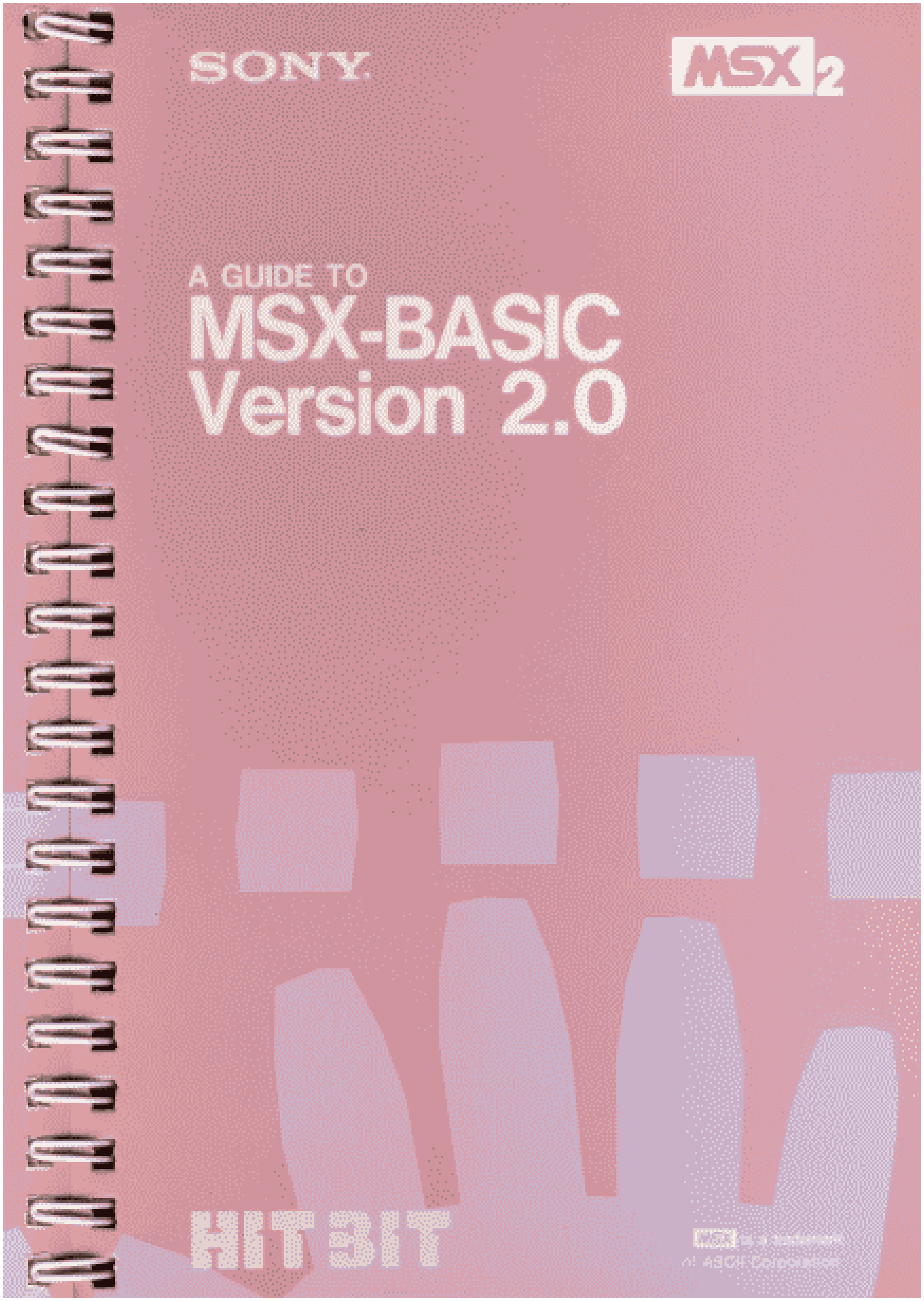


SONY

MSX2

A GUIDE TO

MSX-BASIC Version 2.0

HIT BIT

MSX is a trademark of ASCII Corporation

Scanned and converted to PDF by HansO, 2005

ABOUT THIS BOOK

This book is going to introduce you to MSX2-BASIC. It is divided into two sections—BEGINNING and ADVANCED.

The BEGINNING SECTION is written for those who are learning BASIC for the first time. It shows you what BASIC is all about and how to write simple BASIC programs. By practicing writing the programs in the BEGINNING SECTION, you will learn all of the most important commands used in writing BASIC programs.

If you are already familiar with MSX-BASIC, you can skip the BEGINNING SECTION and go on to the ADVANCED SECTION. Or if you haven't written programs in MSX-BASIC before, but have had experience with some other BASIC language, you can glance through the BEGINNING SECTION to see what are the main commands used in MSX-BASIC, and then go on to the ADVANCED SECTION.

MSX-BASIC Version 2.0

This book explains how to write programs in MSX-BASIC Version 2.0. MSX-BASIC Version 2.0 is a more powerful version of the MSX-BASIC used in the Sony HB-55P, HB-75P/B, HB-101P, and HB-201P computers. All of the commands and functions of MSX-BASIC are included in MSX-BASIC Version 2.0. Any program written in MSX-BASIC can be run on a computer that uses MSX-BASIC Version 2.0.

Commands or functions that are used only in Version 2.0 are referred to as **MSX2-BASIC** in this book. Commands or functions that apply to both MSX-BASIC and MSX2-BASIC are referred to simply as **BASIC**.

CONTENTS

INTRODUCTORY COURSE

Chapter 1 What is a Program?

Chapter 2 More Like A Computer

ADVANCED COURSE

Chapter 6 Sprite Patterns

Chapter 7 Using Functions

Chapter 8 Interrupts

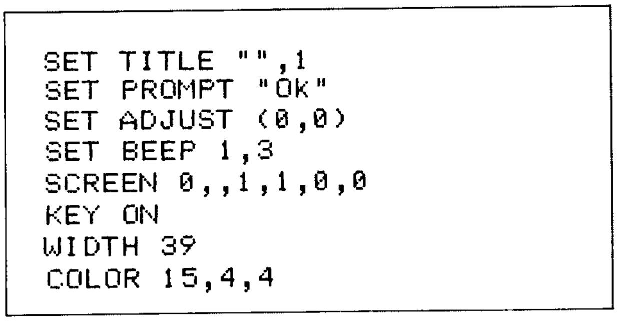
Chapter 9 Processing Files

Chapter 10 Machine Language Subroutines

INTRODUCTORY COURSE

Chapter 1
What Is A Program?

Execute the following statements first.



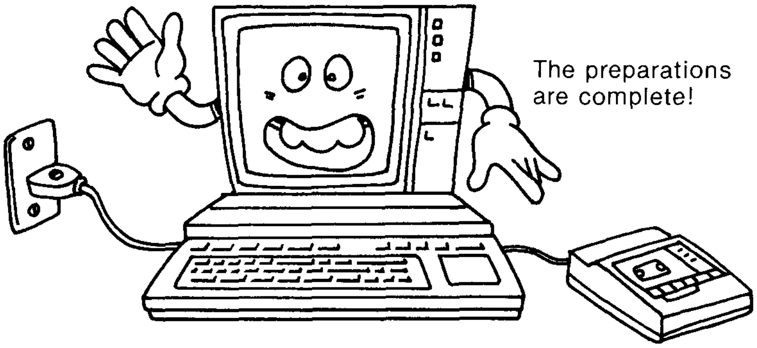
```
SET TITLE "",1
SET PROMPT "Ok"
SET ADJUST (0,0)
SET BEEP 1,3
SCREEN 0,,1,1,0,0
KEY ON
WIDTH 39
COLOR 15,4,4
```

YOUR FIRST COMPUTER COMMANDS

- ●How to Give Commands
- ●MSX-BASIC
- ●Doing Calculations—PRINT "expression"
- ●Variables and the LET Command
- ●Displaying Words—Print "character string"
- ●String Variables

GETTING READY

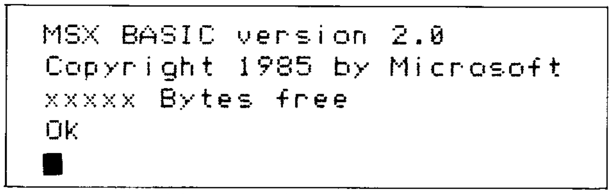
Do you have everything ready to begin learning how to make BASIC programs? Have you hooked up your computer and monitor TV? If not, refer to your computer manual for how to do it. If your computer doesn't have a built-in floppydisk drive, you will need to connect either a cassette tape recorder or a floppydisk drive. That's all the preparations you need.



If your monitor TV is a color monitor without a speaker, you should connect an external speaker, since sound plays an important part in operating a computer.

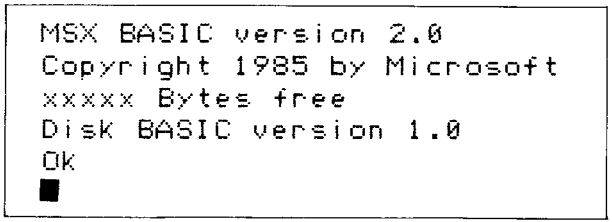
STARTING UP BASIC

When you have everything ready, turn on your computer following the directions in your computer manual. The procedure for starting up BASIC is also given in your manual. When you start BASIC, your screen will show one of these displays:



```
MSX BASIC version 2.0
Copyright 1985 by Microsoft
xxxxx Bytes free
Ok
■
```

Without a floppydisk drive



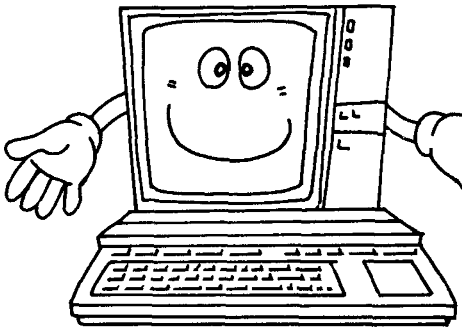
```
MSX BASIC version 2.0
Copyright 1985 by Microsoft
xxxxx Bytes free
Disk BASIC version 1.0
Ok
■
```

With a floppydisk drive

If, as shown above, the second line from the bottom displays Ok with a square mark below it, then BASIC is ready for you to use.

USING THE KEYBOARD TO ENTER COMMANDS

You use your computer's keyboard to give commands to the computer. By typing letters or numbers you can tell your computer to perform many different kinds of operations.



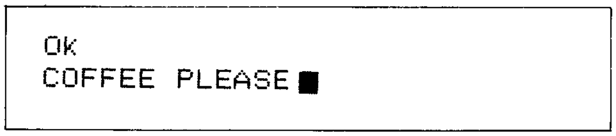
Input your
commands
here!

COMMAND 1—"COFFEE PLEASE"

Let's ask the computer for some coffee. Type the following letters on the keyboard.

[C] [O] [F] [F] [E] [E] [   ] [P] [L] [E] [A] [S] [E]

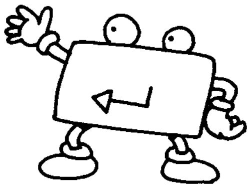
The empty [   ] space in the middle represents the **space bar**. When you hit a key, the character appears on your screen at the position where the square mark is located.
This square mark is called the **cursor**. It shows where the next character will be written.
If you typed your command, "COFFEE PLEASE" correctly, the screen should look like this:



```
Ok
COFFEE PLEASE■
```

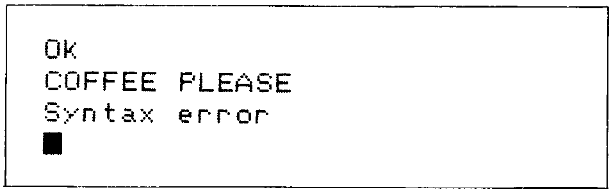
The next step is very important. After inputting your command, you must press the [↵] key (RETURN key). When you press this key, you tell the computer that you have completed inputting your command, and now the computer should execute it.

COFFEE PLEASE



"The command stops here!"

What happened to the screen when you hit the ↵ key?



```
Ok
COFFEE PLEASE
Syntax error
■
```

The instant you hit the ↵ key, you heard a BEEP!, and the "Syntax error" message was displayed. This is the computer's answer to your command "COFFEE PLEASE."

The "Syntax error" message means that you made a mistake in giving your command to the computer. The message was transmitted to the computer, but the computer didn't understand it.

BASIC

The computer can't understand a command like "COFFEE PLEASE," and therefore didn't give you any coffee. But there are many other commands that the computer does understand. These are the BASIC commands. If you use these BASIC commands to give commands to the computer, it will perform many interesting and even difficult feats for you. Let's take a look at some of these commands that the computer understands.

DOING CALCULATIONS PRINT expression

Let's give a command like the "COFFEE" command, which will have the computer add 10 + 5. Input the following characters:

[P] [R] [I] [N] [T] [ ] [1] [0] [+] [5] [↵]

The screen will look like this:

```
PRINT 10+5
 15
Ok
■
```

The computer followed the PRINT 10 + 5 command, and displayed the total, 15.

PRINT expression

PRINT tells the computer to display whatever comes after it on the screen. This is a BASIC command, and therefore one that the computer can easily understand. When you write an expression (math formula) after PRINT, this tells the computer to "do this calculation and display the answer on the screen."

Doing Various Calculations

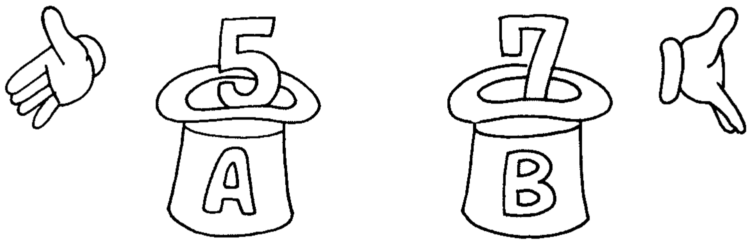
"10 + 5" is simple addition, but the computer can do other, more complicated, calculations, too. The following signs are BASIC calculating signs that the computer understands.

	Sign	Example	Meaning
Addition	+	PRINT 123 + 234	123 + 234
Subtraction	−	PRINT 300 − 125	300 − 125
Multiplication	*	PRINT 9 * 8	9 × 8
Division	/	PRINT 72/36	72 ÷ 36
Power	∧	PRINT 5∧3	5^3

If you want an operation in a formula to be performed first, you can enclose it in parentheses, ( ).

GIVING VALUES TO VARIABLES LET

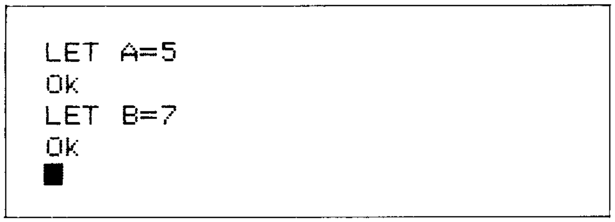
Let's put a 5 in the "A" hat, and a 7 in the "B" hat.



What would A + B be? It would be 5 + 7 = 12, wouldn't it.
Then what would A × B be? It would be 35.

Now let's do the same thing on the computer. You begin by putting 5 in "A" and 7 in "B." To do this type the following:

```
L E T     A = 5 ↵
L E T     B = 7 ↵
```



```
LET A=5
Ok
LET B=7
Ok
■
```

The computer only responds with "Ok", but inside the computer an operation just like putting the numbers in the hats has been performed.

LET variable = value

LET is the command used to put a number in a variable.

The A and B in the commands above are the same as the hats in our illustrations. A letter like A or B that is used to contain a number is called a **variable**.
Let's take another look at the above command.

```
LET A=5
```

The rule to remember in using the LET command is:

Put the variable letter on the left of the equal sign, and the number on the right of the equal sign.

Here the = sign doesn't mean "equals," as it does in arithmetic. Instead, you can think of it as meaning that "**the number on the right goes into the variable on the left**."

We gave the computer two commands.

```
LET A=5
LET B=7
```

Therefore, 5 has been placed in A, and 7 has been placed in B. Let's use the PRINT command to check and see if this really happened.

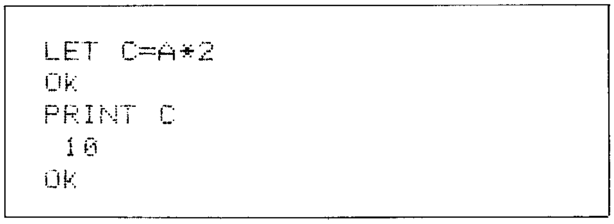
```
PRINT A
 5
Ok
PRINT B
 7
Ok
```

This is how you use the "PRINT expression" command. And in this command, you can see that it is all right to use **variables** as part of the expression. Let's try the following commands.

```
PRINT A+B
 12
Ok
PRINT A*B
 35
Ok
```

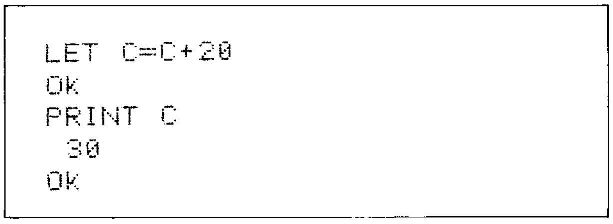
When we used only the variables in the expression part of the "PRINT expression" command, the computer used the numbers in the variables, and gave us the answers.

A Few Magic Tricks Using the LET Command



```
LET C=A*2
Ok
PRINT C
 10
Ok
```

Here we have used a new variable, C. Using the LET command, we made the value of C be the value of the formula A × 2. A is one of our previous variables, and we assigned the value of 5 to it. The value of C therefore became 10.
Let's now use C to get a new value.

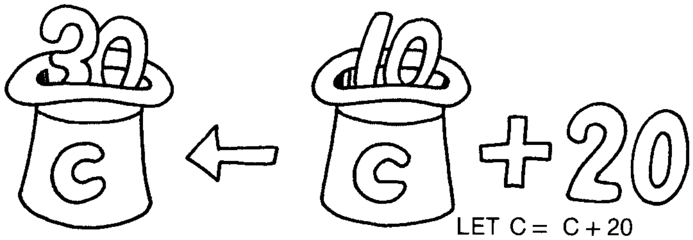


```
LET C=C+20
Ok
PRINT C
 30
Ok
```

Let's look at the first command you gave to the computer:

```
LET C=C+20
```

If you think of the = sign above as meaning "equals", then the formula looks a little strange. But remember that **the = sign means that what is on the right becomes the value of what is on the left**. Since the original value of C is 10, the formula is telling the computer to use this value, and then add 20 to it, to give C the new value of 30.



LET C= C+20

It might seem a little complicated at first, but once you understand the trick, it becomes very simple.

We've seen how to use three variables up to now—A, B, and C. Now we can make some general rules about how variables are used.

THE RULES FOR USING VARIABLES

- You can use as many letters as you want for a variable but the computer will only read the first two letters. Consequently, such variables as

  ABC, ABCD, ABD, AB1, AB2

  will all be read by the computer as the same variable:

  AB
- You cannot use a number or a sign as the first character of a variable. You can use A1, or C3, as variables, but you cannot use variables like 1AB or *AB.
- You cannot use functions or words that are used in BASIC as variables. Also, if a part of the variable contains such words, the computer will not accept it.

  Words like PRINT or LET, or PRINTA or ALET, cannot be used as variables.

Omitting LET

Variables are used very frequently in BASIC, and this means that the LET command is also used frequently. Because it is used so often, BASIC allows you to omit LET when you write a variable command. Instead of

```
        LET A = 10
```

you can simply write

```
        A = 10
```

Now let's make some other commands which uses the abbreviated form of the LET command.

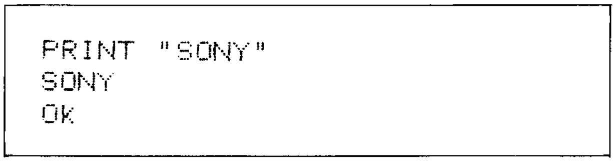
```
PRINT AB       ) No value has been given to AB yet.
 0             } So its value is still 0
Ok             )
                 The value 100 has been placed in
AB=100 ———————   the variable AB
Ok               (This is the same as LET AB = 100)
CD=200 ———————   The value 200 is placed in CD
Ok
X=AB+CD ——————   The result of adding AB and
                 CD (300) is placed in variable X
Ok
PRINT X        )
 300           } X is displayed
Ok
PRINT ABC      )
 100           } ABC is read as the same as AB,
Ok             ) so its value is 100
```

DISPLAYING CHARACTERS
PRINT “character string”

This command is used to display words on the screen. The same PRINT command is used for words, also. Try the following command.

[P] [R] [I] [N] [T] [   ] [“] [S] [O] [N] [Y] [”] [↵]

The quotation marks (”) are typed by holding down the shift key and hitting [2].

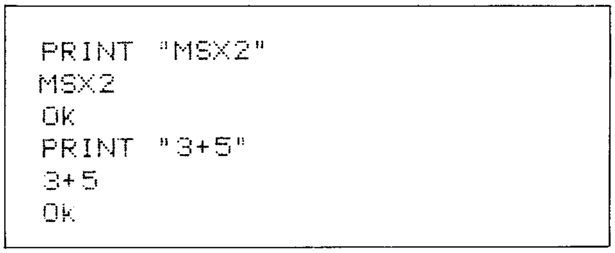


```
PRINT "SONY"
SONY
Ok
```

Any word you want to print can be displayed by enclosing it in “ ”, after the PRINT command. Since “SONY” is the word enclosed in quotation marks above, it is the word that is displayed. Words enclosed in “ ” are called a **character string**.

PRINT “character string”

Try putting different words inside the quotation marks. For example:



```
PRINT "MSX2"
MSX2
Ok
PRINT "3+5"
3+5
Ok
```

In the second PRINT command

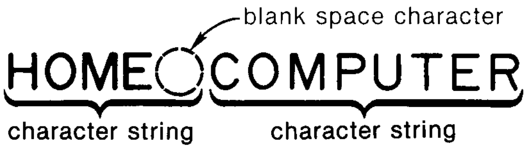
```
PRINT "3+5"
```

3 + 5 is an expression, but because it has quotation marks around it, it is treated like the three words “three plus five” instead of as an expression for adding. If you remove the quotation marks, then 3 + 5 would be calculated, and the answer 8 would be displayed. In making

a command for a computer, it is important to remember that **numbers can be treated as either numerical values or as letters, depending on whether or not quotation marks are used.**

```
PRINT "SO"+"NY"
SONY
Ok
PRINT "HOME "+"COMPUTER"
HOME COMPUTER
Ok
```

In the first example above, the character strings "SO" and "NY" are connected and displayed as the word "SONY." But in the second example, a blank space has been left after the letter E in "HOME". (The blank space is made by hitting the space bar once.) Leaving one space blank after E results in a space between the two character strings when they are joined. From this you can see that a blank space is treated as one character, also.

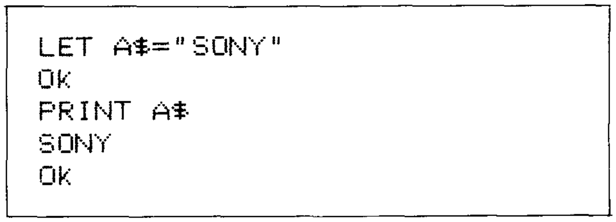


Now that we can make character strings like this, we need some hats (variables) to put them in.

