";L
60    LOCATE 6, I
70    PRINT STRING$ (L, "*")
80    NEXT
90    END
```

## Plottine sine curve

```
10      REM A SINE WAVE
20      FOR T=0 TO 12.5 STEP  .25
30      A=INT (16+15*SIN(T))
40      PRINT TAB(A);"SINE"
50      NEXT T
60      END
```

## Truth table : (AUB) ∩ ($\bar{C}$)

```
10      PRINT "  A  B  C  X"
20      FOR I=1 TO 8
30      READ A, B, C
40      X=(A OR B) AND (NOT C)
50      PRINT
60      PRINT USING "##"; A, B, C, X
70      NEXT I
80      END
90      DATA 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0
100     DATA 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0
```

# The area and perimeter of a triangle

```
10    PRINT "THE LENGTHS OF THE  ";
20    PRINT "SIZES OF A"
30    INPUT "TRIANGLE  "; A, B, C
35    PRINT
40    P=A+B+C
50    PRINT "PERIMETER=";P
60    P=0.5*P
70    S=SQR (P*(P-A)*(P-B)*(P-C))
80    PRINT "AREA=";S
90    END
```

# Calculation of $\pi$ (Monte-Carlo Method)

```
10    FOR B=1 TO 8
20    FOR A=1 TO 100
30    X=RND (1)
40    Y=RND (1)
50    Z=X*X+Y*Y
60    IF Z<1 THEN S=S+1
70    N=N+1
80    P=4*S/N
90    NEXT A
100   PRINT P
110   NEXT B
120   END
```

## Arithmetic mean

```
 10    M=0:N=0
 20    READ A
 30    IF A=-999 THEN 70
 40    N=N+1
 50    M=M+A
 60    GO TO 20
 70    M=M/N
 80    PRINT "NUMBER OF"
 90    PRINT "SAMPLES=";N
100    PRINT
110    PRINT "MEAN"; TAB (19);
120    PRINT "=" ;M
130    DATA 12, 25, 15, 0,-999
140    END
```

## Combination

```
 10   REM COMBINATION
 20   FOR N=0 TO 10
 30   PRINT N; "I"
 40   FOR R=0 TO N
 50   F=1
 60   FOR K=1 TO N
 70   F=F*K
 80   NEXT K
 90   FOR K=1 TO R
100   F=F/K
110   NEXT K
120   FOR K=1 TO N-R
130   F=F/K
140   NEXT K
150   PRINT F;
160   NEXT R
170   PRINT
180   NEXT N
190   END
```

# Expectation

```
10    DIM X(12), P(12)
20    S=0
30    FOR I=1 TO 11
40    READ X(I), P(I)
50    S=S+X(I)*P(I)
60    NEXT I
70    X1=S/4.5E+06
80    PRINT "EXPECTATION=";X1; "$"
90    END
100   DATA 10000000, 2, 100000, 88, 5000000, 3
110   DATA 4000, 132, 1000000, 4, 10000, 196
120   DATA 500000, 45, 100000, 90, 10000, 900
130   DATA 1000, 9000, 100, 900000
```

## Conversion (from base ten to another base)

```
10    DIM A(15)
20    INPUT "THE NEW BASE?";B
30    PRINT "FIRST AND LAST ";
40    PRINT "NUMBER TO"
50    INPUT "CONVERT?"; F,L
60    FOR I=F TO L
70    PRINT
80    GOSUB 160
90    REM PRINT A TABLE ENTRY
100   PRINT  I; TAB (7);
110   FOR D=J TO 1 STEP -1
120   PRINT  A(D);
130   NEXT D
140   NEXT I
150   END
160   I1=I
170   J=1
180   Q=INT (I1/B)
190   R=I1-Q*B
200   I1=Q
210   A(J)=R
220   J=J+1
230   IF Q > =B THEN 180
240   A(J)=Q
250   RETURN
```

## Graphic (1)

```
10     SCREEN 2
20     Y1=30:Y2=0:PI=3.141592#
30     PSET (190, 0), 11
40     FOR T=0 TO 6*PI STEP PI/25
50     X=60*COS(T):GX=126+X
60     Y1=Y1+1:Y2=Y2+1
70     LINE-(GX, Y2), 11
80     LINE (126, Y1)-(GX, Y2), 11
90     NEXT T
100    GO TO 100
```

## Graphic (2)

```
10     SCREEN 2
20     B=-540 : E=-270 : C=2 : GOSUB 100
30     B=-270 : E=-180 : C =13 : GOSUB 180
40     B=-180 : E=90 : C=14 : GOSUB 100
50     B=90 : E=270 : C=6 : GOSUB 100
60     B=270 : E =450 : C=10 : GOSUB 100
70     B=450 : E=540 : C=15 : GOSUB 100
80     GOTO 80
100      FOR I=B TO E STEP 9
110      V1=I*P/180 : V2=I*P/540
120      X=COS(V2)*120*(I+540)/1000+124
130      D=256-X
140      Y=COS(V2)*45+96+SIN(V1)*45
150      LINE (124, 96)-(D,Y), C
160      LINE (132, 96)-(X,Y), C
170      NEXT
180      RETURN
```

## Graphic (3)