USING THE $ MARK WITH STRING VARIABLES

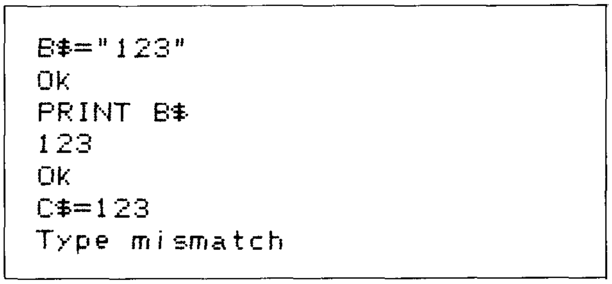
Before, we put numbers in variables such as A, B, and CD. We can also use variables to contain character strings. This kind of a variable is called a **string type variable** (or simply, string variable), to distinguish it from a variable which contains numbers, which is called a **numeric type variable** (or simply, numerical variable).
A $ sign is added to a variable to make it a string variable—A$, B$, or CD$. Variables that have a $ sign attached to them will only accept character strings as their contents. (If a number is being used as a word, then it also can be placed in a variable with the $ sign attached).

Let's practice using string variables.



```
LET A$="SONY"
Ok
PRINT A$
SONY
Ok
```

Here we used the LET command to put the character string "SONY" in the string variable A$. In writing the LET command, we could have omitted the word LET. Also, a character string must be enclosed in "   " in order to put it in a string variable.



```
B$="123"
Ok
PRINT B$
123
Ok
C$=123
Type mismatch
```

In the above example we tried to put the numerical value 123 into the string type variable C$. But instead the **"Type mismatch" error message** was displayed. This is the computer's way of telling you that you have tried to match two different types—like trying to put a numeric type value into a string type variable.

ERROR MESSAGE

When you give the computer the wrong command, it displays a message which tells you what kind of mistake you made. These messages are called **"error messages."** If the computer cannot understand the meaning of a command you give it, it will display

```
Syntax error
```

Or if you haven't matched the right types of values and variables, it displays

```
Type mismatch
```

to tell you what your mistake is.

MAKING A BASIC PROGRAM

- A One-Line Program
- Checking a Program—LIST
- Running a Program—RUN
- Erasing a Program in Memory—NEW
- Entering Variable Values From the Keyboard—INPUT
- Getting the Most From the PRINT Command
- Erasing One Part of the Program—DELETE
- To Erase the Display on the Screen—CLS
- Making a Loop—GOTO

In the previous section we learned how to give commands to the computer. When we input a command from the keyboard and pressed the ↵ key, the command was immediately executed.
But a computer is designed to perform many operations in succession, and executing a command as soon as you input it from the keyboard allows you to do only one operation at a time. This is where programs come in.

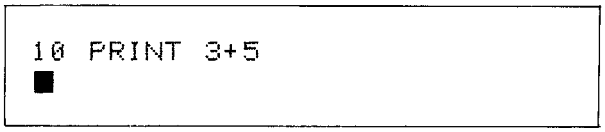
DIRECT MODE AND PROGRAM MODE

Entering just one command and then executing it by hitting the ↵ key, like we have done up to now, is called the **direct mode**. But there is another method of using the computer. It is called the **program mode**. Using the program mode makes the computer act like a computer should.

A ONE-LINE PROGRAM

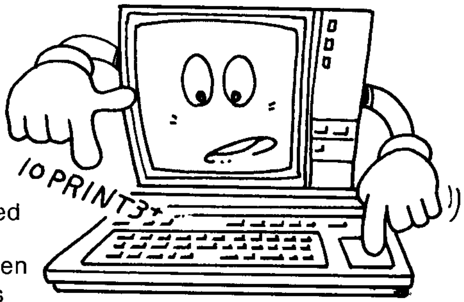
Now let's use the program mode to make a program. You don't have to do anything special to use the program mode. Just input the following characters.

1 0 ␣ P R I N T ␣ 3 + 5 ↵



```
10 PRINT 3+5
■
```

The command PRINT 3+5 means that the computer should add 3 and 5 and display the answer, 8. But when you hit the ↵ key after inputting the command, nothing was displayed. This was because you entered the 10 before the PRINT command. When a number is entered before a command, this tells the computer that instead of executing the command when the ↵ key is hit, it should remember the command in its memory. This kind of number is called a **line number**.



The program is remembered inside the computer when the ↵ key is pressed.

The command with the line number that the computer remembered is called a **program**.

CHECKING A PROGRAM LIST

The computer has remembered the program

```
10 PRINT 3+5
```

We can check to make sure of this by using

LIST

Input the following characters from the keyboard.

L I S T ↵

```
LIST
10 PRINT 3+5
Ok
```

LIST is the command that tells the computer to display the program list. You input it in the direct mode, without a line number in front of it.

RUNNING A PROGRAM RUN

In the direct mode the ↵ key was pressed to execute a command. In the program mode, the RUN command is used.

RUN

Input the following characters.

R U N ↵

```
RUN
 8
Ok
```

The program that has been remembered by the computer

```
10 PRINT 3+5
```

is now executed for the first time, and the answer, 8, is displayed on the screen. If you input the RUN command again, 8 will again be displayed. You can do this as many times as you want.

```
RUN
 8
Ok
RUN
 8
Ok
RUN
 8
Ok
```

What is a Program?
Let's compare

```
        PRINT 3+5
```

as it is executed in the direct mode and in the program mode.

	Direct Mode	Program Mode
Line no.	No	Yes
To execute	Press ↵ key	Input RUN command
Command remembered	No	Yes
No. of times executed	1 time only	Any number of times

Because a program uses line numbers, the computer remembers the program. And it can be executed any number of times using the RUN command.
Without programs, a computer would not be very useful to you.
Programs are procedures which can be written to have the computer perform a series of operations. They are called **software**. The computer itself is a machine designed to faithfully perform any of a given set of actions. In contrast to software, a computer itself is called **hardware**.

ERASING A PROGRAM IN MEMORY NEW

We now want to write a new program and have the computer remember it, but the computer still has the program

```
10 PRINT 3+5
```

in its memory. We therefore must execute the NEW command so that we can have the computer remember the new program.

NEW

The NEW command erases the entire program that is currently remembered in the computer's memory.

Input the NEW command, and then use the LIST command to check that the program has been erased.

N E W ↵

```
NEW
Ok
LIST
Ok
```

As you can see, the program is not displayed.

Ways To Erase The Memory

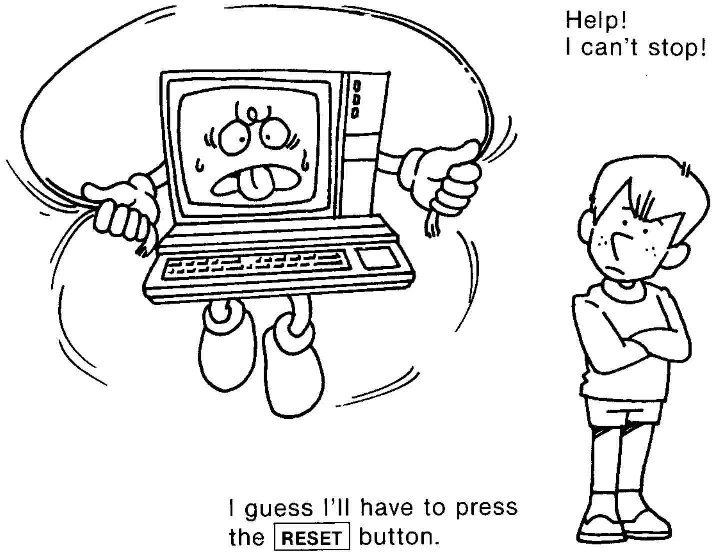
You can erase a program in the computer's memory by using the NEW command. The program will also be erased in the following situations.

- When the RESET button is pressed.
- When the computer is turned off.

The RESET button is provided for emergency use in case there is some problem with a program you have made, and you cannot stop the program no matter what key you press.

Pressing the RESET button is just like turning the computer off and on again. This action erases the program in the computer's memory and places the computer in the same condition as when it is first turned on.

Be careful not to press the RESET button or the on/off switch while you are writing a program.

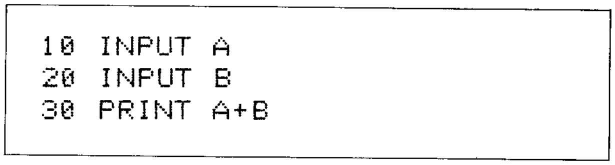


ENTERING VARIABLE VALUES FROM THE KEYBOARD INPUT

Let's now make a program in which you give a different value to A and B every time you execute it. When we do this, then each time the program is executed it will add different numbers. But for such a program we need to use a different command in place of the LET command—the INPUT command.

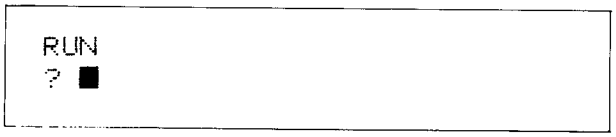
INPUT variable name

INPUT is the command used to enter a value in a variable from the keyboard.



```
10 INPUT A
20 INPUT B
30 PRINT A+B
```

Once you have input the lines, check the program like you always do using the LIST command to make sure that they have been remembered. Once the change is made, execute the program using the RUN command.



```
RUN
? ■
```

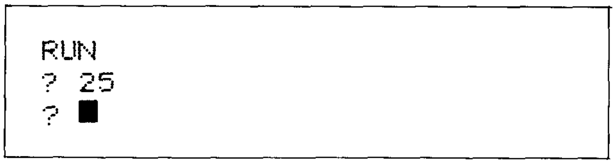
When you run the program, a question mark immediately appears below RUN, with the cursor beside it. This is the result of the

```
INPUT A
```

command in line 10. The computer is waiting for you to give it an instruction from the keyboard telling it what value to put in the variable A. You can input any number you want. Let's try 25.

2 5 ↵

Remember to press the [↵] key after typing 25. This tells the computer that the number it should place in the variable A is 25.

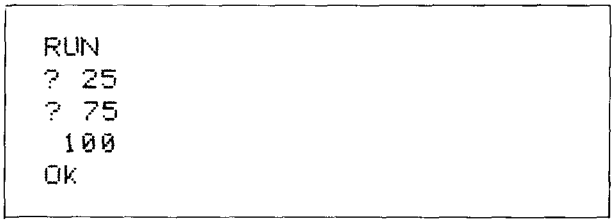


```
RUN
? 25
? ■
```

After you input 25, the question mark is again displayed. This is the result of the computer executing line 20. Let's enter 75 for variable B.

[7] [5] [↵]

When 25 is entered for A and 75 is entered for B, line 30 is executed and 100, the total of A + B, is displayed.

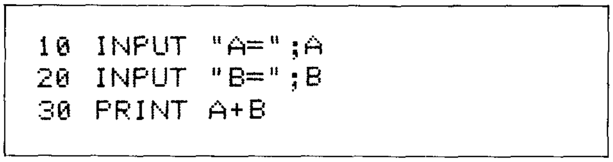


```
RUN
? 25
? 75
 100
Ok
```

Run the program again, and enter different values for A and B. You will see that the correct total is always displayed on the screen.

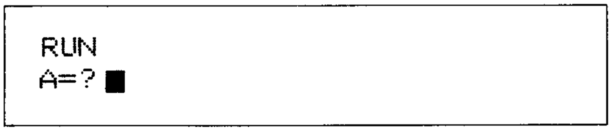
Making the INPUT Command More "User Friendly"

When this program is run, the only thing you see on the screen is a question mark and the cursor. If you are the one who wrote the program, you will know that this means that you are to enter the value for A. But someone else might not know what to do when they see just the question mark. To make the program more "user friendly" for other people, we can modify the INPUT command as follows.



```
10 INPUT "A=";A
20 INPUT "B=";B
30 PRINT A+B
```

When you execute the program after the above changes have been made, the screen will first display:



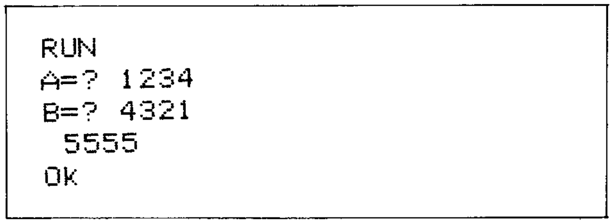
```
RUN
A=? ■
```

The character string that is enclosed in quotation marks before the variable name is called a **prompt statement**. The prompt statement is displayed in front of the question mark when the program is executed. When you use a prompt statement anyone will know immediately what they are being asked to input.

INPUT "prompt statement"; variable name

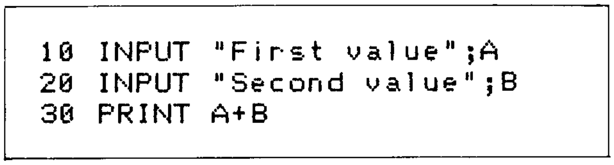
Remember that you have to type a semicolon (;) between the prompt statement and the variable name.

Let's run the whole program.



```
RUN
A=? 1234
B=? 4321
 5555
Ok
```

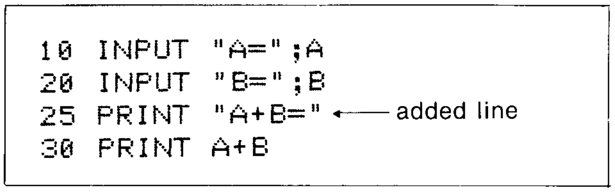
You can use any words you want for the prompt statement. For example:



```
10 INPUT "First value";A
20 INPUT "Second value";B
30 PRINT A+B
```

GETTING THE MOST FROM THE PRINT COMMAND

With the change we made in the above program, the computer is beginning to act more and more like a computer. But there are still other things we can do to make what is displayed on the screen easier to use. Right now the program displays just the total of A and B, and doesn't tell us what it is. It would be better if the computer told us that the number being displayed was the total of A + B. This can be done by using the PRINT command to add a line to the program as follows.

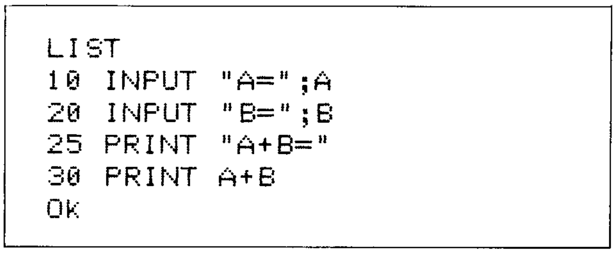


```
10 INPUT "A=";A
20 INPUT "B=";B
25 PRINT "A+B="
30 PRINT A+B
```

Line 25 was added to the program. A line can be added to a program anytime "Ok" is displayed. You type the line in just like you do when you type a line in a new program.

[2] [5] [ ] [P] [R] [I] [N] [T] [ ] [“] [A] [+] [B] [=] [”] [↵]

Input the LIST command to check that line 25 has been added to the program.



```
LIST
10 INPUT "A=";A
20 INPUT "B=";B
25 PRINT "A+B="
30 PRINT A+B
Ok
```

Line 25 has been added between line 20 and line 30, just where we wanted it. It doesn't make any difference what order you enter the lines of a program. The computer automatically rearranges them in the proper order, from the lowest numbered line to the highest numbered line.

When you added line 25 you probably guessed why 10, 20, and 30 had been used as the line numbers for the other lines. This leaves room for adding new lines later, such as line 25.

After adding line 25, the computer will now display "A + B = " before it displays the total of A and B. Let's run the program and see what the display looks like.

```
RUN
A=? 100
B=? 50
A+B=
 150
```

"A + B = " has been displayed before the answer of 150, but it would look better if both A + B = and 150 were displayed on the same line, like this.

```
A+B= 150
```

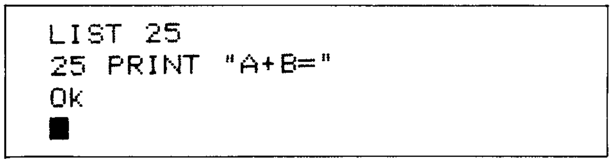
Using the Semicolon (;) to Connect What is Displayed
Let's change line 25 as follows:

```
25 PRINT "A+B=";
                ↑———————— this is added
```

A semicolon (;) has been added after the final quotation mark. You use the following procedure to make this revision. First input the LIST command to display line 25.

```
LIST 25
```

Don't forget to press the [↵] key after typing the command.



```
LIST 25
25 PRINT "A+B="
Ok
■
```

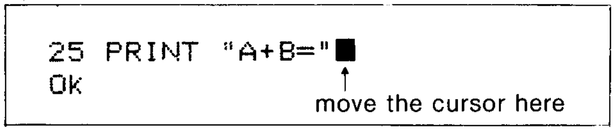
LIST line number

You can use the LIST command to display just one line of a program by typing the line number after LIST. Also, if you type
    LIST line number—line number
you can display just one part of the program. For example,

```
LIST 20-30
```

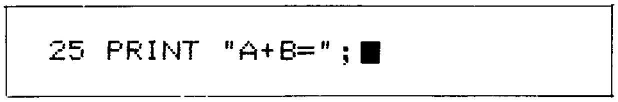
would display lines 20 through 30 of a program. For our program, this would mean that lines 20, 25, and 30 would be displayed. Try it and see.

After you have displayed line 25 with the LIST command, use the cursor keys (✥) to move the cursor (the ■ mark) to a position right after the last character (") in line 25, which is where you want to add the semicolon.



```
25 PRINT "A+B="■
Ok
```

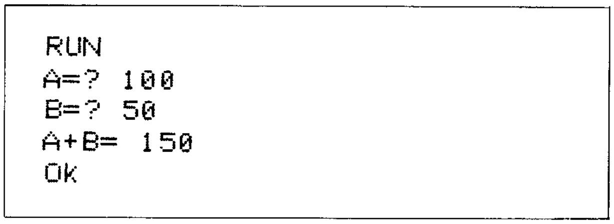
Then press the [;] key



```
25 PRINT "A+B=";■
```

The semicolon has been displayed on the screen, but that's not the end of the operation. To enter the change you have to press the [↵] key. When you wrote the original program, you pressed the [↵] key after each line. When you revise a line you also must remember to press it. It is only by pressing the [↵] key that the revised line is entered into the computer's memory.

Use the LIST command to check that line 25 has been revised, and then execute the program using the RUN command. Adding the semicolon has caused the total to be displayed on the same line as A + B = .



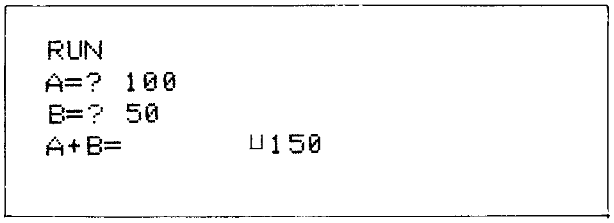
```
RUN
A=? 100
B=? 50
A+B= 150
Ok
```

Using a comma (,) instead of a semicolon (;)

Next, let's use a comma in place of the semicolon in line 25.

```
25 PRINT "A+B=",
                 ↑___ put the comma here
```

After replacing the semicolon with a comma, RUN the program.



```
RUN
A=? 100
B=? 50
A+B=          ␣150
```

Now the 150 is positioned 16 characters (the 15th space is for a minus sign) after the beginning of the line, instead of right after A + B = .

ERASING ONE PART OF THE PROGRAM DELETE

Lines 25 and 30 in our program are currently:

```
25 PRINT "A+B=",
30 PRINT A+B
```

We can combine these two lines into one line.

```
25 PRINT "A+B=",A+B
```

Revise your program to combine the two lines, using the same procedures you used when you changed the semicolon to a comma.

Now we don't need line 30 anymore. Let's erase it from the program. There are two ways of removing one line from a program. One is to use the DELETE command.

```
DELETE 30
```

The DELETE command deletes specified lines from a program. The normal form of this command is:

DELETE line no.-line no.

After typing DELETE you specify the first line and the last line of the part of the program you want to erase, separating them by a hyphen (-).
When you just want to delete one line from a program, you don't have to use the DELETE command. You can just type

3 0 ↵

TO ERASE THE DISPLAY ON THE SCREEN CLS

The way your program is written now, each time you run it the lines advance on the screen, and the result is displayed. The lines that were displayed when you ran the program the previous time also remain displayed. It would be easier to see what you are doing if the screen were cleared each time before the calculation. The **CLS command** is provided for this job. (CLS is read "clear screen".)

CLS

Let's add the CLS command to our program. We can make it line 5.

```
LIST
5 CLS
10 INPUT "A=";A
20 INPUT "B=";B
25 PRINT "A+B=",A+B
```

With line 5 added to the program, the screen is cleared each time you execute the program, which makes it easier to see what you are doing.

MAKING A LOOP GOTO

Loop Using the GOTO Command

We will add a new command to the program.

```
30 GOTO 10
```

Run the program and see what happens after adding the new line.

GOTO line no.

GOTO tells the computer to go to (or return to) the specified line number.

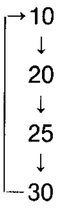
Normally the program is executed starting with the smallest line number. But the GOTO command can be used to change the flow of the program.

Therefore, the command

```
GOTO 10
```

returns the flow of the program to line 10, and the program continues from there. Anytime you want the computer to jump to a different part of the program, you can specify the line number using the GOTO command.

The reason why the program keeps on running is because the GOTO 10 command in line 30 returns the program to line 10, and then after lines 10, 20, and 25 have been executed the flow of the program comes to line 30 again, and is again returned to line 10, and this goes on and on indefinitely. The program continues to perform the following sequence:



The only way this repetition can be stopped is to press CTRL + STOP.

A part of a program that is repeated is called a **loop**. And when the repetition is repeated over and over again, as in our program, it is called an **endless loop**. The GOTO command was what produced the endless loop.

This chapter has served to introduce programs and give you a feel for how they work. In the next chapter we will introduce some new commands, and practice making more advanced programs.

Chapter 2
More Like A Computer

ESCAPING FROM A LOOP

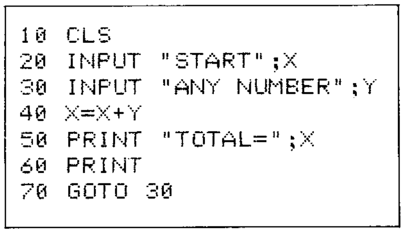
- Satisfying a Condition—IF—THEN
- To End the Program—END
- Renumbering Lines—RENUM

SATISFYING A CONDITION IF—THEN

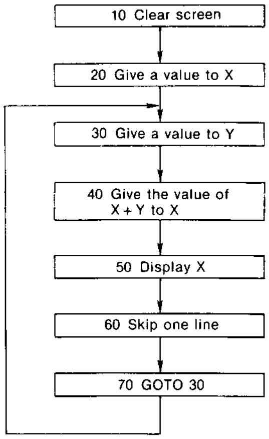
The Endless Loop

When an endless loop program is started with the RUN command, it will normally continue to run over and over again, until you stop it with the CTRL+STOP keys. Here we will show how to get out of an endless loop without using the CTRL+STOP keys. When you escape from an endless loop by hitting the CTRL+STOP keys, the program will stop. But with the method we describe here, the program continues to run after you have escaped from the endless loop.

First, let's make a simple endless loop program.



```
10 CLS
20 INPUT "START";X
30 INPUT "ANY NUMBER";Y
40 X=X+Y
50 PRINT "TOTAL=";X
60 PRINT
70 GOTO 30
```



Stopping the Program On the Basis of the Value of Y

Every day we do all sorts of actions on the basis of some kind of conditions. If it's a nice day, then we will have the school athletic meet, but if it rains, then it will be postponed. Or if we have some money, then we will go to the movies, but if we don't have any money, then we will stay home and take a nap.

There are all sorts of times when we will do one thing if there is such-and-such a condition, but do something else if the condition is different.