```
 10   SCREEN 2
 20   SC=1.16:CX=128:CY=96
 30   C=COS(.1):S=SIN(.1)
 40   FOR T=0 TO 6.3 STEP 0.05
 50   R=88*COS(2*T)
 60   COLOR 11
 70   X=CX+SC*R*COS(T)
 80   Y=CY+R*SIN(T)
 90   LINE (128, 96)-(X, Y)
100   NEXT T
110   COLOR 15
120   FOR R=90 TO 95 STEP 0.8
130   X=R:Y=0
140   FOR I=0 TO 63
150   T=X*C-Y*S
160   Y=X*S+Y*C
170   X=T
180   IF I=0 THEN PSET(CX+SC*X, CY-Y)
190   LINE -(CX+SC*X, CY-Y)
200   NEXT I:NEXT R
210   GO TO 210
```

# Clock

```
10      REM CLOCK
20      SCREEN 1:CLS
30      COLOR 10, 1, 7
40      LOCATE 3, 5:PRINT "WHAT TIME IS IT NOW?"
50      LOCATE 6, 7:PRINT "FOR EXAM. 12:30= > 12, 30"
60      LOCATE 10, 9:INPUT H, M
70      PI=3. 142:SCREEN 2, 2, 1
80      LINE (0, 0)-(255, 191), 14, B
90      CIRCLE (127, 96), 85, 14
100       GOSUB 270
110       PAINT (5, 5), 14, 14
120       REM MAIN
130       S=0 : TIME=0
140       S=INT (TIME/60)
150       IF S=60 THEN S=0 : M=M+1 : TIME=0 : GOSUB 240
160       IF M=60 THEN M=0 : H=H+1
170       IF H=13 THEN H=1
180       PSET (127, 96), 13
190       X=PI*(1-M/30)
200       Y=PI*(1-(H*60+M)/360)
210       LINE (127, 96)-(64*SIN(X)+127, 55*COS(X)+96),15
220       LINE (127, 96)-(45*SIN(Y)+127, 35*COS(Y)+96),15
230       GOTO 140
240       LINE (127, 96)-(64*SIN(X)+127, 55*COS(X)+96),1
250       LINE (127, 96)-(45*SIN(Y)+127, 35*COS(X)+96),1
260       RETURN
270       FOR K=0 TO 360 STEP 30
280       X1=127+70*COS(PI*K/180)
290       Y1=96+64*SIN(PI*K/180)
300       PSET (X1, Y1), 7
310       NEXT K
320       RETURN
```

# UFO

```
10      REM UFO IN THE CITY
20      SCREEN 2, 1 : COLOR 15, 1, 1 : CLS
30      DATA 18, 3c, 24, 7e, db, 7e, 24, 00
40      S$=" " : FOR I=1 TO 8 : READ A$:A=VAL ("&H"+A$):
                 S$=S$+CHR$(A):NEXT I
50      SPRITE$(1)=S$
60      FOR I=1 TO 100
70      PSET (RND(1)*255, RND(1)*191), RND(1)*15
80      NEXT I
90      LINE (0, 120)-(40, 191), 15, BF
100      LINE (48, 152)-(104, 191), 7, BF
110      FOR X=8 TO 24 STEP 16
120      FOR Y=128 TO 176 STEP 16
130      LINE (X, Y)-(X+8, Y+8), 8, BF
140      NEXT Y
150      NEXT X
160      FOR X=56 TO 88 STEP 16
170      FOR Y=160 TO 176 STEP 16
180      LINE (X, Y)- (X+8, Y+8), 9, BF
190      NEXT Y
200      NEXT X
210      CIRCLE (224, 40), 16, 10
220      PAINT(224, 40), 10, 10
230      X=120 : Y=88
240      FOR I=0 TO 500
250      X=X+RND(1)*10-5 : Y=Y+RND(1)*8-4
260      PUT SPRITE 1, (X, Y), 6, 1
270      FOR T=0 TO 10 : NEXT T
280      NEXT I
290      COLOR 15, 4, 7 : END
```

# Football game

```
10    SCREEN 2, 2:COLOR 15, 13, 13:CLS
20    DRAW "bm165, 70;m158, 72;m150, 50;m156, 44;m160,
      48;m182, 36;m201, 36;m219, 50;m206, 68;m189, 53"
30    DRAW "bm160, 56;m184, 102;m194, 98;m213, 72;m209, 66"
40    DRAW "bm216, 53;m229, 69;m202, 104;m192, 112;m188,
      107;m189, 104;m185, 105;m184, 102"
50    DRAW "bm160, 71;m168, 93;m161, 97;m157, 95;m154,
      96;m157, 100;m152, 101;m157, 109;m162,
      108;m163, 107;m181, 98"
60    DRAW "bm156, 44;m156, 50;m160, 48;m163, 51;m182, 36
70    DRAW "bm213, 89;m222, 83;bm220, 81;m237, 101;m198
      138;m179, 122;m189, 109;bm205, 132;m211, 137;m237,
      101"
80    DRAW "bm179, 122;m165, 137;m207, 180;m216, 169;m194,
      141;m197, 138;bm218, 169;m220, 167;m220, 167;m226,
      170;m213, 191;m200, 191;bm195, 166;m206, 155"
90    CIRCLE (203, 165), 6, ,. 7, 4, 5
100   DRAW "bm210, 135;m197, 144;bm196, 167;m207, 156"
110   DRAW "bm169, 141;m128, 171;m137, 182;m181,
      154;bm143, 160;m153, 171;bm137, 182;m137, 183;m131,
      187;m113, 165;m118, 160;bm144, 159;m154, 170"
120   CIRCLE (123, 165), 7,, 5.5, 2.6
130   DRAW "bm152, 40;m168, 40;m176, 32;m176, 16;m144,
      16;m144, 32;m152, 40;bm152,36;m162, 36;bm144,
       22;m150, 24;bm162, 24;m170, 22;bm156, 24;m148,
       32;m156, 32"
140    CIRCLE (162, 24), 20,,. 5, 2.9
150    DRAW "bm180, 16;m179, 26;m175, 32;bm176, 20;m179,
       24;bm155, 40;m156, 44;bm174, 35;m176, 37"
160    PSET (147, 26);PSET (164, 26)
170    PAINT (160, 10):PAINT (210, 184):PAINT (128, 180)
180    CIRCLE (57, 109), 23
 90    DRAW "bm56, 96;m64, 96;m67, 102;m59, 108;m53, 101; m56,
       96":PAINT (60, 100)
```

```
200     DRAW "bm59, 113;m66, 117;m64, 124;m56, 125;m53,
        116;m59, 113":PAINT (60, 120)
210     DRAW "bm40, 101;m47, 105;m47, 111;m40, 113;m35,
        107;m40, 101":PAINT (40, 108)
220     DRAW "bm78, 102;m73, 105;m73, 111;m78, 114":PAINT
        (75, 108)
230     DRAW "bm41, 95;m40, 101;bm50, 88;m56, 96;bm64,
        96;m69, 90;bm48, 104;m53, 101;bm66, 102;m73, 105;bm48,
        112;m53, 117;bm59, 107;m60, 112;bm66, 117;m72, 128"
240     DRAW "bm40, 113;m39, 122;bm56, 125;m51, 129;bm64,
        125;m67, 128"
250     DRAW "bm73, 133;m99, 157;bm74, 131;m99, 153;bm77,
        129;m98, 146;bm80, 128;m91, 138"
260     GOTO 260
```

## Car race