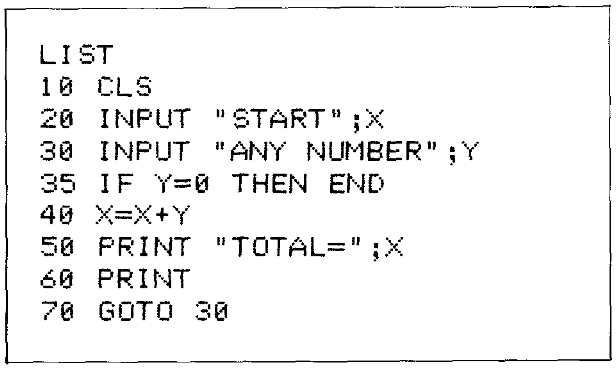
We can also make the computer act on the basis of such conditional decisions. The **IF—THEN** command is the command used to have the computer perform the next command if a certain condition is met. Actually, this type of command, which gives an order to the computer, is normally called a "**statement**," as in the "**IF—THEN statement**." Such commands as **PRINT**, **LET**, or **INPUT** are all commands used to make a statement, so they are referred to as the **PRINT statement**, the **LET statement**, etc.

IF conditional expression THEN statement

IF—THEN tells the computer that if the conditional expression that follows **IF** is true, then it should perform the statement that follows **THEN**.

Let's add the following **IF—THEN** statement to the program we have written.

```
35 IF Y=0 THEN END
```



```
LIST
10 CLS
20 INPUT "START";X
30 INPUT "ANY NUMBER";Y
35 IF Y=0 THEN END
40 X=X+Y
50 PRINT "TOTAL=";X
60 PRINT
70 GOTO 30
```

The meaning of line 35 which we added is "if Y is 0, then stop the program."

TO STOP THE PROGRAM END

In line 35, the conditional expression is Y=0, and END is the statement.

END

END is the statement used to stop a program. When this statement is executed, the program stops at that point.

CONDITIONAL EXPRESSIONS

The following list shows the signs which can be used to make **conditional expressions**.

Sign	Meaning	Example	
=	equals	IF A=B	if A and B are equal
>	is greater than	IF A>B	if A is greater than B
<	is less than	IF A<B	if A is less than B

> =
= >
	is greater than or equal to	IF A> =B
		IF A= >B
			if A is greater than or equal to B

< =
= <
	is less than or equal to	IF A< =B
		IF A= <B
			if A is less than or equal to B

< >
> <
	is not equal to	IF A< >B
		IF A> <B
			if A and B are *not equal*

The sign used in the conditional expression in Line 35 is

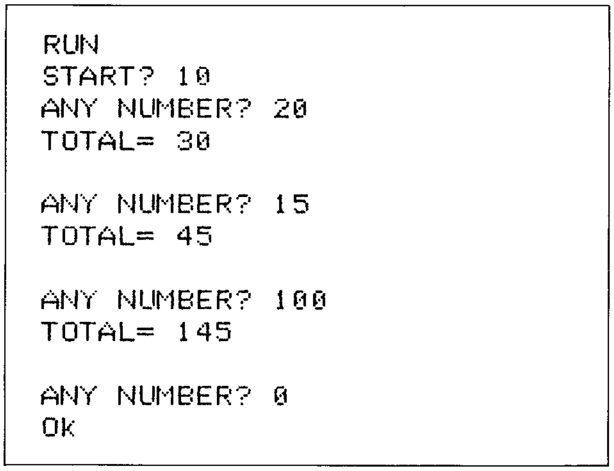
```
IF Y=0
```

which means, "If Y is equal to 0." If this condition is true, then the statement which follows THEN will be executed. In our program, this is the END statement, so the program would be ended.

Reminder

In the LET statement, the = sign was used to mean that the number on the right of the sign is the value of the variable name on the left of the sign. But when the = sign is used in the conditional expression in the IF—THEN statement, it means "is *equal to*."

Now let's RUN the program:



```
RUN
START? 10
ANY NUMBER? 20
TOTAL= 30

ANY NUMBER? 15
TOTAL= 45

ANY NUMBER? 100
TOTAL= 145

ANY NUMBER? 0
Ok
```

First 20 was added to 10, then 15 was added to that total, and then 100 was added to that. Then the next time a number was input, it was 0. The program is at line 30 at this point. Since 0 was input as the value for the variable Y, at line 35 the conditional expression Y = 0 becomes true, and the END statement following THEN is executed, and the program ends.

RENUMBERING LINES RENUM

When we first wrote the program, we numbered the lines in units of 10—10, 20, 30 ... 70. But later we added line 35, which broke up the units of 10. This is all right—if you want, you can use irregular numbers like 1, 3, 28, 29, 78, 105 ... and so forth to number the lines of your programs. But if you want to keep the lines numbered in units of 10, or if you have added so many lines between two of the original line numbers that there is no more space, you can use the RENUM statement to renumber the lines.

Input the

```
RENUM
```

command in the direct mode, and check to see if the numbers are renumbered in units of 10.

THE LOOP SPECIALIST

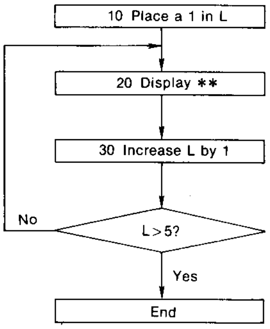
- ●Making a Loop—FOR—NEXT
- ●A Loop Within a Loop

SPECIFYING LOOP REPETITIONS FOR—NEXT

Previously we learned how to escape from a loop using the IF—THEN statement.

There is also a way to specify how many times a loop will be repeated which is sometimes more convenient to use. Let's learn this new method by writing a simple program.

```
10 FOR L=1 TO 5
20 PRINT "** ";
30 NEXT L
```



The following display appears on the screen when this program is run.

```
RUN
** ** ** ** **
Ok
```

The PRINT statement in line 20 of the program display ** ␣ one time. (␣ indicates a space.)

Then with the execution of the whole program, ** ␣ is displayed five times. This is the result of line 20 being repeated five times. The statements in line 10 and line 30 give the instructions for repeating line 20.

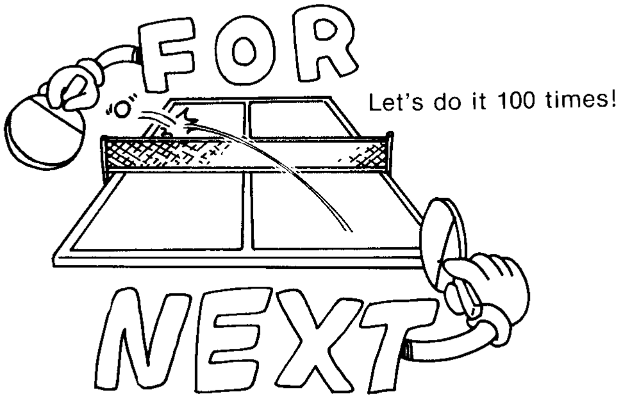
```
FOR variable = first value TO last value
     |
NEXT variable
```

The FOR—NEXT statement tells the computer to repeat the part of the program that is between FOR and NEXT the specified number of times.

```
FOR L=1 TO 5


NEXT L
```

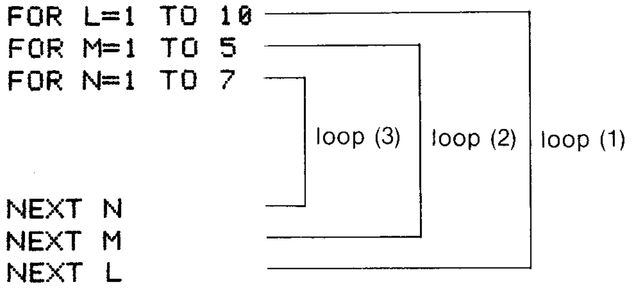
Line 10, FOR L = 1 TO 5, and line 30, NEXT L, form a pair which tell the computer to increase the value of L, beginning with 1, and to repeat the part of the program in between the two statements until the value of L reaches 5. When the program is executed, 1 is placed in L. Line 10 also declares that the final value of L will be 5.
Then line 20 is executed one time. Next line 30 increases L's value by 1. If the value of L has not reached 5, the line after the FOR statement (line 20) is executed again. This process is repeated until the value of L becomes 5.
The FOR statement and the NEXT statement act in this way to repeat the part of the program that is between them for whatever number of times you specify. Since they are always used as a pair, they are often referred to as the FOR—NEXT statement, and a loop made by a FOR—NEXT statement is called a **FOR—NEXT loop**. An important point to remember is that **the same variable must be used** in both the FOR statement and the NEXT statement (in this case it is L).



A Loop Within a Loop

There are many interesting things you can do with the FOR—NEXT statement. For example, you can put one or more FOR—NEXT loops inside the FOR—NEXT loop that is enclosed by the first FOR—NEXT statement.

The structure of a program like this is:



```
FOR L=1 TO 10
FOR M=1 TO 5
FOR N=1 TO 7



NEXT N
NEXT M
NEXT L
```

In this example, loop (2) is inside loop (1), and loop (3) is inside both of them. Loop (1) is repeated ten times, but each time it is repeated, loop (2), which is enclosed by loop (1), is repeated five times. And each time loop (2) is repeated, loop (3), which is inside it, is repeated seven times.

In writing a program like this, you must always use a different variable (like L, M, and N, above) inside each loop.

COMPUTERS AND LANGUAGE

A computer remembers your directions as a series of steps, and executes them in the order you give them. The method used to write this series of steps, or procedure, is called **computer language**. BASIC is one of the computer languages.

If you were able to write a program in regular English as:

1. Display ***
2. Wait one second
3. Display *** on the next line

it would be easier for you. But to have the computer understand commands written in this way would require much more complicated software for the computer than BASIC.

BASIC is a computer language that resembles human language quite closely, and it is fairly easy to understand. But the real strong-point of computers is handling numerical values. Therefore, instead of using normal English like "Wait one second," in BASIC you have the computer perform an operation that changes the value of a variable, such as

```
FOR T=1 TO 480:NEXT T
```

and achieve the same results as you would if the computer understood the regular English.

Variables and line numbers play important roles in making BASIC programs. Becoming familiar with how they are used is one of the keys to making progress in learning how to do BASIC programming.

READING DATA

- Preparing Data in the Program for Variable Values—READ—DATA

ANOTHER METHOD FOR ASSIGNING VALUES TO VARIABLES READ·DATA

As we have seen, using variables is one of the keys to successful BASIC programming. All kinds of values are given to variables, and then they are used for making calculations and decisions. Giving a value to a variable is referred to as **assigning** a value.
Let's review the methods used to assign values to variables.

First, we can assign values using the LET statement.
To assign a value of 100 to variable A, we write:
LET A = 100 (or just A = 100).

The next method we learned was the INPUT statement.
In contrast to the LET statement, which assigns a fixed value to a variable in the program, the INPUT statement allows you to input any value you want from the keyboard while the program is being executed.
If you enter 500 from the keyboard when

```
INPUT A
```

is executed and a question mark (?) is displayed, then 500 will be assigned as the value of A.

The READ·DATA statement is another BASIC statement that can be used to assign values to variables.

Let's try it with a simple program.

```
10 READ A
20 PRINT A
30 DATA 185
RUN
 185
Ok
```

When line 10

```
READ A
```

is executed, the value given in the DATA statement (185, line 30 in this program) is assigned to the variable A. (The value is displayed by the PRINT statement in line 20.) The same is true for character variables, also.

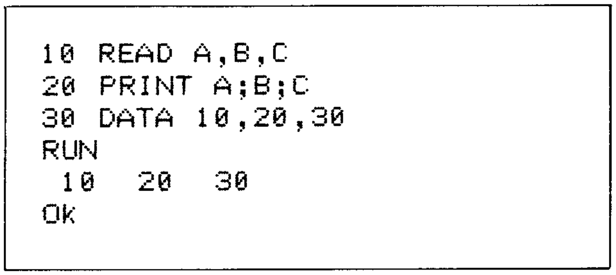
```
10 READ B$
20 PRINT B$
30 DATA SONY
RUN
SONY
Ok
```

SONY is a character string value, but it is not necessary to enclose it in “ ” in the DATA statement.
(However, if you want to include a comma, or want to leave a space before a word, the entire character string must be enclosed by “ ”.)

```
10 READ B$
20 PRINT B$
30 DATA "SONY, HITBIT"
RUN
SONY, HITBIT
Ok
```

Increasing the Amount of Data

The above programs assigned just one value to one variable. Let's increase the data (values) to three. Since our data are increased to three, we increase the number of variables to three also.



```
10 READ A,B,C
20 PRINT A;B;C
30 DATA 10,20,30
RUN
 10  20  30
Ok
```

A comma is used to separate each variable included in a READ statement. The data in the DATA statement that is assigned to each variable is listed in the same order, and also separated by commas.

The most important point to remember is that you must provide at least as much data in the DATA statements as the READ statements demand.

SAVING PROGRAMS ON TAPE

- ●Connecting a Tape Recorder
- ●Saving the Computer Program on Tape—CSAVE
- ●Checking that the Program is Saved—CLOAD?
- ●Loading a Program from the Tape—CLOAD

Anyone who has ever seen a large computer in operation probably noticed that there was another machine beside it that had tapes whirling back and forth inside it and looked like a large tape recorder. Such a machine is in fact a tape recorder—one of the very best. Such tape recorders are used to record the programs the large computers run.

Your own computer remembers a program as long as it is turned on. But when you turn it off, press the RESET button, or execute the NEW command, the program instantly disappears from the computer's memory.

If you had a device like the tape recorders used with large computers, you could record (**save**) your program on its tape before you turn off your computer. Then even though the computer's memory is erased when you turn off the switch, you would be able to read (**load**) the program into the computer's memory from the tape the next time you turn on the computer.

Actually, you don't need a large tape recorder like those used with large computers. The regular cassette tape recorder you probably already have will work very well as a recording device for your computer.

> This section explains how to save a program you have made by recording it on cassette tape. If you have a computer with a disk drive, you can skip this section and go on to the next section, "Saving Programs on Disks," which decribes how to record your programs on floppydisks.

If your cassette tape recorder is not already connected to your computer, turn off the computer and refer to the following explanation for directions on how to connect a tape recorder. When you turn off the computer, the program will be erased from the computer's memory. So after you connect your cassette recorder, enter the program again.

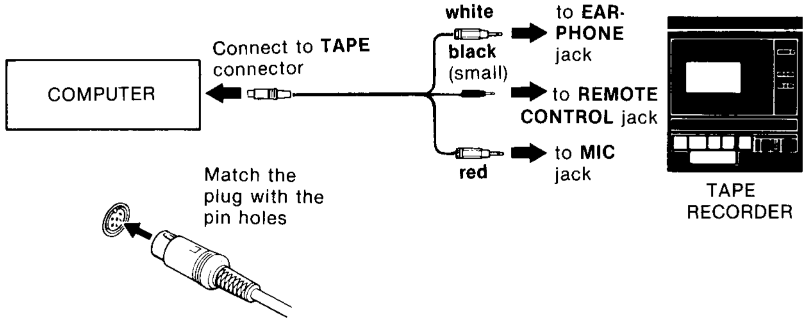
Connecting a Tape Recorder

Connect your cassette tape recorder to the computer TAPE connector using the cassette tape recorder connecting cord. There is also a MIC jack on your cassette recorder which you use when recording sound. And there is another jack for earphones, which is indicated by one of the following labels:

EAR, EARPHONE, MONITOR, MON, or 👂

There also might be a
REMOTE
jack for remote control.

Connect the computer connecting cord to these jacks as shown in the below diagram.



If your cassette recorder does not have a remote control jack, you can leave the black plug unconnected.

SAVING THE COMPUTER PROGRAM ON TAPE
CSAVE

The BASIC command for saving a computer program on tape is CSAVE.

CSAVE “file name”

CSAVE records a program with a specified file name on the cassette tape in intermediate language.

The file name is the name you give to a program to distinguish it from other programs.

There are three rules to remember in giving a program a file name:

- the name can be no longer than 6 characters.
- you can use numbers and signs, in addition to the letters of the alphabet.
- the computer distinguishes between capital letters and small letters.

Now let's give a file name of HEART to a program to be saved, and save it on the tape. Type the following command.

But **do not** press the ↵ key yet.

```
CSAVE "HEART"
```

Now, before pressing the ↵ key, place the cassette recorder in the record mode. If you have the remote control jack connected, the tape will not move. It only starts recording when you press the ↵ key. The program passes through the computer TAPE connector and is recorded on the tape.

If the remote control jack is not connected, the tape will start moving when you place the cassette recorder in the record mode. Check that the tape is moving, and then immediately press the ↵ key.

CHECKING THAT THE PROGRAM IS SAVED CORRECTLY CLOAD?

Always check to make sure that the program has been accurately saved after you record it on the tape. First rewind the tape to a point on the tape just before where you recorded the program. (On some tape recorders you must disconnect the remote jack before you can rewind the tape.) Now set the tape recorder volume control at about its mid-point, and then input the following command.

```
CLOAD? "HEART"
```

CLOAD? "file name"

CLOAD? checks the program on the tape and the program in the computer's memory to make sure that they are exactly the same.

After typing the CLOAD? command, press the ↵ key and then place the cassette recorder in the **PLAY mode**. (If the remote jack is connected and you press PLAY before pressing the ↵ key, the tape will not begin to move until you press the ↵ key.) When the tape comes to the place where the program is recorded.

```
Found:HEART
```

will be displayed. This message tells you that the computer has found the file named HEART on the tape. Then the computer compares the program on the tape with the program it has retained in its memory. The time required for this comparison depends on the length of the program. If the two programs are exactly the same

```
Ok
```

will be displayed on the screen.

If the program in the computer's memory and the one saved on the tape are not the same, the

```
Verify error
```

message will be displayed. The "Verify error" message means that the program has not been recorded correctly on the tape. If this message appears, you should save the program again, using the CSAVE command.

Also, sometimes the

```
Found:HEART
```

message is not displayed, even though the tape has passed the place where the program is recorded. This is usually due to the volume setting on the tape recorder being set either too low or too high. Change the level of the volume, and execute the CLOAD? command again. When the "Found HEART" message is displayed, remember the volume control setting on the cassette recorder, as that is the correct level for using the CLOAD? command.

LOADING A PROGRAM FROM THE TAPE CLOAD

The **CLOAD command** is used to have the computer read (**load**) a program from the tape into its memory.

CLOAD "file name"

CLOAD loads the program with the specified file name into the computer's memory.

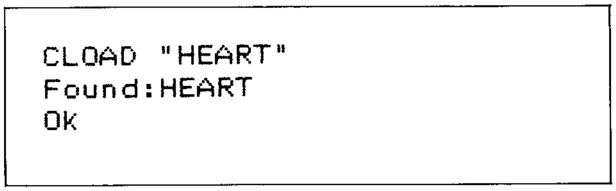
You enter the name of the file that you want to transfer from the tape to the computer in the "file name" space of the command. If this case it would be:

```
CLOAD "HEART"
```

Press the ↵ key and place the cassette recorder in the PLAY mode. (If the remote jack is connected and you press PLAY before pressing the ↵ key, the tape will not begin to move until you press the ↵ key.)
When the tape comes to the place where the program is recorded,

```
Found:HEART
```

will be displayed. Then after the entire program has been loaded into the computer's memory, the screen display will show:



```
CLOAD "HEART"
Found:HEART
Ok
```

If instead of Ok,

```
Device I/O error
```

is displayed, the program has not been properly loaded into the computer's memory. Change the volume level on the tape recorder and use the CLOAD command to load the program again.

Once the program has been loaded, input the LIST command to check that the program is in the computer's memory, and then execute the program with the RUN command.

SAVING PROGRAMS ON DISKS

- Formatting a Disk—CALL FORMAT
- Saving the Computer Program on a Disk—SAVE
- Checking that the Program is Saved—FILES
- Loading a Program from the Disk—LOAD
- Erasing a Program on the Disk—KILL

This section is for computers with disk drives. If your computer does not have a disk drive, refer to the previous section, "Saving Programs on Tape" for directions on how to save programs.
If your MSX2 computer does not have a disk drive, you can still save programs on disks by using a floppydisk drive unit such as the Sony HBD-50. In this case it is necessary to connect the floppydisk drive unit to your computer and start up MSX-Disk BASIC. Refer to the directions "Using an External Disk Drive," below.

Using an External Disk Drive

(1) Turn off the computer and the external disk drive.
(2) Plug the disk drive interface cartridge into the computer cartridge slot.
(3) Turn on the disk drive. Next, turn on the TV monitor and the computer. After a short pause, the following display may appear on the screen.

```
Enter date (Y-M-D):
```

(4) If the above message appears, press the ↵ key.

This completes the start-up procedure for MSX-Disk BASIC.

FORMATTING A DISK CALL FORMAT

When you turn the computer off, the program you worked so hard to write will be erased from the computer's memory. So before turning off the computer the program should be saved on a floppydisk in the computer's disk drive. Then even if you turn off the computer, you will be able to load the program from the floppydisk into the computer's memory again the next time you use the computer.

But before you can save a program on a new floppydisk, the disk must be **formatted**.

When you format a disk, the computer writes special data on the disk following a fixed set of rules, which serve as "guideposts" for the computer to know immediately where each file is located on the disk. Formatting a disk is like drawing lines on a sheet of plain paper so that you will know where to write and where to leave spaces between lines. You don't need to be concerned with the data used to format a disk—the computer takes care of that. But the rule to remember is **always format a new disk before you use it.**

Note: When you format a disk, all the data written on the disk will be erased.

Now, let's format a new disk. The procedure is as follows:
(1) Execute the CALL FORMAT command.

When you press the [↵] key,

```
Drive name ? (A,B)
```

is displayed.

This message asks if you want to format a disk in drive A or a disk in drive B. When you are using only one disk drive, it is drive A, so you would press the [A] key.

(2) If your disk drive accepts double-sided disks,

```
1 - Single sided, 9 sectors
2 - Double sided, 9 sectors
```

will be displayed next. If your disk is the single-sided type, press the [1] key. If it is a double-sided type, press the [2] key.

When you press the key,

```
Strike a key when ready
```

is displayed. (With a single-sided disk drive such as the SONY HBD-50, this message is displayed when you press the A key.)

(3) Insert the new disk into the disk drive and press any key on the keyboard. The computer then formats the disk. When formatting is completed the screen displays:

```
Format complete
Ok
```

SAVING THE COMPUTER PROGRAM ON A DISK SAVE

The SAVE command is used to save on a disk, in intermediate language, the program in the computer's memory.

```
SAVE “A: file name. type name”
```

A: specifies the A disk drive, but if you are using only one disk drive it can be omitted.

The **file name** is the name you give to a program to distinguish it from other programs.

There are four rules to remember in giving a program a file name:

- the name can be no longer than 8 characters
- you can use numbers and signs, in addition to the letters of the alphabet
- the computer does not distinguish between capital letters and small letters
- The following characters cannot be used as part of a file name: , . ; : “ * ? and space

The **type name** shows the type of the file. It consists of a period (.) followed by a name of three letters or less. Since the type of file we are saving is a BASIC file, let's use “.BAS” as the type name. This indicates that the file is a program written in BASIC. The type name can be omitted, but as you begin saving different types of files on disks you will see that it is very convenient always to include it as part of the SAVE command. So you should develop the habit right from the start of including the .BAS type name in the names of all of your BASIC programs.

Let's give a program the file name HEART, and the type name .BAS and save it on the disk.