```
10      SCREEN 0, 3:WIDTH 30:CLS:DIM E(22)
20      FORJ =0 TO 2
30      S$=" ":FOR I=0 TO 32 : READ A$:A=VAL ("&H"+A$)
        : S$=S$+CHR$(A) : NEXT I
40      SPRITE$(J)=S$:NEXT J
50      FOR I=0 TO 22:E(I)=10:NEXT I
60      C=110:E=10
70      FOR T=0TO 100
80      FOR I=0 TO 21:E(22-I)=E(21-I):NEXT I
90      E=E+INT(RND(2)*3)-1
100     IF E<1 THEN E=1
110     IF E<19 THEN E=19
120     E(0)=E
130     FOR I=0 TO 22
140     K$=INKEY$:IF K$=" " THEN 170
150     IF ASC (A$)=28 THEN C=C+1
160     IF ASC (A$)=29 THEN C=C-1
170     LOCATE 0, I:PRINT SPC (E(I)):
180     PRINT "]" ; SPC (8) ; "[" ; SPC (23-E(I))
190     PUT SPRITE 1, (C, 128), 6, 1
200     PUT SPRITE 2, (C, 128), 1, 2
210     FOR B=0 TO 3:IF C < (E(16+B)+1)*8-1 OR C >
        (E (16+B)+7)*8-2 THEN 250:NEXT B
220     NEXT I:NEXT T
230     PLAY "c", "e", "g":SCREEN 1, 3
240     LOCATE 10, 10:PRINT "GOOD !!!":END
250     PLAY "c", "c+", "d":SCREEN 1,3
260     LOCATE 10, 10:PRINT "NO GOOD !!!":END
270     DATA 07, 0f, 0f, 0f, 0f, 0f, 08, 07
280     DATA 0b, 0b, 0b, 0b, 0b, 08, 0f, 0f
290     DATA e0, f0, f0, f0, f0, f0, 10, 10
300     DATA d0, d0, d0, d0, d0, 10, f0, f0
310     DATA 00, 0b, 00, 10, 10, 10, 00, 00
320     DATA 00, 00, 00, 10, 10, 10, 00, 00
330     DATA 00, 00, 00, 08, 08, 08, 00, 00
340     DATA 00, 00, 00, 08, 08, 08, 00, 00
```

# APPENDICES

# Appendix A

## CONTROL CODE TABLE

| CODE (Decimal) | CODE (HEX) | FUNCTION | KEY |
|---|---|---|---|
| 0 | 00 | | |
| 1 | 01 | Header byte for Graphic character | CTRL + A |
| 2 | 02 | To move the cursor to the first character of previous word | CTRL + B |
| 3 | 03 | To terminate waiting for input | CTRL + C |
| 4 | 04 | | CTRL + D |
| 5 | 05 | To delete behind the current cursor position | CTRL + E |
| 6 | 06 | To move the cursor to the first character of next word | CTRL + F |
| 7 | 07 | To beep | CTRL + G |
| 8 | 08 | To delete the character to the left of the cursor | CTRL + H, BS |
| 9 | 09 | To move the cursor to the next horizontal tab position | CTRL + I, TAB |
| 10 | 0A | Line feed | CTRL + J |
| 11 | 0B | To move the cursor to the home position | CTRL + K |
| 12 | 0C | To move the cursor to the home position and clear the screen | CTRL + L |
| 13 | 0D | Carriage return | CTRL + M, RETURN |
| 14 | 0E | To move the cursor to last of the current line | CTRL + N |
| 15 | 0F | | CTRL + O |
| 16 | 10 | | CTRL + P |
| 17 | 11 | | CTRL + Q |
| 18 | 12 | To toggle the insert mode | CTRL + R, INS |
| 19 | 13 | | CTRL + S |
| 20 | 14 | | CTRL + T |
| 21 | 15 | To delete the current line from the screen | CTRL + U |
| 22 | 16 | | CTRL + V |
| 23 | 17 | | CTRL + W |
| 24 | 18 | | CTRL + X, SELECT |
| 25 | 19 | | CTRL + Y |
| 26 | 1A | | CTRL + Z |
| 27 | 1B | | ESC |
| 28 | 1C | To move the cursor to the right | → |
| 29 | 1D | To move the cursor to the left | ← |
| 30 | 1E | To move the cursor to the up | ↑ |
| 31 | 1F | To move the cursor to the down | ↓ |
| 127 | 7F | To delete the character of current cursor position | DEL |

# Appendix B

## CHARACTER CODE TABLE

| CODE (Decimal) | CODE (Hexa) | Chara cter | CODE (Decimal) | CODE (Hexa) | Chara cter | CODE (Decimal) | CODE (Hexa) | Chara cter | CODE (Decimal) | CODE (Hexa) | Chara cter |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | | 32 | 20 | | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 01 | ☺ | 32 | 21 | ! | b5 | 41 | A | 97 | 61 | a |