Input the following command:

```
SAVE "HEART.BAS"
```

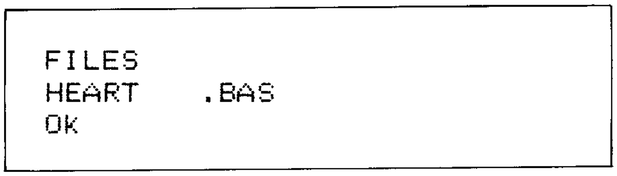
When you press the ↵ key, you will hear the sound of the disk drive operating. When the program has been saved, Ok will be displayed.

CHECKING THAT THE PROGRAM IS SAVED FILES

The FILES command is used to check if the program has been saved on the disk.

FILES

The FILES command displays the file names and type names of all files saved on a disk.



```
FILES
HEART    .BAS
Ok
```

When you execute the FILES command, HEART.BAS, the name of the file you just saved, should be displayed. If other files are saved on the same disk, their file names will also be displayed.

LOADING A PROGRAM FROM THE DISK LOAD

Use the NEW command to clear the computer's memory before loading a program from the disk.

LOAD is the command for loading a program from the disk.

LOAD "A: file name .type name"

A: can be omitted. The file name and the type name are the same names you used when you saved the program. Therefore, you would input

```
        LOAD "HEART.BAS"
```

When you press the ↵ key, the disk drive will operate and begin to load the program. When it has been loaded, Ok will be displayed. Check that the program is in the computer's memory using the LIST command, and then execute the program with the RUN command.

Anytime you want to use a particular program that has been saved on a disk, you can load it into the computer's memory using the LOAD command.

ERASING A PROGRAM ON THE DISK KILL

The disk will become full as you save more programs on it.

KILL is the command for erasing programs on the disk you no longer need.

KILL "A: file name file .type name"

A: can be omitted. To erase the HEART .BAS program you saved on the disk, input the command:

```
KILL "HEART.BAS"
```

Chapter 3
Array Variables

A PROGRAM USING ARRAY VARIABLES

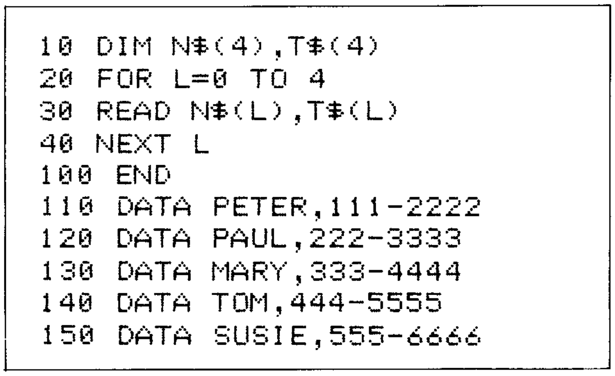
- The Array Variable Declaration—DIM

HOW TO USE ARRAY VARIABLES **DIM**

Let's use some array variables in an actual program. We'll use two string type array variables, N$(N) and T$(N), and have each of them contain five different pieces of data. The value of N will therefore be from 0 to 4.
We'll use the N$(N) array variable for people's names, and the T$(N) array variable for telephone numbers. They will contain the following data.

Array variable	Data	Array variable	Data
N$(0)	PETER	T$(0)	111-2222
N$(1)	PAUL	T$(1)	222-3333
N$(2)	MARY	T$(2)	333-4444
N$(3)	TOM	T$(3)	444-5555
N$(4)	SUSIE	T$(4)	555-6666

Assigning Data to Array Variables
Just as you did with regular variables, you use the LET statement, INPUT statement, or READ—DATA statement to assign data to array variables. Here we will use a READ—DATA statement.



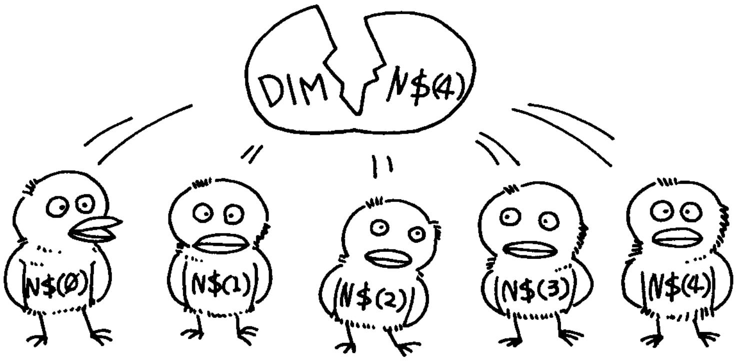
```
10 DIM N$(4),T$(4)
20 FOR L=0 TO 4
30 READ N$(L),T$(L)
40 NEXT L
100 END
110 DATA PETER,111-2222
120 DATA PAUL,222-3333
130 DATA MARY,333-4444
140 DATA TOM,444-5555
150 DATA SUSIE,555-6666
```

We have used a completely new statement in line 10.

```
10 DIM N$(4),T$(4)
```

The DIM statement is used to declare how many array variables will be used in a program, and how many variables they will contain.

DIM N$(4), T$(4) declares that N$, which will have 5 variables from N$(0) to N$(4), and T$, with 5 variables from T$(0) to T$(4), will be used in the program.

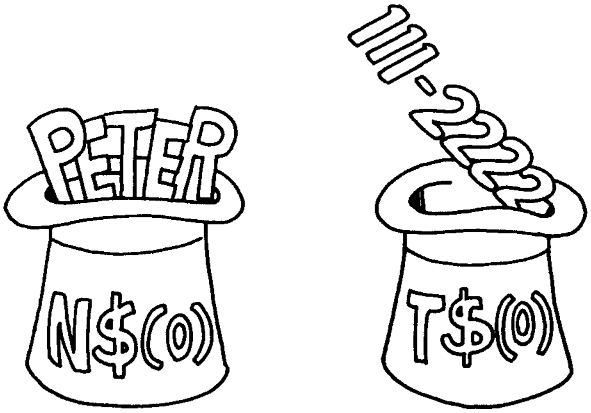


DIM array variable name (maximum value)

The DIM statement specifies the maximum value of the number of variables within the ( ) of an array variable. If the maximum value is N, then the array variable will have variables form 0 to N, which would be equal to the value of N plus 1.

Always remember to declare an array variable with the DIM statement before you use the array variable in a program.

After the array variables have been declared in line 10, the FOR—NEXT loop in lines 20 to 40 assigns data to the array variables. The first time the loop is executed, the value of L is 0, so the READ statement in line 30 assigns PETER to N$(0) and 111-2222 to T$(0).



Then data is assigned consecutively to N$(1), T$(1) and so forth. The program ends when the final loop assigns SUSIE to N$(4) and 555-6666 to T$(4).
You can use the PRINT statement in the direct mode to check that these data have in fact been assigned to the variables.

When you input

```
PRINT N$(0),T$(0)
```

```
PRINT N$(0),T$(0)
PETER         111-2222
Ok
```

will be displayed. If you input PRINT N$(1), PAUL will be displayed, or if you input PRINT T$(2), 333-4444 will be shown.

This ends the introduction to MSX2-BASIC. If you have learned the fundamentals of making BASIC programs, the objective of this introduction has been accomplished.

Now you can use the BASIC commands you have learned to make your own programs. The best way to continue making progress in writing programs is to use the ability you now have to make some revisions to the programs you made using this book. There are many ways these programs can be made easier to use if you just think about them a little bit. And when you start to make your own programs, don't be afraid of making a mistake. Even if there is a mistake in your program, it won't damage the computer. The computer will simply read the mistake and then will display an error message to tell you what and where you did something wrong. If an error message is displayed, use the LIST command to display the program and see what you did wrong. Trying different things out, and correcting any mistake you make, is the way to make progress in becoming a good BASIC programmer.

Chapter 4
The Memory Switch Function

THE SET STATEMENT

- ●The Memory Switch Function
- ●Adding a Title—SET TITLE
- ●Changing the Prompt Statement—SET PROMPT
- ●Specifying a Password—SET PASSWORD
- ●Changing the Location of the Display on the Screen—SET ADJUST
- ●Setting the "BEEP"—SET BEEP
- ●Specifying the initial Status of the Screen—SET SCREEN

THE MEMORY SWITCH FUNCTION

MSX2-BASIC has a special function called the memory switch function which can be used to change the initial settings of the computer when BASIC is started up, and have the changed settings retained by a battery-backup RAM in the timer IC.

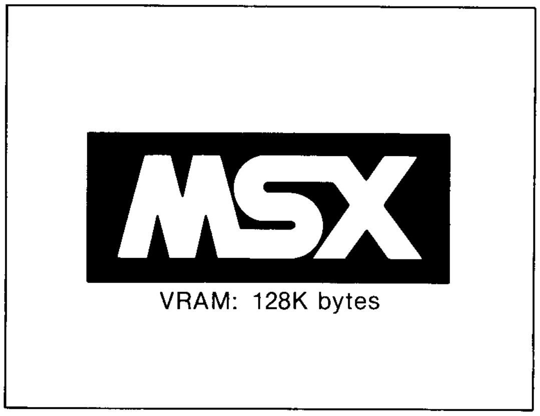
RAM (random access memory) is the main memory of the computer. (We referred to it as the "computer's memory" in the BEGINNING SECTION.) When the computer is turned off, the content of the RAM is erased. But a battery-backup RAM will retain its content even if the power is turned off. Consequently, if we have this special RAM retain the initial BASIC settings, the next time the computer is turned on these settings will be the settings it uses.

The following initial settings can be changed and retained by the battery-backup RAM.

- ●Title and Prompt Statements
- ●Password
- ●The position of the display on the screen
- ●The type and loudness of the "BEEP" sound
- ●SCREEN statement settings

ADDING A TITLE SET TITLE

When BASIC is started up, the following display is shown on the screen before Ok is displayed.



VRAM: 128K bytes

You can add a title of your choice below the "VRAM: 128K bytes" (or VRAM 64K bytes) line in this display. The title is defined by the SET TITLE statement.

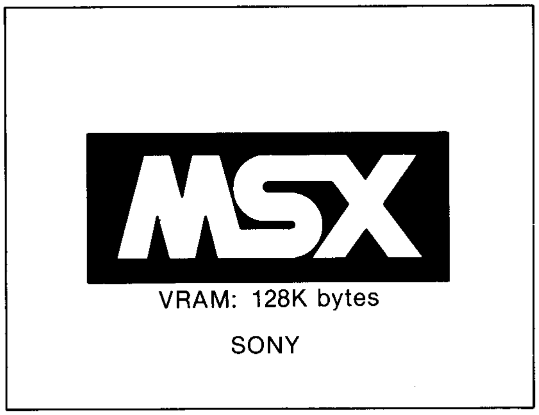
SET TITLE ["title statement"], [color]

The title can contain up to six characters. Also, four different color combinations can be specified for the MSX logo display. If you want, you can omit the title and specify only the color combination of your choice.

To display the title "SONY," execute the following SET TITLE statement in the direct mode.

```
SET TITLE "SONY"
```

After executing this statement press the RESET button or turn off the computer briefly and then turn it back on. The title will now be displayed as shown in the below illustration.



VRAM: 128K bytes

SONY

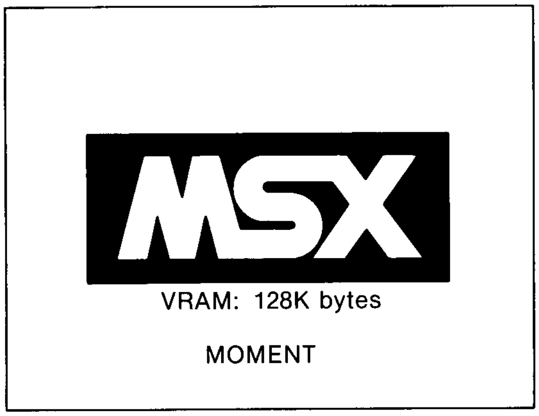
This title will be retained in the battery-backup RAM even if the computer is turned off, and will be displayed each time BASIC is started up.

Freezing the Screen Display with the Title Statement

If you specify a title of exactly six characters, the display will be frozen on the screen when the title appears. For example, if you specify

```
        SET TITLE "MOMENT"
```

the next time you start BASIC the screen will show the following display



VRAM: 128K bytes

MOMENT

Since "MOMENT" is exactly six characters, this display will remain on the screen until a key is pressed on the keyboard. Then Ok will be displayed and BASIC will be ready for use. If you use the SET TITLE statement to specify six spaces with the space bar, instead of characters, no title will be displayed, but the MSX display will remain on the screen until a key is pressed.

To Cancel a Title
A title can be canceled by executing

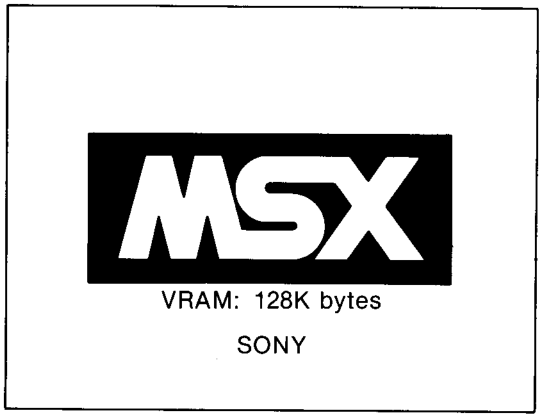
```
SET TITLE ""
```

Do not type any character or space between the quotation marks. Also, if the SET PROMPT statement or SET PASSWORD statement is executed, the title specification will be canceled.

Changing the Color of the Title Display
The color specification in the SET TITLE statement can be used to specify any of four different screen color combinations when the MSX logo is displayed. The color is specified with the numbers from 1 to 4. When

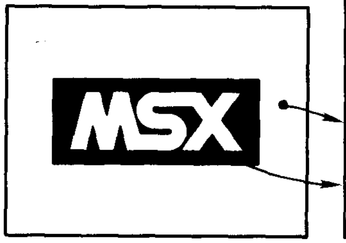
```
SET TITLE "SONY",3
```

is executed, the screen will show the following colors.



```
VRAM: 128K bytes
SONY
```

The following table shows the color combinations that can be specified.



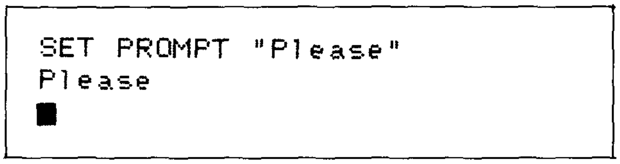
Color			
1	2	3	4
dark blue	dark green	red	dark yellow
black	dark blue	magenta	red

CHANGING THE PROMPT STATEMENT
SET PROMPT

The Ok message is displayed when BASIC is in the ready condition to accept commands. Ok is called the prompt statement. **You can change this prompt statement to any word of up to six characters in length.** For example, the following SET PROMPT statement would change the prompt from Ok to Please.

```
SET PROMPT "Please"
```

Once the PROMPT has been changed with the SET PROMPT statement, the new PROMPT will be displayed each time BASIC is started up, and will not change until another PROMPT statement, a SET TITLE statement, or a SET PASSWORD statement is executed. When a SET TITLE statement or a SET PASSWORD statement is executed, the PROMPT will revert to the original Ok prompt.



```
SET PROMPT "Please"
Please
■
```

SPECIFYING A PASSWORD SET PASSWORD

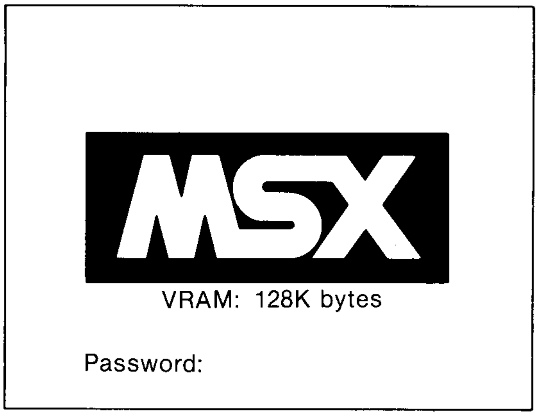
A password is a word known only to you which must be input before BASIC can be started up. The SET PASSWORD statement is used to specify a password.

SET PASSWORD “password”

The password can be a character string of any length up to 255 characters. For example, if you want to use “I LIKE BASIC” (12 characters, including spaces) as the password, you would input:

```
SET PASSWORD "I LIKE BASIC"
```

Once this statement has been executed, the next time BASIC is started up the screen will show the following display until the password is entered.



```
VRAM: 128K bytes

Password:
```

BASIC will not start until you enter the password I LIKE BASIC and press the ↵ key. If the password is not entered, or the wrong password is entered, BASIC will not start. In this way, only you (or whoever knows the password) can use the computer.

Once a password has been set, cartridge software such as games also cannot be started until the password is entered.

To Cancel the Password

The password is canceled when a SET PROMPT statement or a SET TITLE statement is executed. For example, when

```
SET PROMPT "Ok"
```

is executed, the PROMPT will become Ok, and the password will be canceled. In summary, the last SET statement—SET PROMPT, SET TITLE, or SET PASSWORD—to be executed becomes the valid statement, and any previously executed SET statement is canceled.

If You Forget the Password

If you forget the password you have set, hold down both the GRAPH key and the STOP key and press RESET. Continue to press the three keys, and the MSX logo display will appear on the screen. After you have checked that

```
Password:
```

is not displayed on the screen, release the three keys. BASIC will then start.

Another way to bypass the password is to press and hold down both the GRAPH key and the STOP key when you turn the computer power switch ON.

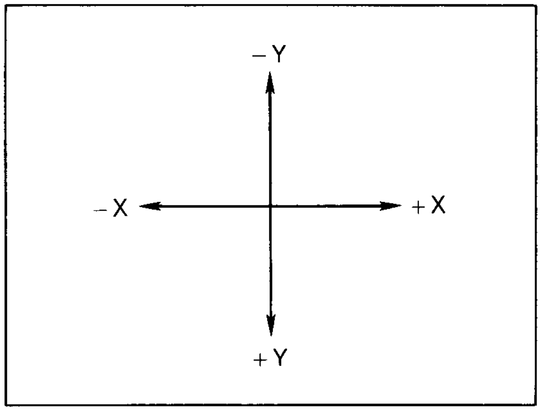
CHANGING THE LOCATION OF THE DISPLAY ON THE SCREEN SET ADJUST

Depending on the type of monitor TV you use, the display area sometimes will not be centered precisely on the screen. The SET ADJUST statement is provided to allow you to adjust the display area so that it is properly centered.

SET ADJUST (X,Y)

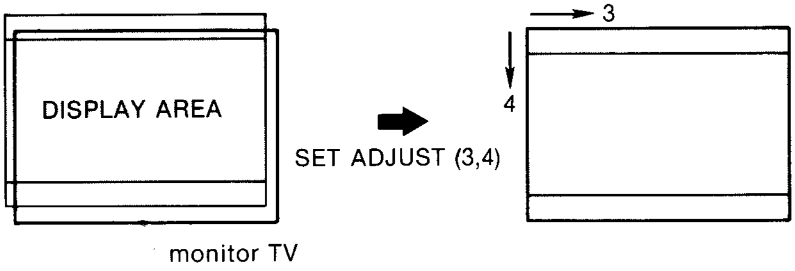
Numbers from −7 to +8 can be specified for both X and Y. The location of the display area will be changed one coordinate point with each change in the value of the number specified.

The initial default settings of X and Y are 0. Specifying a + value for X will move the display area to the right, and a − value will move it to the left. A + value for Y moves the display area vertically down the screen, and a − value moves it up.



For example, if the display area is off-center to the top and left, as shown in the below illustration, you can center it by moving it 3 coordinate points to the right, and 4 coordinate points down by executing

```
SET ADJUST (3,4)
```



Let's use the SET ADJUST statement in a short program to obtain an interesting effect.

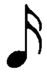
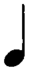
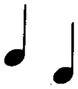
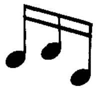
```
10 SCREEN 2
20 CIRCLE (125,95),60
30 FOR Y=-7 TO 8
40 FOR X=-7 TO 8
50 SET ADJUST (X,Y)
60 NEXT X
70 NEXT Y
80 GOTO 30
```

SETTING THE "BEEP" SET BEEP

MSX BASIC produces a "BEEP" sound when there is an error in a program. The SET BEEP statement is used to set the sound pattern and volume.

SET BEEP [pattern] [,volume]

The sound patterns and volumes are specified with the numbers from 1 to 4. For the volume, 1 is the lowest volume and 4 is the highest volume.

No.	1	2	3	4
Pattern				
Volume	lowest volume			highest volume

To set the BEEP pattern at the 3 setting, and the volume at the 4 setting, you would execute

```
SET BEEP 3,4
```

SPECIFYING THE INITIAL STATUS OF THE SCREEN **SET SCREEN**

The SCREEN statement will be explained in detail in Chapter 5. Here we will just list the settings that can be made with the SCREEN statement.

- character mode (SCREEN 0 or 1)
- key click switch ON/OFF
- printer type
- cassette baud rate
- interlace mode

Also, the number of characters in one line can be specified with the WIDTH statement.
The KEY ON/OFF statement specifies whether the contents of the function keys are listed at the botton of the screen or not.
The COLOR statement is used to specify the foreground color, the background color, and the border color.

When the SET SCREEN statement is executed in the direct mode, the current settings specified by the SCREEN statement, the WIDTH statement, the COLOR statement, and the KEY ON/OFF statement become the initial settings for when BASIC is started.

SET SCREEN

For example, when

```
SET SCREEN
```

is executed after the SCREEN 1 mode is set and the foreground color has been set as black, the background color as gray, and the border color as light blue as follows:

```
SCREEN 1
COLOR 1,14,5
```

these colors will be the screen colors the next time BASIC is started.

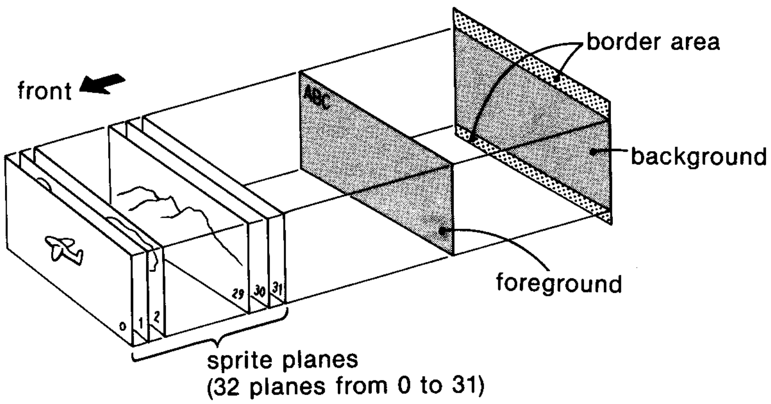
Chapter 5
Screen Configuration and Graphics

SCREEN MODE

- Screen Configuration
- Setting the Mode—SCREEN
- Specifying the Number of Characters Per Line—WIDTH
- Graphic Mode Coordinates and STEP Specification

SCREEN CONFIGURATION

The following diagram shows the MSX2-BASIC screen configuration.



SETTING THE MODE SCREEN

Characters (letters, numbers, etc.) are displayed in the **character modes**, and figures and drawings are displayed in dot units in the **graphic modes**. There are a total of nine text and graphic screen modes in MSX2-BASIC. The SCREEN statement is used to specify a screen mode.

SCREEN [mode number], [sprite size], [key click switch], [baud rate], [printer type], [interlace]

The SCREEN statement specifies the screen mode, sprite size, key click on/off, cassette tape baud rate, printer type, and interlace mode.

Nine screen modes can be selected with the SCREEN statement. (The parts of the SCREEN statement other than the screen mode are explained beginning on page 145.)