| 2 | 02 | ☻ | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | ♥ | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | ♦ | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | ♣ | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | ♠ | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | ▣ | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | ◘ | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | ○ | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | ◙ | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | ♂ | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | ♀ | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | ♪ | 45 | 2D | — | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | ♫ | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | ☼ | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | ┼ | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | ┴ | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | ┬ | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | ┤ | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | ├ | 52 | 34 | 4 | 84 | 54 | T | 116 | 76 | t |
| 21 | 15 | ┼ | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | │ | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | ─ | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | ┌ | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | ┐ | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | └ | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | ┘ | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | ☒ | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | ¦ |
| 29 | 1D | ╱ | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | ╲ | 62 | 3E | > | 94 | 5E | ∧ | 126 | 7E | ~ |
| 31 | 1F | ═ | 63 | 3F | ? | 95 | 5F | — | 127 | 7F | |

| CODE (Decimal) | CODE (Hexa) | Character | CODE (Decimal) | CODE (Hexa) | Character | CODE (Decimal) | CODE (Hexa) | Character | CODE (Decimal) | CODE (Hexa) | Character |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 | 80 | Ç | 160 | A0 | á | 192 | C0 | | 224 | E0 | $\alpha$ |
| 129 | 81 | ü | 161 | A1 | í | 193 | C1 | | 225 | E1 | $\beta$ |
| 130 | 82 | é | 162 | A2 | ó | 194 | C2 | | 226 | E2 | $\Gamma$ |
| 131 | 83 | â | 163 | A3 | ú | 195 | C3 | | 227 | E3 | $\pi$ |
| 132 | 84 | ä | 164 | A4 | ñ | 196 | C4 | ▣ | 228 | E4 | $\Sigma$ |
| 133 | 85 | à | 166 | A5 | Ñ | 197 | C5 | | 229 | E5 | $\sigma$ |
| 134 | 86 | å | 166 | A6 | ª | 198 | C6 | | 230 | E6 | $\mu$ |
| 135 | 87 | ç | 167 | A7 | º | 199 | C7 | | 231 | E7 | $\tau$ |
| 136 | 88 | ê | 168 | A8 | ¿ | 200 | C8 | | 232 | E8 | $\Phi$ |
| 137 | 89 | ë | 169 | A9 | ⌐ | 201 | C9 | | 233 | E9 | $\theta$ |
| 135 | 8A | è | 170 | AA | ¬ | 202 | CA | | 234 | EA | $\Omega$ |
| 139 | 8B | ï | 171 | AB | ½ | 203 | CB | ▨ | 235 | EB | $\delta$ |
| 140 | 8C | î | 172 | AC | ¼ | 204 | CC | ▧ | 236 | EC | $\infty$ |
| 141 | 8D | ì | 173 | AD | ¡ | 205 | CD | | 237 | EC | $\phi$ |
| 142 | 8E | Ä | 174 | AE | « | 206 | CE | | 238 | EE | $\in$ |
| 143 | 8F | Å | 175 | AF | » | 207 | CF | | 239 | EF | $\cap$ |
| 144 | 90 | É | 176 | B0 | Ã | 208 | D0 | | 240 | F0 | $\equiv$ |
| 145 | 91 | æ | 177 | B1 | ã | 209 | D1 | | 241 | F1 | $\pm$ |
| 146 | 92 | Æ | 178 | B2 | Ĩ | 210 | D2 | | 242 | F2 | $\geq$ |
| 147 | 93 | ô | 179 | B3 | ĩ | 211 | D3 | | 243 | F3 | $\leq$ |
| 148 | 94 | ö | 180 | B4 | Õ | 212 | D4 | | 244 | F4 | ⌠ |
| 149 | 95 | ò | 181 | B5 | õ | 213 | D5 | | 245 | F5 | ⌡ |
| 150 | 96 | û | 182 | B6 | Ũ | 214 | D6 | | 246 | F6 | $\div$ |
| 151 | 97 | ù | 183 | B7 | ũ | 215 | D7 | ▩ | 247 | F7 | $\approx$ |
| 152 | 98 | ÿ | 184 | B8 | Ĳ | 216 | D8 | Δ | 248 | F8 | ◙ |
| 153 | 99 | Ö | 185 | B9 | ĳ | 217 | D9 | ‡ | 249 | F9 | ● |
| 154 | 9A | Ü | 186 | BA | ¾ | 218 | DA | $\omega$ | 250 | FA | - |
| 155 | 9B | ¢ | 187 | BB | ∼ | 219 | DB | | 215 | FB | $\sqrt{}$ |
| 156 | 9C | £ | 188 | BC | ◇ | 220 | DC | | 252 | FC | n |
| 157 | 9D | ¥ | 189 | BE | ‰ | 221 | DD | | 253 | FD | 2 |
| 158 | 9E | Pt | 190 | BD | ¶ | 222 | DE | | 254 | FE | I |
| 159 | 9F | ƒ | 191 | BF | § | 223 | DF | | 255 | FF | |

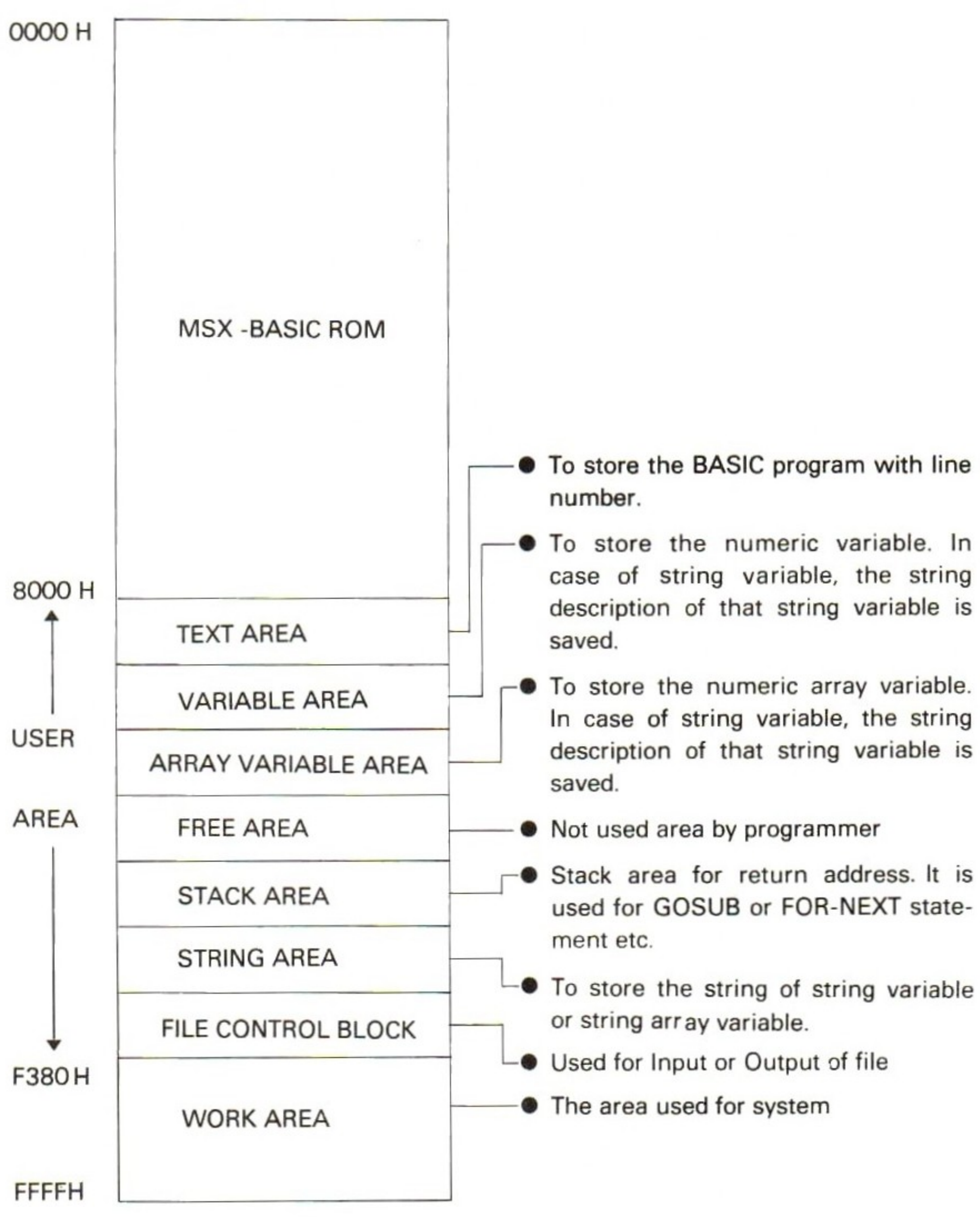
# Appendix C

## SLOT ARRANGEMENT

| | Slot 0 | Slot 1 | Slot 2 | Slot 3 |
|---|---|---|---|---|
| 0000H | | | | |
| PAGE 0 | BASIC-ROM (32KB) | | | |
| 4000 H | | | | |
| PAGE 1 | | | | |
| 8000 H | | | RAM (64KB) | |
| PAGE 2 | | | | |
| C 000 H | | | | |
| PAGE 3 | | | | |
| FFFF H | | | | |

# Appendix D

## Memory MAP



# Appendix E

## I/O MAP

| Address | Device |
|---|---|
| 00H | |
| 80H | * RS-232C |
| | |
| 90H | PRINTER |
| 98H | VDP |
| A0H | PSG |
| A8H | PPI |
| B0H | |
| D0H | * FDC |
| D8H | |
| FFH | |

* means optional device

| I/O ADDRESS | R/W | DESCRIPTION | REMARK |
|---|---|---|---|
| 98H | W | To write data to V-RAM | equivalent |
| | R | To read date from V-RAM | with 9129 |
| 99H | W | To write Command or Set Address | |
| | R | To read Status | |
| A0H | W | To latch Address | equivalent |
| A1H | W | To wirte data to PSG | with 8910 |
| A2H | R | To read data from PSG | |
| A8H | W | To wirte data to port A | equivalent |
| | R | To read data from port A | with 8255A |
| A9H | W | To write data to port B | |
| | R | To read data from port B | |
| AAH | W | To write data to port C | |
| | R | To read data from port C | |
| ABH | W | To set mode | |
| 90H | W | To output strobe signal (b0) | latch signal |
| | R | To read status signal (b1) | High when "Busy" |
| 91H | W | To output the print-out data | latch signal |

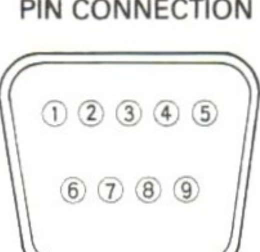
# Appendix F

## PINOUTS FOR INPUT/OUTPUT DEVICES

### Joystick

| PIN | SIGNAL |
|---|---|
| 1 | FORWARD |
| 2 | BACKWARD |
| 3 | LEFT |
| 4 | RIGHT |
| 5 | +5V |
| 6 | TRIGGER 1 |
| 7 | TRIGGER 2 |
| 8 | OUTPUT |
| 9 | GND |

PIN CONNECTION



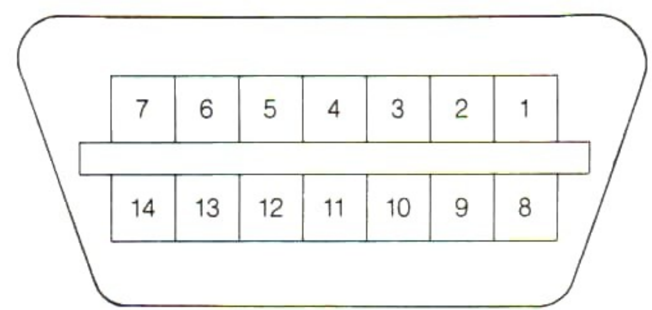

### Data Recorder Interface

| PIN | SIGNDL |
|---|---|
| 1 | GND |
| 2 | GND |
| 3 | GND |
| 4 | CASSETTE OUT |
| 5 | CASSETTE IN |
| 6 | REMOTE + |
| 7 | REMOTE - |
| 8 | GND |

# Printer Interface

| PIN | Type | Description |
|---|---|---|
| 1 | OUTPUT | PSTB |
| 2 | OUTPUT | PDBO |
| 3 | OUTPUT | PDB 1 |
| 4 | OUTPUT | PDB 2 |
| 5 | OUTPUT | PDB 3 |
| 6 | OUTPUT | PDB 4 |
| 7 | OUTPUT | PDB 5 |
| 8 | OUTPUT | PDB 6 |
| 9 | OUTPUT | PDB 7 |
| 10 | — | |
| 11 | INPUT | BUSY |
| 12 | — | |
| 13 | — | |
| 14 | — | GND |

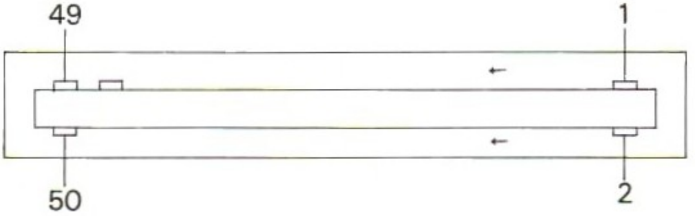
PIN CONNECTION



## Cartridge Bus

| PIN No. | NAME | I/O | PIN No. | NAME | I/O |
|---|---|---|---|---|---|
| 1 | $\overline{\text{CS1}}$ | O | 2 | $\overline{\text{CS2}}$ | O |
| 3 | $\overline{\text{CS12}}$ | O | 4 | $\overline{\text{SLTSL}}$ | O |
| 5 | Reserved *1 | — | 6 | $\overline{\text{RFSH}}$ | O |
| 7 | $\overline{\text{WAIT}}$ *2 | I | 8 | $\overline{\text{INT}}$ *2 | I |
| 9 | $\overline{\text{MI}}$ | O | 10 | $\overline{\text{BUSDIR}}$ | I |
| 11 | $\overline{\text{IORQ}}$ | O | 12 | $\overline{\text{MERQ}}$ | O |
| 13 | $\overline{\text{WR}}$ | O | 14 | $\overline{\text{RD}}$ | O |
| 15 | $\overline{\text{RESET}}$ | O | 16 | Reserved *1 | — |
| 17 | A9 | O | 18 | A15 | O |
| 19 | A11 | O | 20 | A10 | O |
| 21 | A7 | O | 22 | A6 | O |
| 23 | A12 | O | 24 | A8 | O |
| 25 | A14 | O | 26 | A13 | O |
| 27 | A1 | O | 28 | A0 | O |
| 29 | A3 | O | 30 | A2 | O |
| 31 | A5 | O | 32 | A4 | O |
| 33 | D1 | I/O | 34 | D0 | I/O |
| 35 | D3 | I/O | 36 | D2 | I/O |
| 37 | D5 | I/O | 38 | D4 | I/O |
| 39 | D7 | I/O | 40 | D6 | I/O |
| 41 | GND | — | 42 | CLOCK | O |
| 43 | GND | — | 44 | SW1 | — |
| 45 | +5V | — | 46 | SW2 | |
| 47 | +5V | — | 48 | + 12V | — |
| 49 | SUNDIN | I | 50 | - 12V | — |

*1:Not allowed to use
*2:Open Collector



| PIN No. | NAME | Description |