SCREEN Modes

Mode No.	Mode	Screen Display	Color	Page	Sprite
0	Text	Maximum 80 characters horizontal, 24 lines vertical	Color palette function 16 colors/512 colors	—	Not usable
1		Maximum 32 characters horizontal, 24 lines vertical	Color palette function 16 colors/512 colors	—	Used
2	Graphic 64K or 128K VRAM	256 x 192 dots	Color palette function 16 colors/512 colors (2 colors/8 dots)	—	Used
3		64 x 48 dots multicolor	Color palette function 16 colors/512 colors	—	Used
4		256 x 192 dots	Color palette function 16 colors/512 colors (2 colors/8 dots)	—	Used (enhanced sprite)
5		256 x 212 dots	Color palette function 16 colors/512 colors	2 pages (VRAM 64K) 4 pages (VRAM 128K)	Used (enhanced sprite)
6		512 x 212 dots	Color palette function 4 colors/512 colors	2 pages (VRAM 64K) 4 pages (VRAM 128K)	Used (enhanced sprite)
7	Graphic 128K VRAM only	512 x 212 dots	Color palette function 16 colors/512 colors	2 pages	Used (enhanced sprite)
8		256 x 212 dots	256 colors	2 pages	Used (enhanced sprite)

In all modes, characters and figures are displayed on the **foreground screen**. Behind the foreground is the **background screen**. You can change the color of the background screen, but you cannot display characters or figures on it. There is a **border area** at the top and bottom of the display screen. Like the background screen, you can only change the color of the border area, and cannot display anything on it.

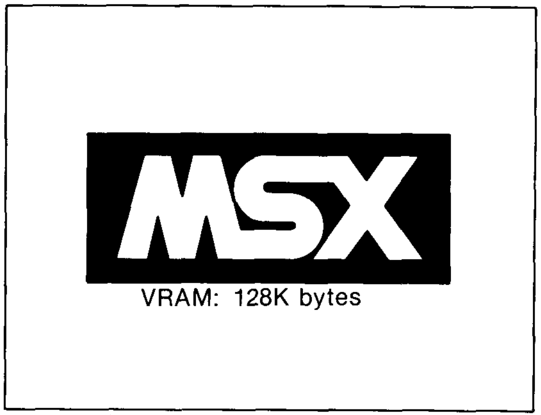
There are 32 **sprite planes** in front of the foreground screen which can be used to display and animate sprite patterns in all modes except SCREEN 0. Sprite patterns and their use are explained in the following chapter.

Note on the VRAM Size

As is indicated in the above chart, SCREEN 7 and SCREEN 8 modes are used only by computer with 128K bytes of VRAM. Also, the number of pages which can be used in SCREEN 5 and SCREEN 6 modes is different depending on the size of the VRAM.

VRAM stands for video RAM. It is the memory which remembers the content of what is to be displayed on the screen.

The size of the VRAM in your computer is shown on the MSX logo display which is displayed when BASIC is started.



VRAM: 128K bytes

the VRAM size

The initial display when the computer is turned on

CHARACTER MODE

The character mode for displaying text on the screen is specified by SCREEN 0 and SCREEN 1. In SCREEN 0, characters are displayed in 6 dot (width) × 8 dot (height) size. In SCREEN 1, characters are displayed in 8 dot (width) × 8 dot (height) size. The width of some of the MSX computer graphic characters is 8 dots, so you should use SCREEN 1 when you want to display graphic characters.

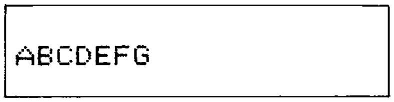
SPECIFYING THE NUMBER OF CHARACTERS PER LINE WIDTH

SCREEN 0 displays up to 80 characters per line. SCREEN 1 displays up to 32 characters per line. **The WIDTH statement specifies the number of characters to be displayed on each line in the text mode.**

WIDTH number of displayed characters

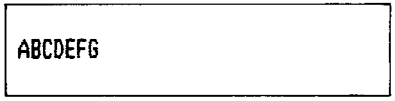
In SCREEN 0, the width of the characters displayed is different when 1 to 40 characters and when 41 to 80 characters are displayed on one line.

When SCREEN 0: WIDTH 40 is executed



```
ABCDEFG
```

When SCREEN 0: WIDTH 80 is executed



```
ABCDEFG
```

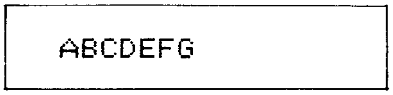
When from 1 to 40 characters are specified for one line, it is called the **40 character mode**. When 41 to 80 characters are specified, it is called the **80 character mode**.

The number of characters per line in SCREEN 1 is from 1 to 32, and the size of the characters are all the same.

```
                          ┌80 character mode ⎫ specified by
               ┌SCREEN 0─┤                   ⎬ the WIDTH
Character mode─┤          └40 character mode ⎭ statement
               │
               └SCREEN 1──32 character mode
```

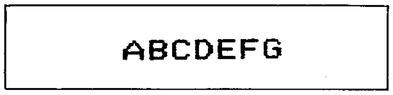
When WIDTH 80 is executed in the SCREEN 0, 80 character mode, when WIDTH 40 is executed in the SCREEN 0, 40 character mode, and WIDTH 32 is executed in the SCREEN 1 mode, the number of characters first fill the screen from left to right, and then as the number of characters displayed on each line is decreased in each mode, the display area of the characters is centered at the center of the screen.

SCREEN 0: WIDTH 30



ABCDEFG

SCREEN 1: WIDTH 10



ABCDEFG

The following program demonstrates the kinds of changes in the number of characters per line which can be made.

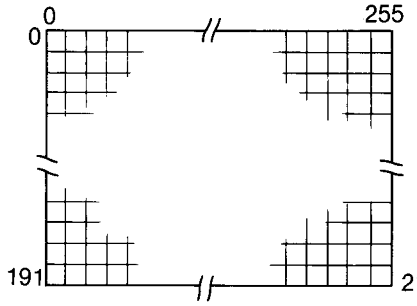
```
10 A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
20 B$="abcdefghijklmnopqrstuvwxyz"
30 SCREEN 0:CLS
40 FOR W=80 TO 10 STEP -10
50 WIDTH W
60 GOSUB 150
70 NEXT W
80 SCREEN 1
90 FOR W=32 TO 12 STEP -10
100 WIDTH W
110 GOSUB 150
120 NEXT W
130 SCREEN 0:WIDTH 40
140 END
150 PRINT "WIDTH";W
160 PRINT:PRINT:PRINT
170 PRINT A$;B$
180 FOR T=0 TO 1000:NEXT T
190 RETURN
```

THE GRAPHIC MODE AND COORDINATES

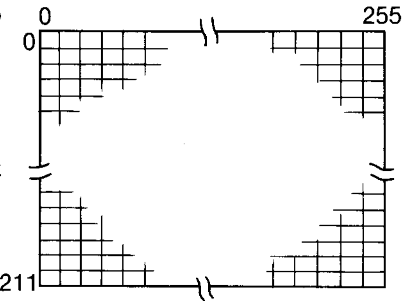
SCREEN 2 through SCREEN 8 specify graphic modes. The following statements are used to draw figures in all graphic modes.

PSET, PRESET ... draws dots on the screen
LINE ... draws lines and rectangles
CIRCLE ... draws circles, ovals, arcs, and fan shapes
PAINT ... colors a figure
DRAW ... draws figures specified by graphics sub-commands

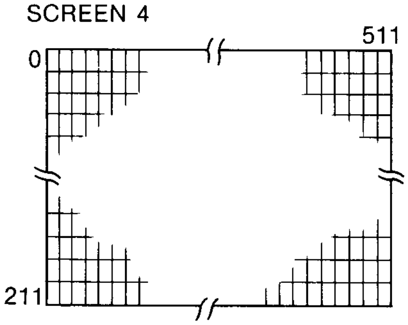
The screen is divided into coordinates which are used to specify locations on the screen when the above statements are used. The coordinates are different in different modes.



SCREEN 2, SCREEN 3, SCREEN 4

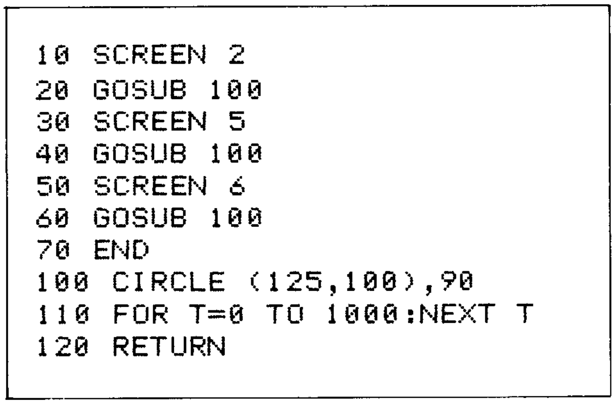


SCREEN 5, SCREEN 8



SCREEN 6, SCREEN 7

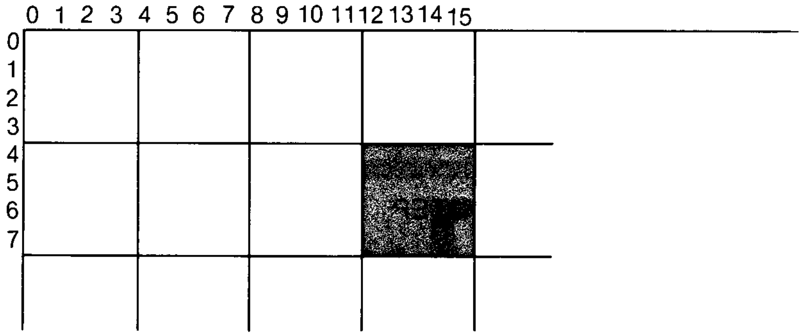
In the following program the same coordinates are used in SCREEN 2, SCREEN 5, and SCREEN 6 but the circle is positioned at different locations on the screen.



```
10 SCREEN 2
20 GOSUB 100
30 SCREEN 5
40 GOSUB 100
50 SCREEN 6
60 GOSUB 100
70 END
100 CIRCLE (125,100),90
110 FOR T=0 TO 1000:NEXT T
120 RETURN
```

MULTICOLOR MODE (SCREEN 3)

The same 256 × 192 dot coordinates are used in the SCREEN 3 mode as are used in the SCREEN 2 and SCREEN 4 modes, but the unit for drawing a figure is a 4 × 4 dot block.



```
PSET (12,4),1
PSET (14,5),1
PSET (15,7),1
```

For example, the above statements all specify locations within the same 4 × 4 dot block, so any of these PSET statements will paint the entire block black, as shown above.

The LINE statement

```
LINE (17,5)-(130,110)
```

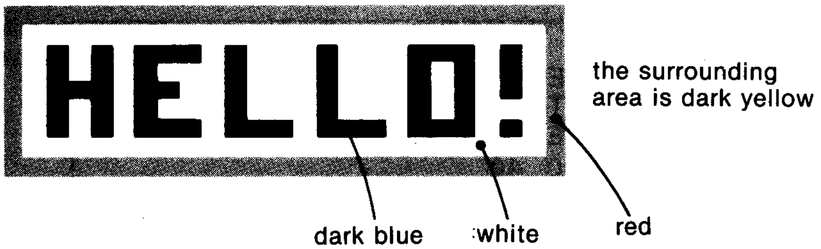
will draw a rough line between the blocks which includes the coordinates (17,5) and (130,110).

Let's run a program which uses the multicolor mode.

```
10 SCREEN 3
20 COLOR ,10,10:CLS
30 LINE (80,76)-(176,108),8,BF
40 FOR Y=80 TO 104 STEP 4
50 FOR X=84 TO 172 STEP 4
60 READ P
70 IF P=0 THEN SET BEEP 1 ELSE 100
80 PSET (X,Y),15:BEEP
90 GOTO 120
100 SET BEEP 2
110 PSET (X,Y),4:BEEP
120 NEXT X
130 NEXT Y
140 GOTO 140
150 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0
160 DATA 0,1,0,1,0,1,1,1,0,1,0,0,0,1,0,0
,0,1,1,1,0,1,0
170 DATA 0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,0
,0,1,0,1,0,1,0
180 DATA 0,1,1,1,0,1,1,0,0,1,0,0,0,1,0,0
,0,1,0,1,0,1,0
190 DATA 0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,0
,0,1,0,1,0,0,0
200 DATA 0,1,0,1,0,1,1,1,0,1,1,1,0,1,1,1
,0,1,1,1,0,1,0
210 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0
```

The SCREEN 3 statement in line 10 specifies the multicolor mode. In the loop, the READ statement in line 60 assigns data to variable P. If the value of P is 0, then the dot (block) drawn by the PSET statement will be color 15 (white). If the value is other than 0, the color will be 4 (dark blue). The value of P also determines which SET BEEP statement will be executed, and therefore which sound will be specified. This produces a different sound each time the PSET statement is executed.

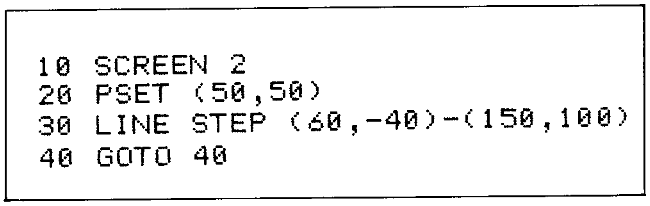
When the program is executed, the following display should be drawn:



SPECIFYING STEP

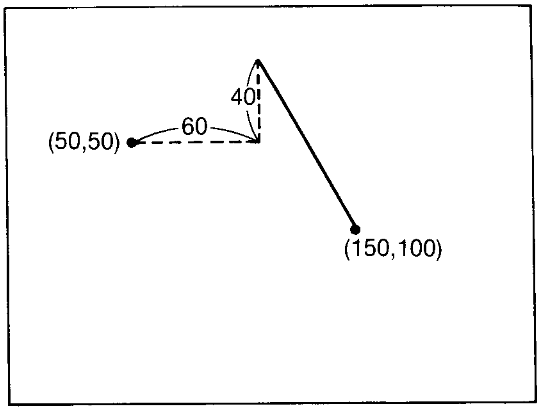
The STEP (X,Y) specification can be used with the PSET, PRESET, LINE, CIRCLE, PAINT, and PUT SPRITE (explained in the following chapter) statements to specify the (X,Y) coordinates.

When these graphic statements are executed, the last specified point is remembered. If STEP (X,Y) is then executed, the location of (X,Y) is determined on a new coordinate system with the point specified last as the origin (0,0). If the STEP specification is omitted, locations are specified in the regular coordinate system in which the top left corner of the screen is the origin.



```
10 SCREEN 2
20 PSET (50,50)
30 LINE STEP (60,-40)-(150,100)
40 GOTO 40
```

In this program, the coordinate (50,50) specified by the PSET statement in line 20 is remembered. Then in line 30, STEP (60, – 40) is used to specify the starting point for the LINE statement. (50,50) becomes the new origin, and the location 60 in the X direction and – 40 in the Y direction from the new origin is the starting point for the line.



The following formats are used when STEP is included in graphic statements.

```
PSET STEP(X,Y), color
PRESET STEP(X,Y), color
                                     B
LINE STEP(X,Y)—STEP(X,Y), color,
                                     BF
CIRCLE STEP(X,Y), radius, color, start angle, end angle,
aspect ratio
```

Displaying Characters in the Graphic Modes

Characters can also be displayed on graphic mode screens. To do this, the graphic screen is used as a file device. The file is opened and the characters to be displayed are output to the file. See page 242 for a full explanation.

SPECIFYING COLORS

- ●The Palette Function
- ●Palette Specification—COLOR
- ●SCREEN 8 Mode Color
- ●Color Spill (SCREEN 2 and SCREEN 4)
- ●Returning the Color Specifications to the Initial Settings —COLOR

THE COLOR CODE AND THE PALETTE FUNCTION

In the SCREEN 2 mode, 16 colors can be used, and each color has a color code. We will list the color codes below:

Color Code Table

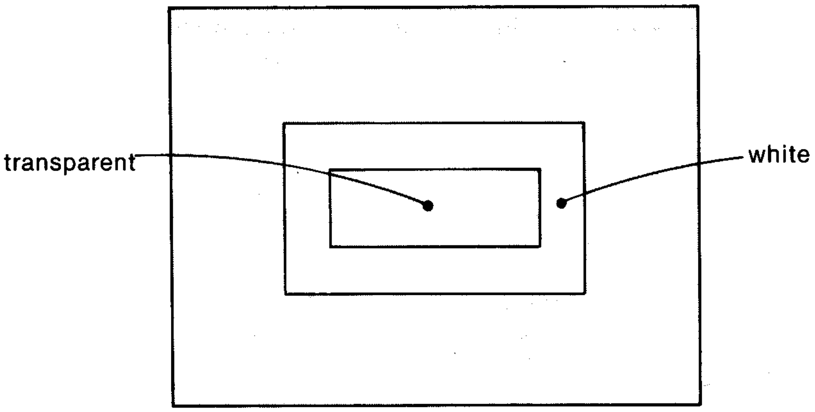
Code	Color	Code	Color	Code	Color	Code	Color
0	transparent	4	dark blue	8	medium red	12	dark green
1	black	5	light blue	9	light red	13	magenta
2	medium green	6	dark red	10	dark yellow	14	gray
3	light green	7	sky blue	11	light yellow	15	white

The Transparent Color

When color code 0 is specified, the color is transparent. This means that when the transparent color is used to draw a figure in the foreground, the background color will show through the figure. You can check this with the following program.

```
10 SCREEN 2
20 FOR B=2 TO 14
30 COLOR ,B,B:CLS
40 LINE (50,50)-(180,140),15,BF
50 LINE (70,70)-(160,120),0,BF
60 FOR T=0 TO 1000:NEXT T
70 NEXT B
80 COLOR ,4,4:CLS
90 END
```

The FOR—NEXT loop successively changes the background color and the border area color from 2 (medium green) to 14 (gray). A white square is drawn (line 40), and inside it a smaller transparent square is drawn (line 50).
Then each time the background color changes, the transparent square becomes the same color, since the background color shows through the transparent color.



The Color Palette
As was shown on page 90, the modes which use the 16 colors from color code 0 to color code 15 are: SCREEN **0**, SCREEN **1**, SCREEN **2**, SCREEN **3**, SCREEN **4**, SCREEN **5**, and SCREEN **7**.
The 16 colors shown in the color code table above are the 16 colors that can be used when BASIC is started up. But they are by no means all of the colors available with the MSX2. The color codes from 0 to 15 can be used to create 512 different colors of your choice.
It is not possible to give names to all of these 512 colors, so the colors are specified by designating the amount of red, green, and blue that is combined to produce any given color—for example, red 3, green 2, and blue 7—just as though you were mixing the red, green and blue on an artist's color palette. This is called the **palette function**.

Note
Colors can be specified for color code 0 also, but this is a special case. Here we will explain how to use color codes 1 through 15 to create different colors.

HOW TO USE THE PALETTE FUNCTION

The three colors red, green, and blue have brightness levels from 0 through 7. These levels are called the **brightness**. Since there are 8 different levels for each color, these levels can be combined to produce 512 colors—8×8×8=512.
The color is black when the red, green, and blue brightness is set at 0 for each of the colors, and it is white when the brightness of all three colors is set at 7. Setting the same brightness for the three colors (for example, brightness 4 for red, green and blue) produces gray.

red	green	blue	color
0	0	0	black
1	1	1	↑
2	2	2	dark gray
3	3	3	↑
4	4	4	↓
5	5	5	light gray
6	6	6	↓
7	7	7	white

When the brightness of one of the colors is made greater than that of the other two colors, (or the closer the brightness setting of the other two colors is to 0) the brighter color will predominate. For instance, 7, 0, 0 would produce a pure red color, as shown in the following brightness setting examples.

red	green	blue	color
4	3	3	gray, with a slightly red tinge
5	2	2	a color close to red
5	0	0	red (slightly dark)
7	0	0	red (pure red, the brightest color)
2	0	0	red that is almost black

PALETTE SPECIFICATION COLOR

The COLOR statement is used to specify the color code and the brightness settings necessary to produce a given color.

COLOR = (color code, red brightness, green brightness, blue brightness)

The COLOR statement is used to specify the red, green, and blue brightness levels in values from 0 to 7 and assign these values to the specified color code.

For example, to assign a red brightness of 4, a green brightness of 3, and a blue brightness of 1 to color code 5, you would execute

```
COLOR=(5,4,3,1)
```

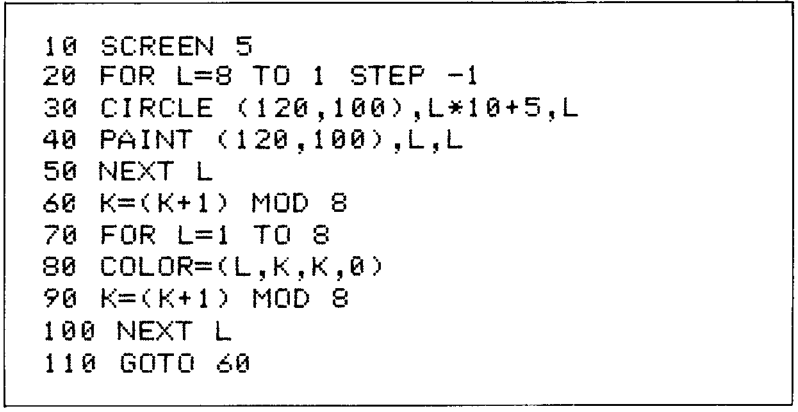
```
10 SCREEN 5
20 COLOR=(1,7,7,7)
30 FOR C=2 TO 9
40 COLOR=(C,0,0,C-2)
50 NEXT C
60 COLOR ,1,1:CLS
70 FOR CC=2 TO 9
80 R=100-CC*10
90 CIRCLE (125,100),R,CC
100 PAINT (125,95),CC
110 NEXT CC
120 GOTO 120
```

The colors for color codes 1 through 9 are specified in lines 20 through 50 in this program.

code	red	green	blue
1	7	7	7
2	0	0	0
3	0	0	1
4	0	0	2
5	0	0	3
6	0	0	4
7	0	0	5
8	0	0	6
9	0	0	7

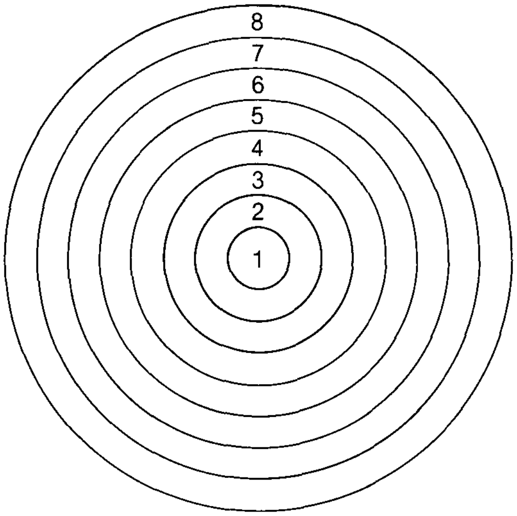
Code 1 is white, code 2 is black, and codes 3 through 9 change the color in steps from a blue color that is almost black to pure blue.
In the COLOR statement in line 60 color code 1 is specified for the background color and the border color, but color code 1 is now white instead of black. From line 70 on, circles of different shades of blue are drawn inward towards the center point, with the final circle colored pure blue.
The palette function can be used in this way to make a drawing which uses different shades of the color.
The above program uses the SCREEN 5 mode, which is the best mode for making graphic displays rapidly. The drawing speed made with graphic statements in the SCREEN 2 and SCREEN 4 modes is slower, and there is also a possibility of color spill. Color spill is explained in a later section.