|---|---|---|
| 1 | CS1 | Select signal for address 4000H-7FFF H of ROM |
| 2 | CS2 | Select signal for address 8000H-BFFF H of ROM |
| 3 | CS12 | Select signal for address 4000H-BFFF H of ROM |
| 4 | SLTSL | Select signal for slot |
| 5 | Reserved | Reserved bus for future use |
| 6 | RFSH | Refresh cycle for dynamic RAM |
| 7 | WAIT | Wait request to CPU |
| 8 | INT | Interrupt request to CPU |
| 9 | M1 | Instruction fetch cycle of CPU |
| 10 | BUSDIR | Control signal for direction of external data bus buffer |
| 11 | IORQ | I/O request signal of CPU |
| 12 | MERQ | Memory request signal of CPU |
| 13 | WR | Write signal of CPU |
| 14 | RD | Read signal of CPU |
| 15 | RESET | Reset for system |
| 16 | Reserved | Reserved bus for future use |
| 17-32 | A0-A15 | Address bus |
| 33-40 | D0-D7 | Data bus |
| 41 | GND | Ground |
| 42 | Clock | Clock signal for CPU |
| 43 | GND | Ground |
| 44, 46 | SW1, SW2 | Switch for protection |
| 45, 47 | +5V | Power line. +5V |
| 48 | +12V | Power line. +12V |
| 49 | SUNDIN | Input for external sound |
| 50 | -12V | Power line. -12V |

# Appendix G

## ERROR MESSAGES AND ERROR CODES

Bad file name : 56

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

Bad file number : 52

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.

Can't continue : 17

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.

Device I/O error : 19

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.

Direct statement in file : 57

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

Division by zero : 11

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

FIELD Overflow : 50

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.

File already open : 54

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.

File not found : 53

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

File not OPEN : 59

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

Illegal direct : 12

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

Illegal function call : 5

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$, INSTP$ or ON~GOTO.

Input past end : 55

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.

Internal error : 51

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

Line buffer overflow : 25

An entered line has too many characters.

Missing operand : 24

An expression contained an operator with no operand following it.

NEXT without FOR : 1

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

No RESUME : 21

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

Out of DATA : 4

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

Out of memory : 7

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.

Out of string space : 14

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.

Overflow : 6

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

Redimensioned array : 10

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.

RESUME without error : 22

A RESUME statement is encountered before an error trapping routine is entered.

RETURN without GOSUB : 3

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

Sequential I/O only : 58

A statement to random access is issued for a sequential file.

String formula too complex : 16

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

String too long : 15

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

Subscript out of range : 9

An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

Syntax error : 2

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

Type mismatch : 3

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.

Undefined line number : 8