Let's write another program which uses the palette function.



```
10 SCREEN 5
20 FOR L=8 TO 1 STEP -1
30 CIRCLE (120,100),L*10+5,L
40 PAINT (120,100),L,L
50 NEXT L
60 K=(K+1) MOD 8
70 FOR L=1 TO 8
80 COLOR=(L,K,K,0)
90 K=(K+1) MOD 8
100 NEXT L
110 GOTO 60
```

The following illustration shows the color codes for the concentric circles drawn by the FOR-NEXT loop in lines 20 to 50.



At first, the colors corresponding to color codes 1 to 8 are black, medium green, light green, dark blue, light blue, dark red, sky blue, and medium red. The circles are drawn in these colors.
Next, the palette function is used to change the colors corresponding to the code numbers. This is done in lines 60 through 110.
In the COLOR statememt in line 80, the variable K is used to specify the red brightness and the green brightness. The value of K is set in lines 60 and 90. The MOD used in these lines is one of the arithmetic signs, like +, −, *, and /.
a + b will add a and b, but **a MOD b gives the remainder of dividing a by b**. The following table shows the relationship between K and (K + 1), (K + 1) MOD 8.

K	K+1	(K+1) MOD 8
0	1	1 (1÷8=0 ... 1)
1	2	2 (2÷8=0 ... 2)
2	3	3 (3÷8=0 ... 3)
3	4	4 (4÷8=0 ... 4)
4	5	5 (5÷8=0 ... 5)
5	6	6 (6÷8=0 ... 6)
6	7	7 (7÷8=0 ... 7)
7	8	0 (8÷8=1 ... 0)

If no value is assigned to a variable, its value will be 0. Consequently, the value of K before line 60 is executed is 0, and when the line is executed it becomes 1. Then each time K=(K+1) MOD 8 is executed in line 90 and line 60, the value of K is increased by 1. After K becomes 7 it next becomes 0, and then increases up to 7 again.
Then as a result of the changes in the value of K and the COLOR statement inside the FOR—NEXT loop, the colors of the color codes from 1 to 8 are changed by the changes in the red and green brightness levels from 0 to 7. Throughout the program the blue brightness level remains at 0.

If line 80 were changed as follows:

```
80 COLOR=(L,0,K,K)
```

then the red brightness level would remain at 0 and the green and blue brightness levels would be changed. Green can also be set at 0, and the brightness of the other two colors changed. Try the various combinations and see the results on the screen.

THE SCREEN 6 MODE AND THE PALETTE FUNCTION

The palette function can be used in the SCREEN 6 mode also, but only the color codes from 0 to 3 can be used, or only 4 of the 512 colors. The SCREEN 6 colors are set as shown in the following table when BASIC is started up.

code	color
0	transparent
1	black
2	green
3	bright green

THE SCREEN 8 MODE AND COLOR

The palette function is not used in the SCREEN 8 mode, but 256 colors are available using the color codes.

In the SCREEN 8 mode red and green each have 8 levels of brightness from 0 to 7, and there are 4 levels of blue brightness, from 0 to 3. Since $8 \times 8 \times 4 = 256$, color codes from 0 to 255 can be used. A color code is determined by the following formula:

$$\textbf{color code} = \mathbf{32} \times \textbf{(green brightness)} + \mathbf{4} \times \textbf{(red brightness)} + \textbf{(blue brightness)}$$

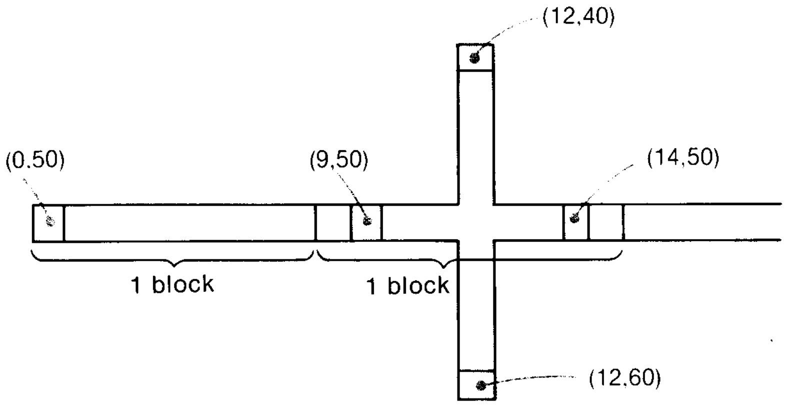
For instance, if green brightness is 1, red brightness 5, and blue brightness 3, the color code would be 55:

$$32 \times 1 + 4 \times 5 + 3 = 55$$

COLOR SPILL IN SCREEN 2 AND SCREEN 4 MODES

In the SCREEN 2 and SCREEN 4 modes, only 2 colors (including the background color) can be specified for one block of 8 horizontal dots. If more than two colors are specified, the color specified last becomes the valid color.

```
10 SCREEN 2
20 LINE (9,50)-(14,50),15
30 LINE (12,40)-(12,60),1
40 GOTO 40
```



In this program, a horizontal line is drawn from X9 to X14 in the horizontal block of 8 dots extending from X8 to X15. Then a vertical line is drawn at X12 from Y40 to Y60, which intersects the horizontal line. This adds a third color to the horizontal block from X8 to X15, and therefore even though white has been specified, the horizontal line is displayed as black, since black, as the last color specified, has become the valid color for this block.

When the specified color becomes a different color in this way, it is called color spill. Care is required in assigning colors in the SCREEN 2 and SCREEN 4 modes to avoid such color spill.

If line 20 is revised as follows:

```
LINE (8,50)-(15,50)
```

the horizontal line will then fill the entire block of eight dots from X8 to X15 and the color specified (white) will remain valid.
However, if another color is subsequently specified for this same block, with a statement such as

```
PSET (8,50),8
```

then the color of the entire block will change to the color last specified (in the case of the PSET statement, color code 8).

Always remember that in the SCREEN 2 and SCREEN 4 modes, a maximum of two colors only can be used in any given block of 8 dots.

In SCREEN modes 5 to 8, colors can be freely specified in units of 1 dot each.

RETURNING THE COLOR SPECIFICATIONS TO THE INITIAL SETTINGS COLOR

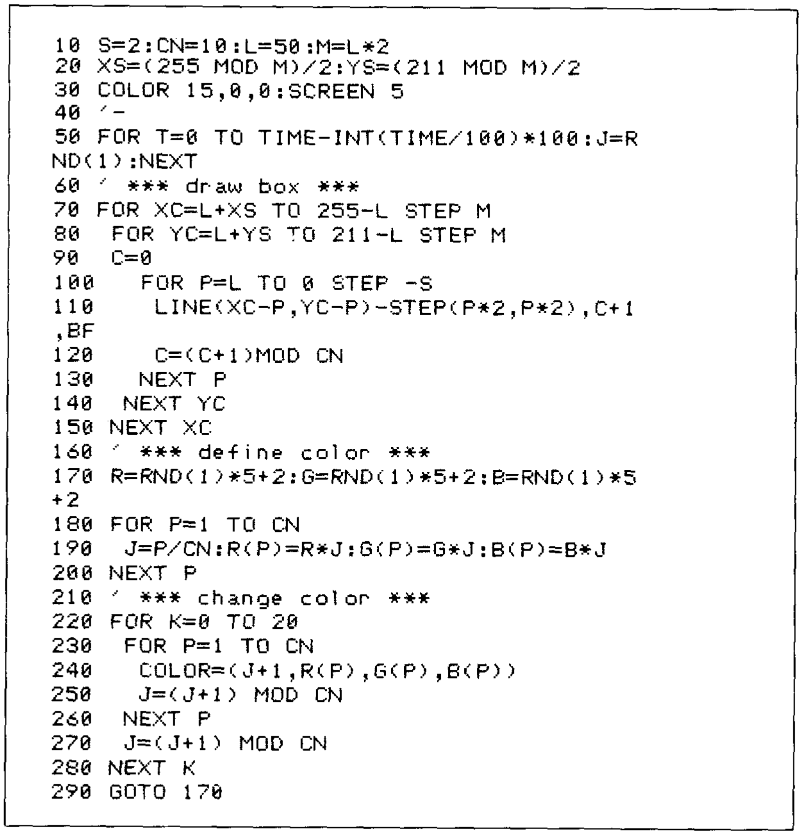
Color codes and the color specifications were changed using the COLOR statement. To return the specifications within a program to their initial status when BASIC is started up

COLOR = NEW

is executed.

Sample Program

The following program uses the palette function in the SCREEN 5 mode.

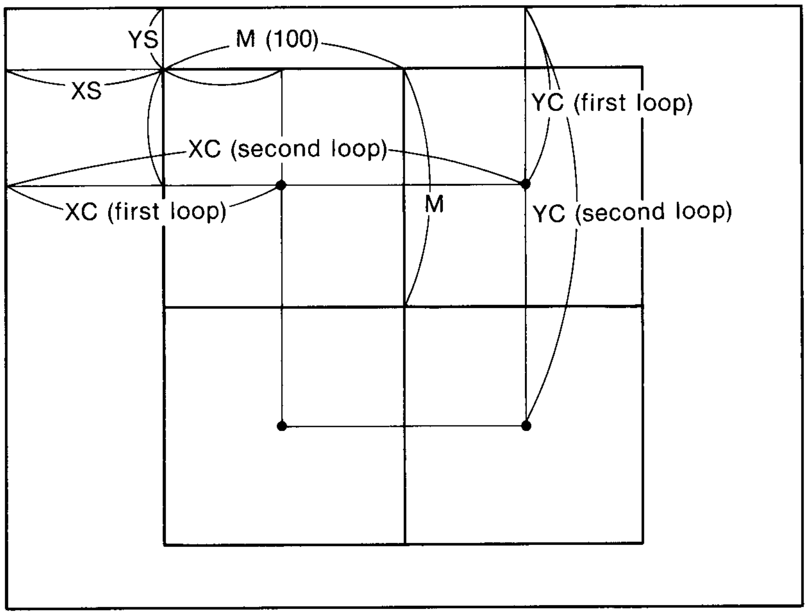


```
10 S=2:CN=10:L=50:M=L*2
20 XS=(255 MOD M)/2:YS=(211 MOD M)/2
30 COLOR 15,0,0:SCREEN 5
40 '-
50 FOR T=0 TO TIME-INT(TIME/100)*100:J=R
ND(1):NEXT
60 ' *** draw box ***
70 FOR XC=L+XS TO 255-L STEP M
80  FOR YC=L+YS TO 211-L STEP M
90  C=0
100   FOR P=L TO 0 STEP -S
110    LINE(XC-P,YC-P)-STEP(P*2,P*2),C+1
,BF
120    C=(C+1)MOD CN
130   NEXT P
140  NEXT YC
150 NEXT XC
160 ' *** define color ***
170 R=RND(1)*5+2:G=RND(1)*5+2:B=RND(1)*5
+2
180 FOR P=1 TO CN
190  J=P/CN:R(P)=R*J:G(P)=G*J:B(P)=B*J
200 NEXT P
210 ' *** change color ***
220 FOR K=0 TO 20
230  FOR P=1 TO CN
240   COLOR=(J+1,R(P),G(P),B(P))
250   J=(J+1) MOD CN
260  NEXT P
270  J=(J+1) MOD CN
280 NEXT K
290 GOTO 170
```

There are several commands used in this program which have not yet been explained, but input them as they are written and run the program.

The following is a brief description of the program.

First, lines 70 to 150 draw squares. The variables used in these lines are defined in lines 10 and 20. The following illustration shows how the variables are used.



Array variables R(P), G(P), and B(P) are used in lines 170 to 200. Values from 2 to 7 are assigned to R(1)—R(10), G(1)—G(10), and B(1)—B(10). The RND function in line 170 determines what value is assigned. The RND function gives a positive number greater than 0 and less than 1. Functions are explained in Chapter 7.

In lines 220 to 280 the values of R(P), G(P), and B(P) are used to change the colors of color codes 1 to 10 using the palette function.

The single quote mark (') in lines 40, 60, 160, and 210 is used in place of REM. **The REM or (') statememt is used to write a remark in the program which is not executed as part of the program.**

SETTING PAGES

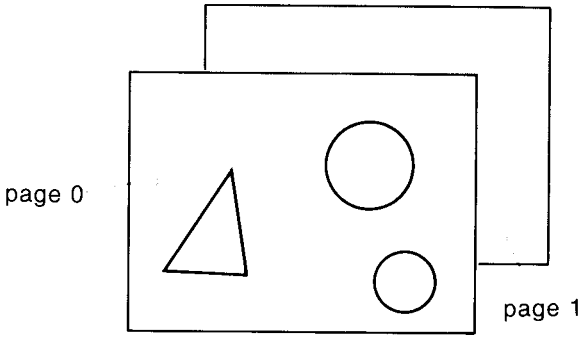
- ●Concerning Pages
- ●The Display Page and the Active Page
- ●Setting Pages—SET PAGE

GRAPHIC MODE DISPLAYS AND PAGES

Take another look at the SCREEN Mode Chart on page 90. It has a column title "Page." Modes from SCREEN 5 through 8 in the graphic mode use "page." This part of the chart can be rewritten as follows:

Mode	VRAM 64K	VRAM 128K
SCREEN 5	2 pages	4 pages
SCREEN 6	2 pages	4 pages
SCREEN 7	—	2 pages
SCREEN 8	—	2 pages

As the name implies, page is like the page in a notebook. For example, for 64K VRAM computers, two pages can be used in the SCREEN 5 and SCREEN 6 modes. When a drawing is drawn on the screen with graphic statements, only one of the two pages is used, and the other page remains blank.



For 64K VRAM, SCREEN 5, SCREEN 6

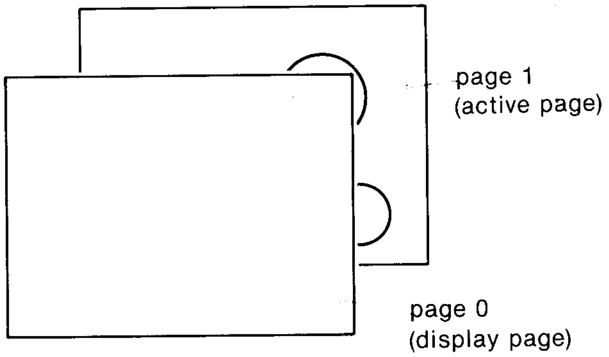
As shown in the above illustration, when two pages are used, they are given **page numbers**. One is called page 0 and the other is called page 1. With a 128K VRAM, four pages can be used in SCREEN 5 and SCREEN 6. They are called page 0, page 1, page 2, and page 3.

The Display Page and the Active Page

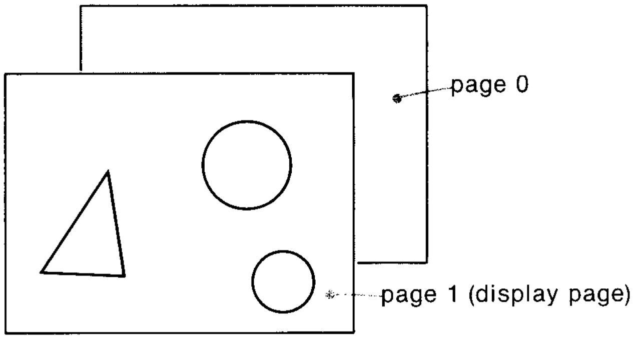
In the modes in which 2 pages or 4 pages can be used, page 0 is always the page on which drawings can be drawn and the page that is displayed on the monitor TV at the time BASIC is started up.
The page on which drawings can be drawn is called the **active page**.
The page that you see on the monitor TV is called the **display page**.

EFFECTS THAT CAN BE ACHIEVED BY SETTING THE PAGE

Unless specified otherwise, the display page and the active page are both page 0. Therefore, when a statement such as LINE, CIRCLE, or DRAW is executed, the figure will be drawn on page 0, and displayed as is on the monitor TV. But if page 0 is set as the display page, and page 1 set as the active page, and graphic statements are then executed, the figures will be drawn on page 1. But since what you see on the monitor TV is page 0, the figures will not be displayed on the screen.



If page 1 is made the display page, then the figures drawn on it will be displayed on the screen.

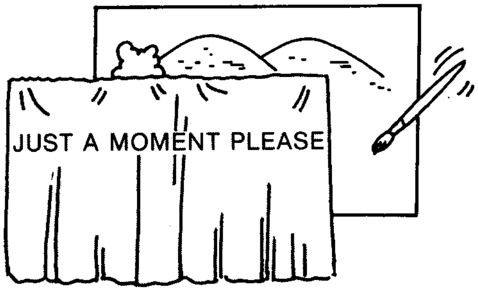


At this time, if page 0 is set as the active page, then subsequent figures will be drawn on page 0. If page 1 is made the active page, then the subsequent figures drawn will be added to page 1.

Using Pages

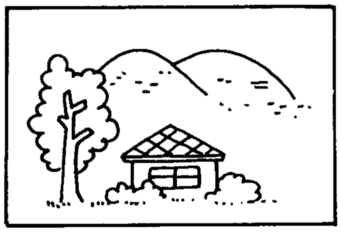
There are many ways to utilize the page setting function. Two interesting effects that can be achieved are described below.

- **Show only the completed drawing on the screen.**
  It takes time for a complicated drawing to be drawn on the screen, especially if a lot of PAINT statements are used. If you don't want to show the drawing being drawn on the screen, you can set different pages for the display page and the active page, and then draw the drawing on the active page. When the drawing is completed, you can change the active page to the display page, and the completed drawing will be displayed all at once on the screen.

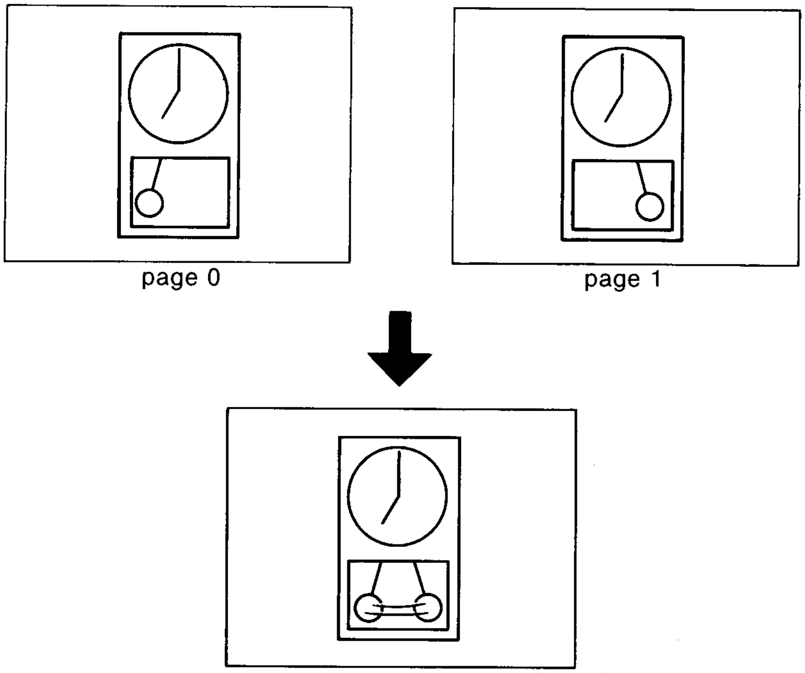


Don't show what's going on backstage!

The completed picture appears all at once!



- **Draw a different drawing on each page, and then shift back and forth between pages to create an effect of movement.**
  For example, the following effect can be achieved:



Change the display page back and forth from page 0 to page 1

SETTING PAGES SET PAGE

The SET PAGE statement sets page numbers as the display page and the active page.

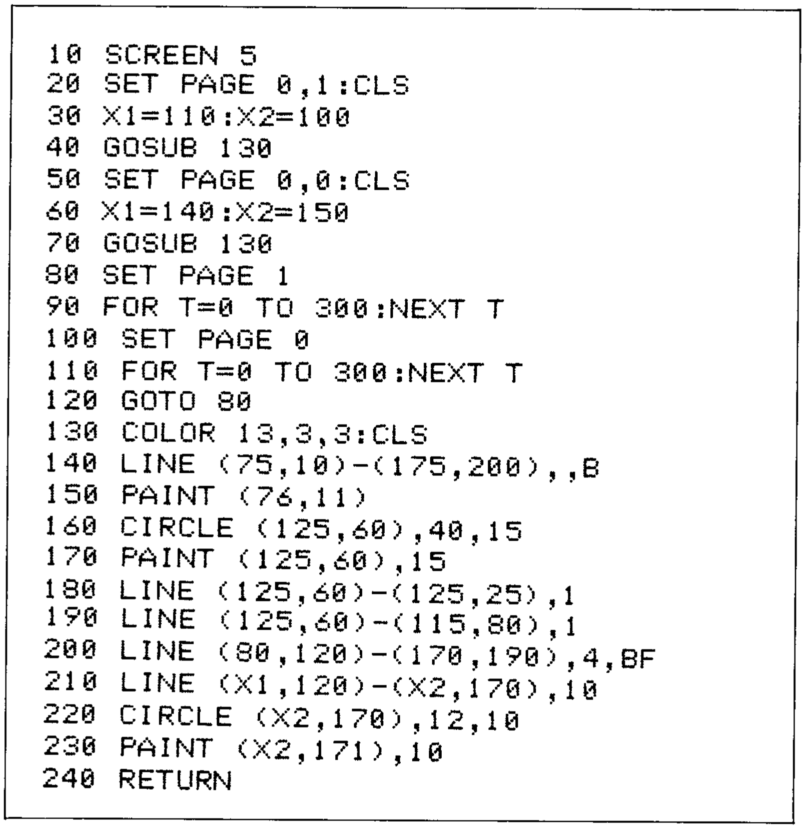
SET PAGE [display page], [active page]

For example, to set page 0 as the display page and page 1 as the active page, execute

```
SET PAGE 0,1
```

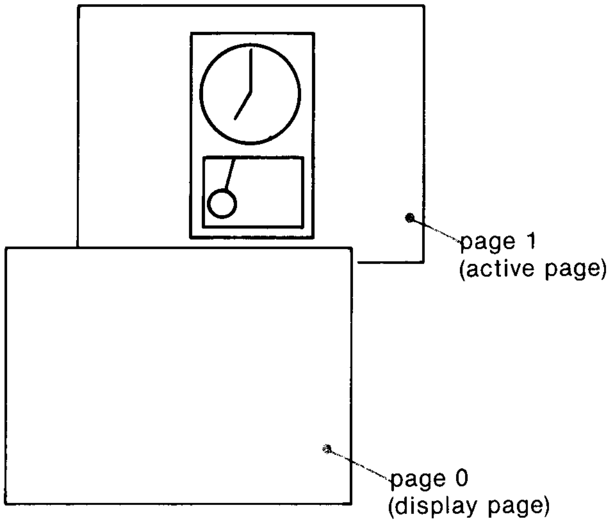
After this SET PAGE statement is executed, page 0 is the page that is seen, and page 1 is the page that figures are drawn on.

Changing the Pages
The following program changes the pages back and forth.

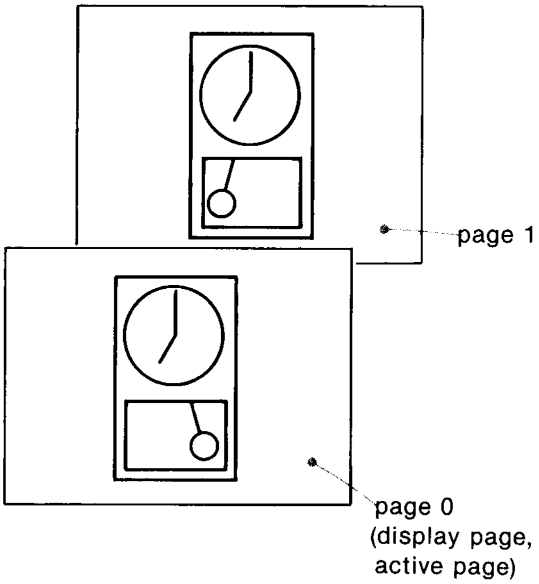


```
10 SCREEN 5
20 SET PAGE 0,1:CLS
30 X1=110:X2=100
40 GOSUB 130
50 SET PAGE 0,0:CLS
60 X1=140:X2=150
70 GOSUB 130
80 SET PAGE 1
90 FOR T=0 TO 300:NEXT T
100 SET PAGE 0
110 FOR T=0 TO 300:NEXT T
120 GOTO 80
130 COLOR 13,3,3:CLS
140 LINE (75,10)-(175,200),,B
150 PAINT (76,11)
160 CIRCLE (125,60),40,15
170 PAINT (125,60),15
180 LINE (125,60)-(125,25),1
190 LINE (125,60)-(115,80),1
200 LINE (80,120)-(170,190),4,BF
210 LINE (X1,120)-(X2,170),10
220 CIRCLE (X2,170),12,10
230 PAINT (X2,171),10
240 RETURN
```

This program draws different drawings on page 0 and page 1 in the SCREEN 5 mode. First, page 0 is set as the display page and page 1 is set as the active page (line 20).
Then the subroutine from line 130 draws the following drawing.



Then the display page and the active page are both set at page 0 (line 50), and the following drawing is drawn.



The page 1 drawing remains as it is. The loop from line 80 to line 120 then changes the display page back and forth from page 1 to page 0, which results in the two drawings being displayed alternately on the screen. This gives the impression that the pendulum of the clock is moving back and forth.

Specification of the active page has been omitted in the SET PAGE statements in lines 80 and 100. When the specification is omitted, the previous specification remains in effect.

COPYING GRAPHICS DATA

- Copying Graphics—COPY
- Copying Between Screens
- Copying Between a Screen ɑnd Internal Memory
- Copying Between a Screen and a Disk File
- Copying Between Memory and a Disk File
- Logical Operations

COPYING GRAPHICS

Drawings drawn in the SCREEN 5 to SCREEN 8 graphic modes can be copied. In copying an area of the screen is specified, and the color data for each dot within that area is copied to another place. There are three places to which such data can be copied:

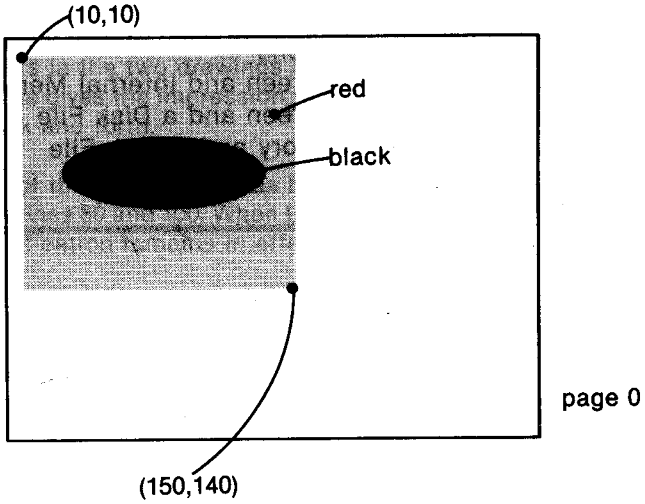
- the screen (VRAM)
- internal memory (an array variable)
- a floppydisk (file)

The COPY statement is used to copy graphic data.

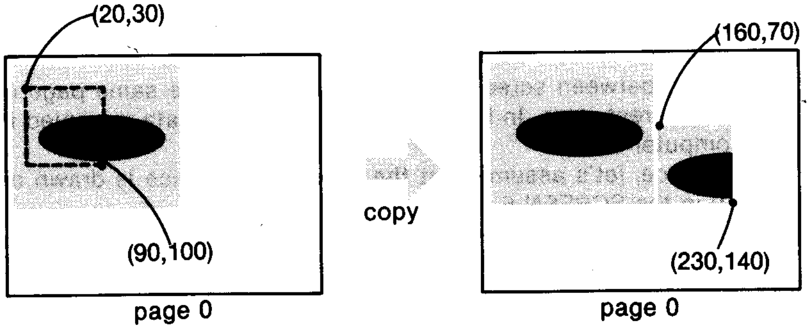
COPYING BETWEEN SCREENS COPY (1)

In copying between screens, you can copy onto the same page or onto a different page. In both cases, the graphics data is copied in the computer VRAM.

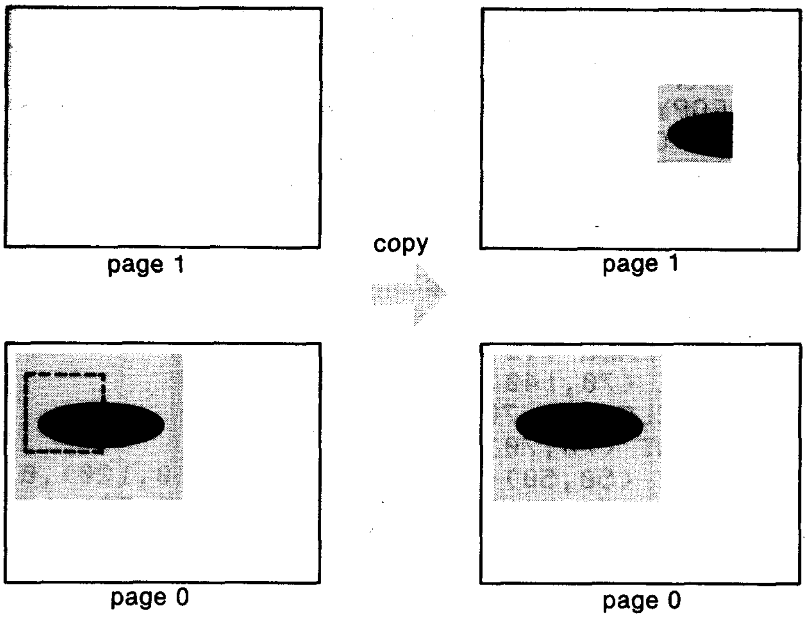
For instance, let's assume that the following graphics is drawn on page 0 in the SCREEN 5 mode.



If the part of this graphics from (20,30) to (90,100) (shown by the dotted lines in the below illustration) is copied to the same page in the area that has (160,70) as its top left coordinate, the following result will be obtained.



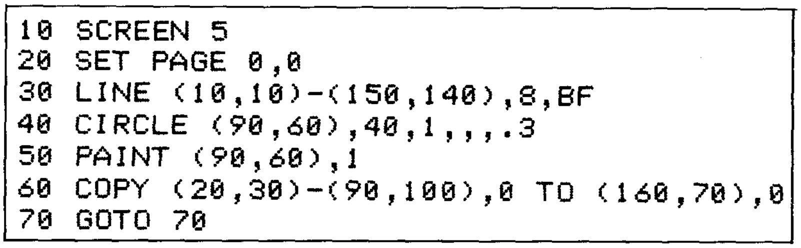
You can also copy to a different page. If you copy the same part of the graphics as was copied above to the same location on page 1, the result will be as shown below.



When you copy, the figures and colors in the foreground are copied to the designated destination area, while the original figures and colors also remain on the original source page.
The COPY statement format is:

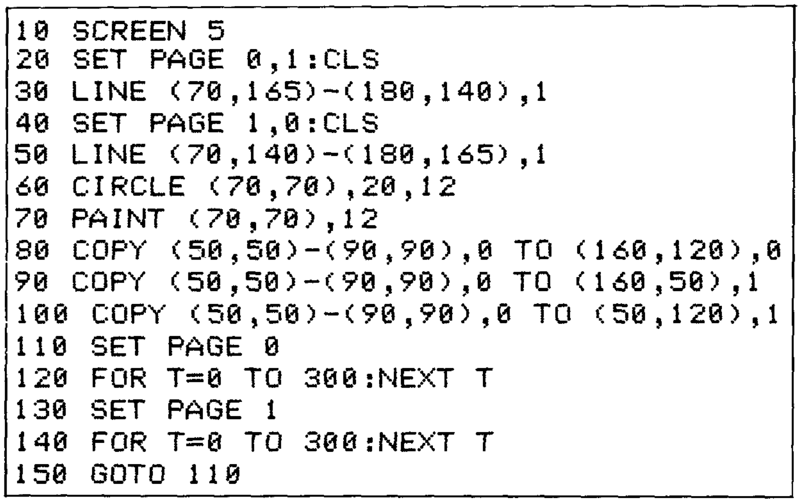
```
COPY (X1,Y1)—(X2,Y2)[,source page]TO(X3,Y3)
[,destination page]
```

(X1,Y1) is the top left coordinate of the source area, (X2,Y2) is the bottom right coordinate of the source area, and (X3,Y3) is the top left coordinate of the destination area. If the source page and/or the destination page are not specified, the active page is assumed to be specified.



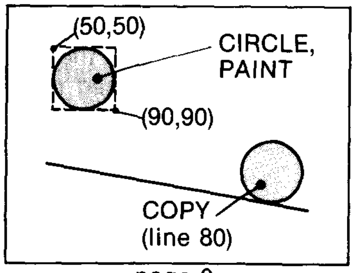
```
10 SCREEN 5
20 SET PAGE 0,0
30 LINE (10,10)-(150,140),8,BF
40 CIRCLE (90,60),40,1,,,.3
50 PAINT (90,60),1
60 COPY (20,30)-(90,100),0 TO (160,70),0
70 GOTO 70
```

Line 60 copies the area (20,30)—(90,100) on page 0 to the area with the top left coordinate of (160,70) on page 0.

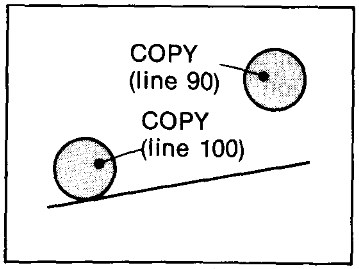


```
10 SCREEN 5
20 SET PAGE 0,1:CLS
30 LINE (70,165)-(180,140),1
40 SET PAGE 1,0:CLS
50 LINE (70,140)-(180,165),1
60 CIRCLE (70,70),20,12
70 PAINT (70,70),12
80 COPY (50,50)-(90,90),0 TO (160,120),0
90 COPY (50,50)-(90,90),0 TO (160,50),1
100 COPY (50,50)-(90,90),0 TO (50,120),1
110 SET PAGE 0
120 FOR T=0 TO 300:NEXT T
130 SET PAGE 1
140 FOR T=0 TO 300:NEXT T
150 GOTO 110
```

This program draws the figures shown below on page 0 and page 1 in the SCREEN 5 mode. The first circle is drawn on page 0 with the CIRCLE and PAINT statements in lines 60 and 70. The rest of the circles are drawn by being copied with the COPY statement (lines 80,90, 100).



page 0

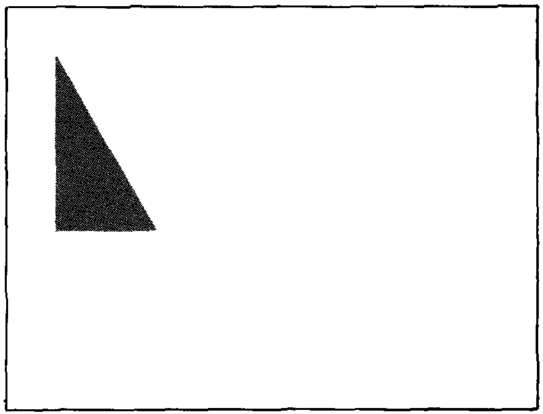


page 1

COPYING BETWEEN THE SCREEN AND INTERNAL MEMORY COPY (2)

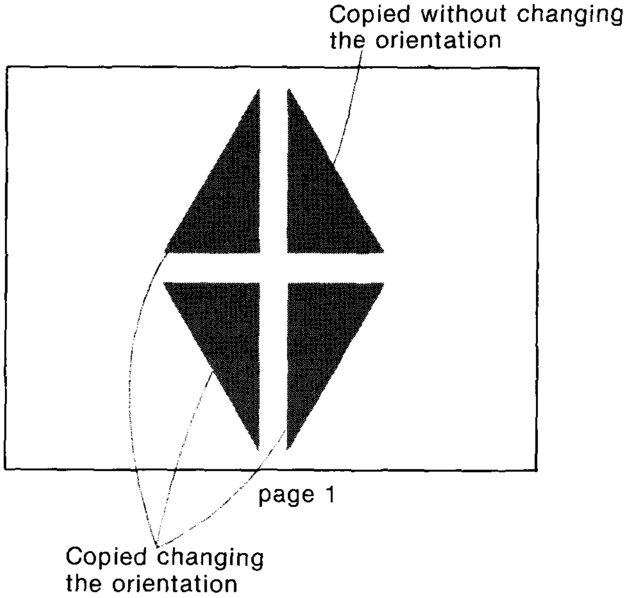
Screen data is retained in the VRAM, but it can also be copied to internal memory (RAM). Also, data that has been copied to internal memory can be copied back to the screen (VRAM) and re-displayed. When data is copied back to the VRAM, the orientation of the data on the screen can be changed.

For example, let's say you draw the following figure on page 0 in the SCREEN 5 mode.



page 0

Then if you copy the screen area from (20,20)—(60,105) to the internal memory, you can copy the following display to page 1.


Copied without changing the orientation
page 1
Copied changing the orientation

Before copying graphics data to memory, you must define a numeric type array variable with the DIM statement to receive the data.

The following COPY statement format is used to copy graphics data to memory.

COPY(X1,Y1)—(X2,Y2)[,source page] TO array variable name

(X1,Y1) is the top left coordinate, and (X2,Y2) is the bottom right coordinate, of the area to be copied.
The size of the array variable is determined with the following formula:

INT ((((ABS(X1 - X2) + 1) * (ABS(Y1 - Y2) + 1) * pixel size + 7)/8 + 4)/8) + 1

INT and ABS are BASIC functions. They are explained in Chapter 7.
Pixel size (the number of pixels equal to one dot on the screen) differs according to the SCREEN mode.

Mode	Pixel size
SCREEN 5	4
SCREEN 6	2
SCREEN 7	4
SCREEN 8	8

For instance, to copy the data for the area (20,20)—(60,105) in the SCREEN 5 mode, since X1 is 20 and Y1 is 20, and X2 is 60 and Y2 is 105, X1 − X2 = −40 and Y1 − Y2 = −85. Therefore the size of the array variable would be computed as follows:

```
INT((((ABS(-40)+1)*(ABS(-85)+1)*4+7)
/8+4)/8)+1
```

Consequently, the following two lines would be written at the beginning of the program to define an array variable P.

```
S=INT((((ABS(-40)+1)*(ABS(-85)+1)
*4+7)/8+4)/8)+1
DIM P(S)
```

Once this kind of array variable has been defined, the COPY statement can be used to copy the data for the area (20,20)—(60,105) to memory, that is, to the array variable.
It is only necessary to specify P in the COPY statement as follows:

```
COPY (20,20)-(60,105),0 TO P
```

The following COPY statement is used to copy data from memory (array variable) to the screen (VRAM).

COPY array variable name [,orientation] TO(X3,Y3) [,destination page]

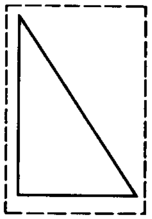
(X3,Y3) is the start location for drawing the data to be copied to the destination page.

There are four "orientations," specified by the numbers 0 through 3. "Orientation" indicates the direction in which the drawing should be drawn from the start location.

"Orientation" no.	Drawing direction
0	from top left to bottom right ↘
1	from top right to bottom left ↙
2	from bottom left to top right ↗
3	from bottom right to top left ↖

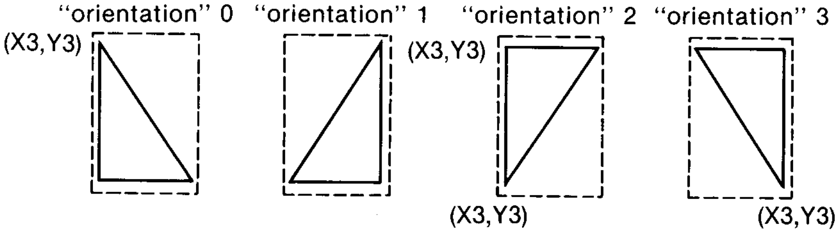
0 is assumed when specification is omitted.

For example, when the data for the following graphics has been copied to memory,



The area within the dotted lines is copied to memory

the data can be copied back to the screen using the four orientation numbers and the (X3,Y3) drawing start locations as follows:

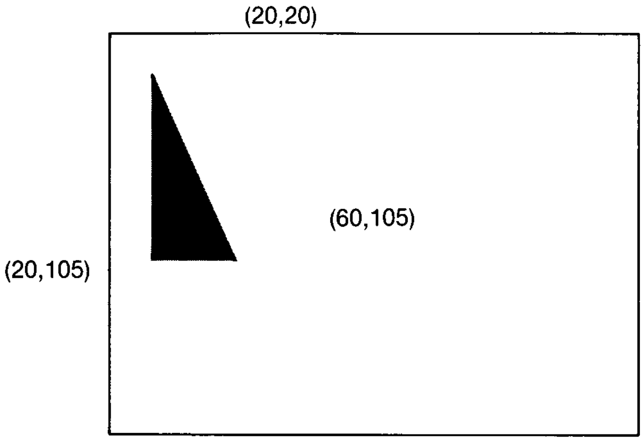


If the orientation number is omitted, the default setting is 0, from top left to bottom right.

Let's write a program which displays the example given on page 128.

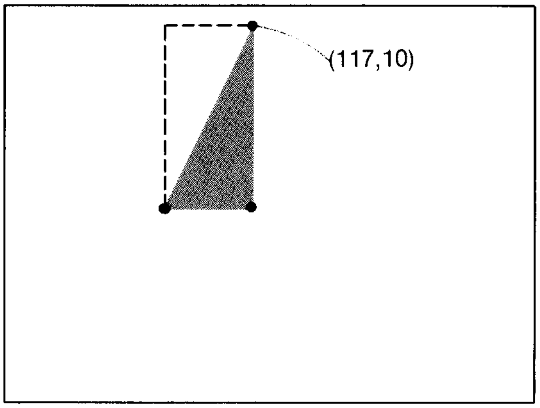
```
10 S=INT((((ABS(-40)+1)*(ABS(-85)+1)*4+7
)/8+4)/8)+1
20 DIM P(S)
30 SCREEN 5
40 SET PAGE 0,1:CLS
50 SET PAGE 0,0:CLS
60 LINE (20,20)-(20,105)
70 LINE (20,20)-(60,105)
80 LINE (20,105)-(60,105)
90 PAINT (25,50)
100 COPY (20,20)-(60,105),0 TO P
110 COPY P,0 TO (138,10),1
120 COPY P,1 TO (117,10),1
130 COPY P,2 TO (138,200),1
140 COPY P,3 TO (117,200),1
150 SET PAGE 1
160 GOTO 160
```

The array variable P is defined in lines 10 and 20. Then the following drawing is drawn on page 0 in the SCREEN 5 mode.



page 0

The COPY statement in line 100 copies the area which includes this drawing, (20,20)—(60,105), to the array variable in memory. Then lines 110 to 140 change the orientation of the data that has been copied to memory on page 1.
For example, line 120 copies the data using "orientation" 1 and a drawing start location of (117,10).



page 1

The other COPY statements perform similar functions, and a drawing like the one shown on page 128 is drawn on page 1.

COPYING BETWEEN THE SCREEN AND A FLOPPYDISK COPY (3)

The COPY statement can also be used to copy data that has been drawn on the screen (VRAM) to a floppydisk file with computers which have an internal disk drive or a floppydisk drive unit attached. The COPY statement format for copying (saving) graphics data on a floppydisk is:

COPY (X1,Y1)—(X2,Y2) [, source page] TO "[drive name], file name [.type name]"

The rules for assigning a file name are exactly the same as those for saving a program. The type name can be omitted, but it is convenient to always include it so that you will know what kind of file it is. In the following program example, .PIC is used as the type name.
Now let's write a program to copy on a floppydisk the same figure we previously copied to memory.

```
10 SCREEN 5
20 SET PAGE 0,0:CLS
30 LINE (20,20)-(20,105)
40 LINE (20,20)-(60,105)
50 LINE (20,105)-(60,105)
60 PAINT (25,50)
70 COPY (20,20)-(60,105),0 TO "TRIANGLE.
PIC"
```

A triangle is drawn by lines 30 through 60. Then line 70 copies the data of the area (20,20)—(60,105) which includes the triangle to the floppydisk file. The file name is TRIANGLE, and the type name is .PIC. When this program is executed, a triangle is drawn on page 0 in the SCREEN 5 mode. Immediately after the triangle is drawn, the disk drive begins to operate, and the drawing data is copied to the disk. When the data is copied, the program also ends, and Ok is displayed on the screen.
If the FILES command is executed, the file name and type name

```
TRIANGLE.PIC
```

will be displayed, and you can check that the graphics data has in fact been copied to the floppydisk file.

The COPY statement format for copying graphics data from a floppydisk file back to the screen (VRAM) is:

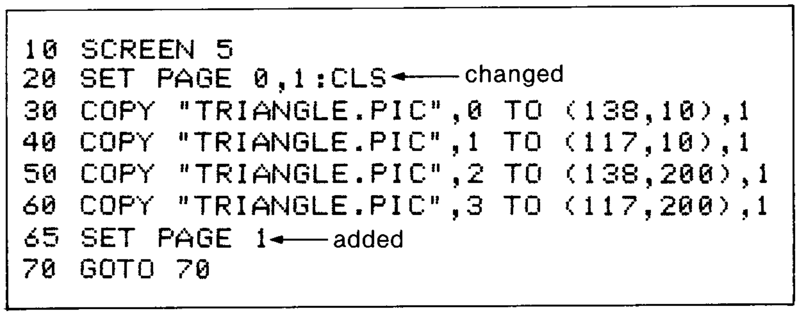
COPY "[drive name] file name [. type name]" [, orientation] TO (X3,Y3) [, destination page]

The same file name and type name that was used when the data was copied to the disk is specified. The "orientation" and drawing start coordinates are just like those used in copying from memory.
The following program changes the data copied to the disk to four different orientations and copies this data to the screen.

```
10 SCREEN 5
20 SET PAGE 1,1:CLS
30 COPY "TRIANGLE.PIC",0 TO (138,10),1
40 COPY "TRIANGLE.PIC",1 TO (117,10),1
50 COPY "TRIANGLE.PIC",2 TO (138,200),1
60 COPY "TRIANGLE.PIC",3 TO (117,200),1
70 GOTO 70
```

The COPY statements in lines 30 to 60 copy the drawing with different drawing start locations and orientations to page 1 in the SCREEN 5 mode. When you execute this program you will see how the data is copied.