A line reference in a GOTO, GOSUB, IF~THEN~ELSE is to a nonexistent line.

Undefined user function : 18

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

Unprintable error : 23

An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

Unprintable errors : 26~49

These codes have no definitions. Should be reserved for future expansion in BASIC.

Unprintable error : 60~255

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

Verify error : 20

The current program is different from the program saved on the cassette.

## Appendix H

# MSX-BASIC RESERVED WORDS

BASIC statements and function names are reserved. That is, the key words cannot be used in variable names. This appendix lists all of the MSX-BASIC language words that are reserved. If you attempt to use any of the words listed below as the name of the variable, an error is indicated by the computer.

ABS
AND
ASC
ATN
ATTR$
AUTO
BEEP
BIN$
BLOAD
BSAVE
CALL
CDBL
CHR$
CINT
CIRCLE
CLEAR
CLOAD
CLOSE
CLS
CMD
COLOR
CONT
COPY
COS
CSAVE
CSNG
CSRLIN
CVD
CVI
CVS
DATA
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
DRAW
DSKF
DSKI$
DSKO$
ELSE
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FN
FOR
FPOS
FRE
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INP
INPUT
INSTR
INT
IPL
KEY
KILL
LEFT$
LEN
LET
LFILES
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
MAX
MERGE
MID$
MKI$
MKS$
MOD
MOTOR
NAME
NEW
NEXT
NOT
OCT$
OPEN
OR
OUT
OFF
ON
PAD
PAINT
PDL
PEEK
PLAY
POINT
POKE
POS
PRESET
PRINT
PSET
PUT
READ
REM
RENUM
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SCREEN
SET
SGN

SIN
SOUND
SPACE$
SQR
SPC
SPRITE
STEP
STICK
STOP
STRIG
STR$
STRING$
SWAP
TAB
TAN
THEN
TIME
TO
TROFF
TRON
USING
USR
VAL
VARPTP
VDP
VPEEK
VPOKE
WAIT
WIDTH
XOR

# Appendix I

# MATHEMATICAL FUNCTIONS

### Derived Functions

Functions that are not available in MSX-BASIC can be derived by using the following formulae:

| Function | MSX-BASIC Equivalent |
|---|---|
| SECANT | = 1/COS(X) |
| COSECANT | = 1/SIN(X) |
| COTANGENT | = 1/TAN(X) |
| INVERSE SINE | = ATN (X/SQR(-X*X + 1)) |
| INVERSE COSINE | = -ATN (X/SQR(-X*X + 1)) + 1.5708 |
| INVERSE SECANT | = ANT (X/SQR (X*X-1)) + SGN (SGN(X)-1)*1.5708 |
| INVERSE COSECANT | = ATN (X/SQR (X*X-1)) + (SGN(X)-1)*1.5708 |
| INVERSE COTANGENT | = ATN(X)+1.5708 |
| HYPERBOLIC SINE | = (EXP(X)-EXP (-X))/2 |
| HYPERBOLIC COSINE | = (EXP(X)-EXP (-X))/2 |
| HYPERBOLIC TANGENT | = (EXP(-X)/EXP(X)+EXP(-X))*2+1 |
| HYPERBOLIC SECANT | = 2/(EXP(X)+EXP(-X)) |
| HYPERBOLIC COSECANT | = 2/(EXP(X)-EXP(-X)) |
| HYPERBOLIC COTANGENT | = EXP(-X)/(EXP(X)-EXP(-X))*2+1 |
| INVERSE HYPERBOLIC SINE | = LOG(X+SQR(X*X+1)) |
| INVERSE HYPERBOLIC COSINE | = LOG(X)+SQR(X*X-1)) |
| INVERSE HYPERBOLIC TANGENT | = LOG((1+X)/(1-X))/2 |
| INVERSE HYPERBOLIC SECANT | = LOG ((SQR(-X*X+1)+1)/X) |
| INVERSE HYPERBOLIC COSECANT | = LOG ((SGN(X)*SQR (X*X+1)+1)/X) |
| INVERSE HYPERBOLIC COTANGENT | = LOG ((X+1)/(X-1))/2 |

# Appendix J

## TROUBLE SHOOTING CHART

| Symptom | Cause | Remedy |
| --- | --- | --- |
| No Power | Power switch not turned on | Make sure power switch is in "ON" position |
| | Power cable not pluged in | Check power socket for loose or disconnected power cable |
| | Bad fuse in computer | Take system to authorized dealer for replacement of fuse |
| No Picture | Video cable not plugged in | Check TV output cable connection |
| No Sound | TV Volume too low | Adjust volume of TV |
| No Color | Poorly tuned TV | Retune TV |
| | Bad color adjustment on TV | Adjust Color control on TV |
| Random Pattern on TV with cartridge in place | Cartridge not properly inserted | Reinsert cartridge after turning off power |
| Sound with excess background noise | TV volume up high | Adjust Volume of TV |