In the above program both the display page and the active page were set at page 1. But if different pages are set for the display page and the active page and the image data is first copied from the floppydisk to the active page, and then when the copying is completed the active page is changed to the display page, the completed drawing will be displayed all at once on the screen.
Revise the previous program as follows, and execute it.



```
10 SCREEN 5
20 SET PAGE 0,1:CLS ←— changed
30 COPY "TRIANGLE.PIC",0 TO (138,10),1
40 COPY "TRIANGLE.PIC",1 TO (117,10),1
50 COPY "TRIANGLE.PIC",2 TO (138,200),1
60 COPY "TRIANGLE.PIC",3 TO (117,200),1
65 SET PAGE 1 ←— added
70 GOTO 70
```

COPYING BETWEEN MEMORY AND A FLOPPYDISK COPY (4)

The COPY statement format for copying graphics data saved on a floppydisk to an array variable in memory is as follows:

COPY "[drive name] file name [. type name]" TO array variable name

To copy graphics data from an array variable in memory to a floppydisk use the following format:

COPY array variable name TO "[drive name] file name [. type name]"

LOGICAL OPERATIONS

In copying graphics data with the COPY statement, logical operations can be performed between the color code of the drawing color and the color code of the destination screen.
The following ten logical operations can be used with the COPY statement.

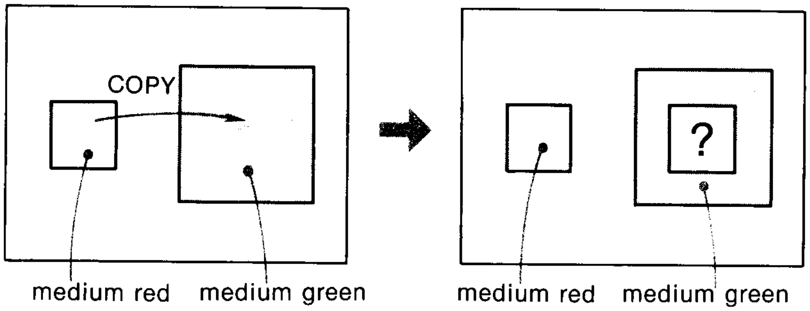
PSET, PRESET, XOR, OR, AND

TPSET, TPRESET, TXOR, TOR, TAND

The result of the logical operation can be seen when the color codes are changed to binary codes. The following chart shows the binary color codes of the 16 colors in SCREEN 5 mode when BASIC is started up.

Color	Color code (decimal)	Color code (binary)	Color	Color code (decimal)	Color code (binary)
transparent	0	0000	medium red	8	1000
black	1	0001	light red	9	1001
medium green	2	0010	dark yellow	10	1010
light green	3	0011	light yellow	11	1011
dark blue	4	0100	dark green	12	1100
light blue	5	0101	magenta	13	1101
dark red	6	0110	gray	14	1110
sky blue	7	0111	white	15	1111

Let's consider what happens when a medium red figure (color code 1000) is copied onto a medium green (color code 0010) figure.



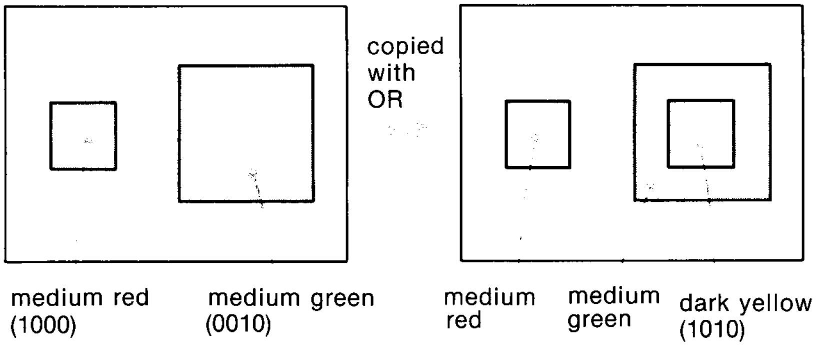
If no logical operation is performed, the medium red square will be superimposed on the green square. But when a logical operation is performed, the color of the $\boxed{?}$ portion of the medium green square will be a color other than medium red. The color the square becomes depends on which logical operation is performed.
Let's take the logical operation OR as an example. OR performs the following operations on each digit (0 or 1) of the two binary color codes.

X	Y	result of X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

In the above example, the OR operation is performed on each pair of digits in the medium red color code 1000 and the medium green color code 0010 as shown below.

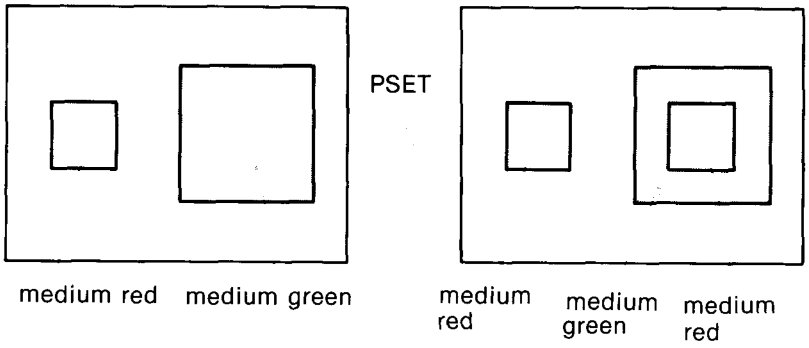
```
        medium red........1000
        medium green...0010
OR...................................1010
```

The result of the OR operation is 1010, which, as shown in the above color code table, is dark yellow. Consequently, when a medium red figure is copied on a medium green figure with the logical operation OR, the color of the copied figure will be dark yellow.

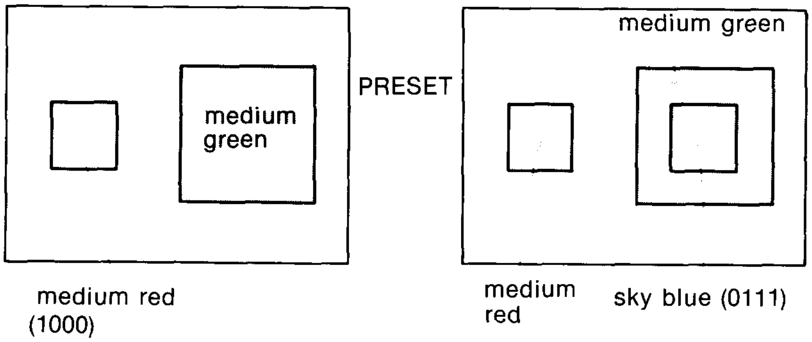


The results obtained by using logical operations other than OR are shown below.

- PSET—the color copied remains the same, regardless of the color of the destination area.



- PRESET—changes each digit of the color code of the color copied to the opposite number—0 to 1, and 1 to 0— with no reference to the color of the destination area. For instance, if medium red (1000) is copied with PRESET, the color will become sky blue (0111).

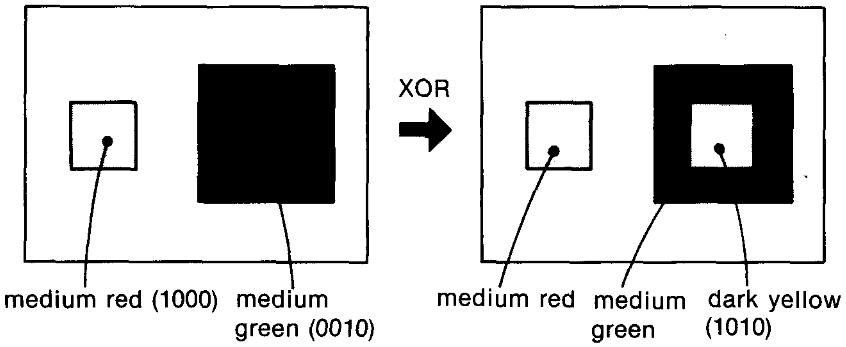


- XOR—the following operations are performed on the color code of the color copied and the color code of the destination area color.

X	Y	result of X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

For example, when the XOR operation is performed on medium red (1000) and medium green (0010), the result is dark yellow (1010).

$$\begin{array}{r}1000\\ \underline{0010}\\ 1010\end{array}$$

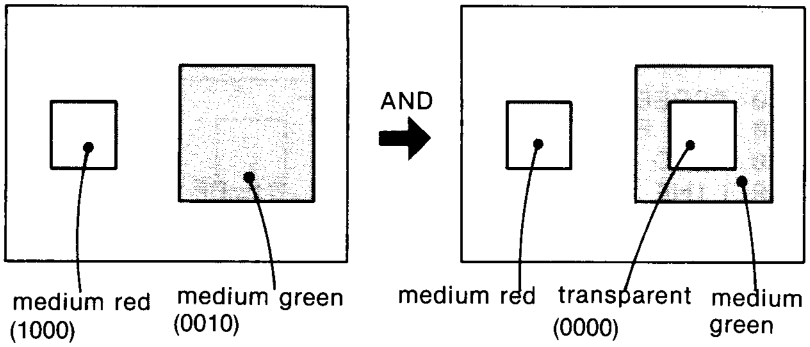


- AND—performs the following operations on the color code of the copied color and the color code of the destination area.

X	Y	result of X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

When the AND operation is performed on medium red (1000) and medium green (0010), the result is the transparent color (0000).

$$\begin{array}{l} 1000 \\ \underline{0010} \\ 0000 \end{array}$$



- TPSET, TPRESET, TOR, TXOR, TAND—these operations are the same as PSET, PRESET, OR XOR, and AND, except that when a color is transparent, if T is prefixed it will remain transparent after being copied, regardless of the color of the destination area.

Using Logical Operations with the COPY Statement
The following COPY statement formats are used when logical operations are performed.

- **From Screen to Screen**

  COPY (X1,Y1)—(X2,Y2) [, source page] TO (X3,Y3) [, destination page] [, **logical operation**]

  Example: COPY (10,10)—(100,100), 0 TO (30,30), 1, XOR

- **From Memory (Array Variable) to Screen**

  COPY array variable name [, orientation] TO (X3,Y3) [, destination page] [, **logical operation**]

  Example: COPY P, 1 TO (30,30), 0, TAND

- **From Floppydisk (File) to Screen**

  COPY "[drive name] file name [. type name]" [, orientation] TO (X3,Y3) [, destination page] [, **logical operation**]

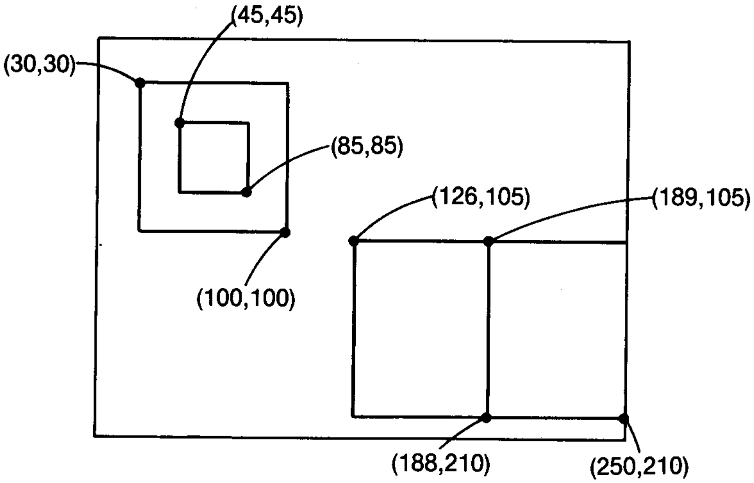
  Example: COPY "TRIANGLE .PIC", 2 TO (30,30), 1, PRESET

When the logical operation is omitted, it is the same as specifying PSET.

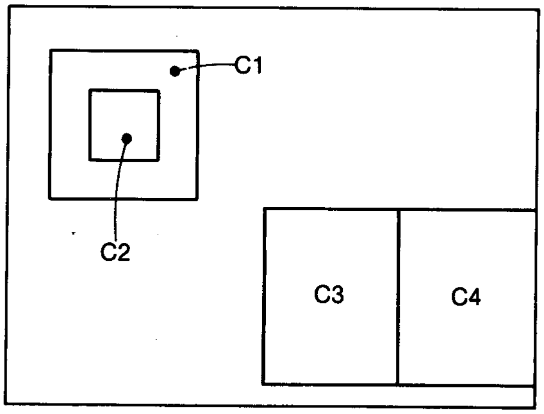
Now let's make a program which uses a logical operation in copying a drawing from screen to screen.

```
10 SCREEN 5
20 SET PAGE 0,0:CLS
30 READ C1,C2,C3,C4
40 LINE (30,30)-(100,100),C1,BF
50 LINE (45,45)-(85,85),C2,BF
60 LINE (126,105)-(188,210),C3,BF
70 LINE (189,105)-(250,210),C4,BF
80 COPY (30,30)-(100,100),0 TO (153,124)
,0,OR
90 GOTO 90
100 DATA 8,3,10,2
```

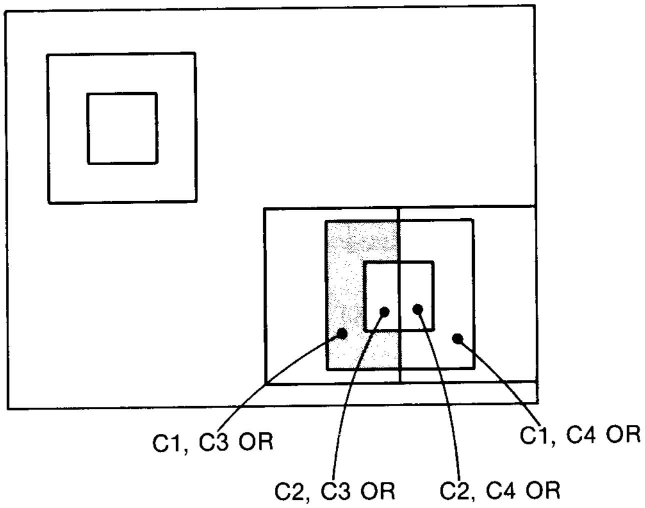
This program makes the following drawing.



The colors are determined by the values assigned to the C1, C2, C3, and C4 variables.

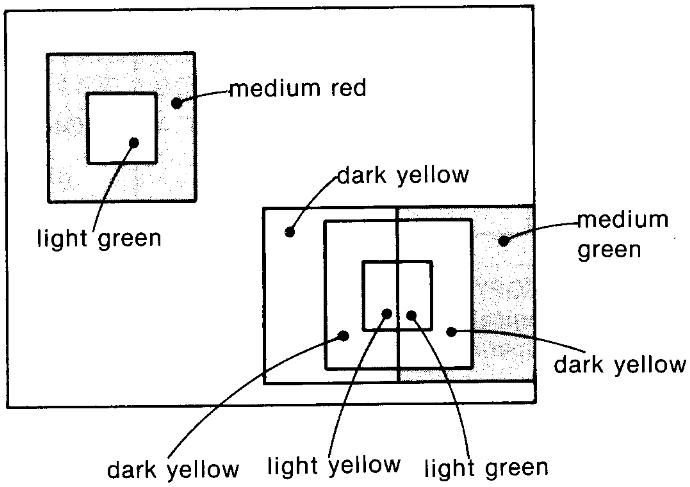


Then the COPY statement in line 80 copies the drawing. During the copying, logical operations are performed on the parts of the drawing that are superimposed.



This program shows the results of logical operations which combine four different colors. The colors assigned to the variables in the program are: C1, medium red; C2, light green; C3, dark yellow; and C4 medium green (the DATA statement in line 100). OR is the logical operation performed (line 80).

Running the program produces the following results:



THE SCREEN STATEMENT

- SCREEN Statement Settings
- Key Click On/Off
- The Cassette Tape Interface Baud Rate
- Printer Type
- Interlace

THE SCREEN STATEMENT

The SCREEN statement is used to make a number of other settings, in addition to the SCREEN mode.

SCREEN [mode], [sprite size], [key click switch], [baud rate], [printer type], [interlace]

KEY CLICK SWITCH, BAUD RATE, PRINTER TYPE

Key Click On/Off

The third parameter (key click switch) of the SCREEN statement is used to specify whether a sound will be made when the keyboard keys are pressed. There is no sound when the key click switch is set at 0. When it is set at any other number (1—255), a click sound is produced when a key is pressed.

```
SCREEN ,,0 —— key click sound off
SCREEN ,,1 —— key click sound on
```

The Cassette Baud Rate Setting

The baud rate is the number of bits per second that data is transmitted. A baud rate of 1200 means that 1200 bits of data are transmitted each second. With a baud rate of 2400, 2400 bits of data are transmitted per second.

The fourth parameter of the SCREEN statement specifies the baud rate for data transmission to and from the cassette tape recorder. Baud rate 1 is 1200 bauds, and baud rate 2 is 2400 bauds. The initial default setting is 1 (1200 baud).

```
SCREEN ,,,1 —— 1200 bauds
SCREEN ,,,2 —— 2400 bauds
```

If the baud rate is set at 2400 by the SCREEN ,,,2 statement before saving a program on cassette tape, the 2400 baud rate must be set with the SCREEN ,,,2 before loading the program back from the tape.

Printer Type

The fifth parameter of the SCREEN statement makes the setting to conform with the type of printer used. 0 is the setting for MSX-type printers. A setting other than 0 (1—255) is made for other types of printers. The initial default setting is 0 (MSX-type printer).

MSX-type printers are especially designed for MSX computers, and have a font of the special MSX graphic characters.

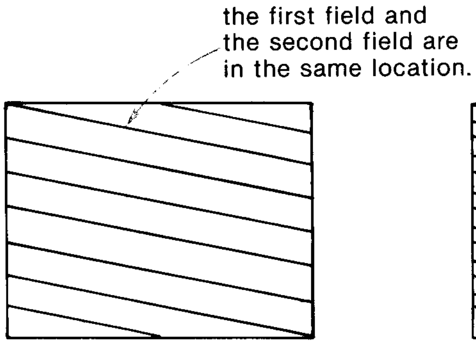
If a printer other than an MSX-type printer is used, and SCREEN ,,,,1 is executed, the special MSX graphic characters will be printed as blank spaces.

THE INTERLACE MODE

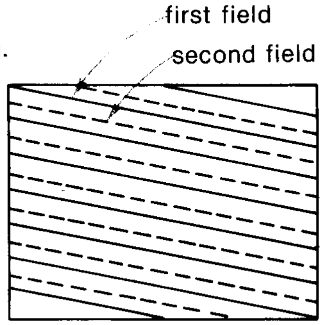
Interlace Scanning

MSX computers normally perform non-interlace scanning. Scanning can be changed to interlace scanning by the sixth parameter (interlace mode) of the SCREEN statement.

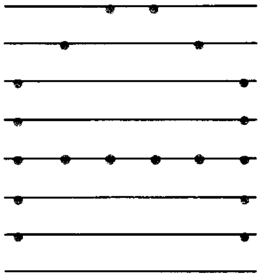
When interlace scanning is used, the locations of the first field scan and the second field scan are different. This provides a more detailed display. A monitor TV with long afterglow properties is necessary if interlace scanning is used. Flicker will result if a normal TV or monitor is used.



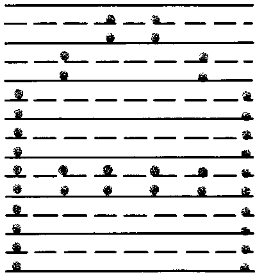
Non-Interlace Scanning



Interlace Scanning



Non-Interlace Scanning



Interlace Scanning

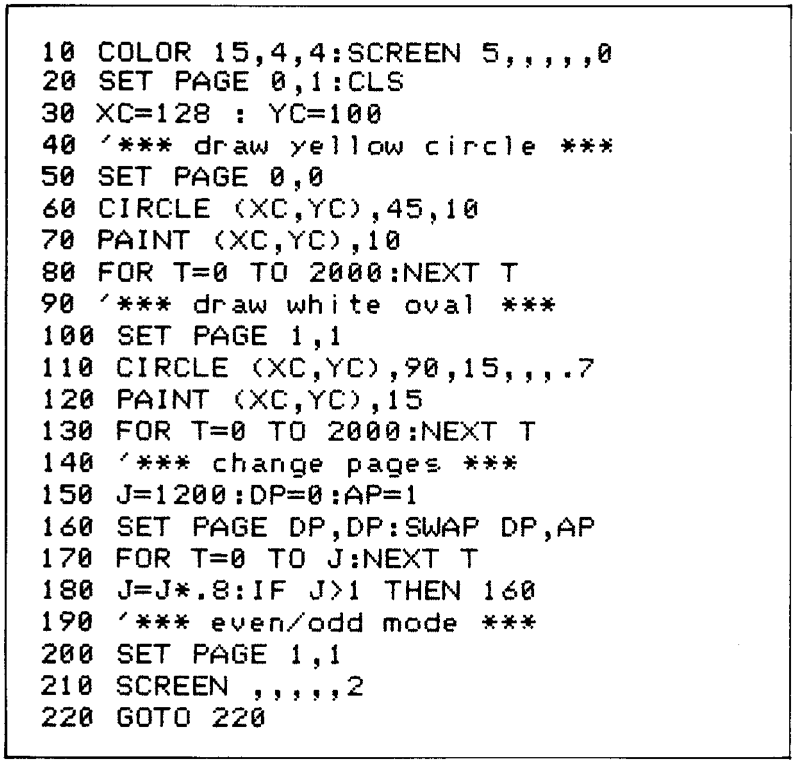
The Even/Odd Alternating Pages Display Mode

The SCREEN statement interlace parameter can be used to select the even/odd alternating pages display mode. To select this mode, first the display page must be set as an odd page (1 or 3) with the PAGE statement. Then when the even/odd alternating pages display mode is specified with the SCREEN statement, the display page and the page numbered one number less than the display page will be displayed alternately at a rapid speed.

The following table shows the interlace mode settings.

Interlace mode	Specified mode
0	normal (non-interlace, no alternating pages)
1	interlace mode
2	even/odd alternating pages display mode
3	even/odd alternating pages, interlace mode

The following program draws a yellow circle on page 0 and a white oval on page 1 in the SCREEN 5 mode. First page 0 and page 1 are displayed alternately by changing the display page with the SET PAGE statement, and the interval between page changes is progressively shortened. Then an interesting effect is achieved by changing to the even/odd alternating pages display mode at the end of the program.



```
10 COLOR 15,4,4:SCREEN 5,,,,,0
20 SET PAGE 0,1:CLS
30 XC=128 : YC=100
40 '*** draw yellow circle ***
50 SET PAGE 0,0
60 CIRCLE (XC,YC),45,10
70 PAINT (XC,YC),10
80 FOR T=0 TO 2000:NEXT T
90 '*** draw white oval ***
100 SET PAGE 1,1
110 CIRCLE (XC,YC),90,15,,,.7
120 PAINT (XC,YC),15
130 FOR T=0 TO 2000:NEXT T
140 '*** change pages ***
150 J=1200:DP=0:AP=1
160 SET PAGE DP,DP:SWAP DP,AP
170 FOR T=0 TO J:NEXT T
180 J=J*.8:IF J>1 THEN 160
190 '*** even/odd mode ***
200 SET PAGE 1,1
210 SCREEN ,,,,,2
220 GOTO 220
```

Chapter 6 Sprite Patterns

SPRITE PATTERN DEFINITION AND USE

- ●Sprite Patterns
- ●Sprite Pattern Definition—SPRITE$
- ●Displaying a Sprite Pattern—PUT SPRITE
- ●Animating Sprite Patterns

SPRITE PATTERNS

A sprite is a freely defined pattern composed of 8×8 dots or 16×16 dots which can be moved about on the screen. In MSX2-BASIC, sprite patterns can be displayed and moved about on 32 different sprite planes.

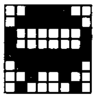
The Screen Mode and Sprite Patterns

Sprite patterns can be used in SCREEN 1 through 8, or all modes except SCREEN 0. A sprite pattern used in the SCREEN 4 through 8 modes has several functions which are not available in the SCREEN 1 through 3 modes.

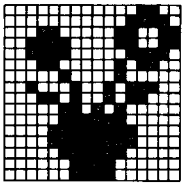
Type of Sprite Patterns

One sprite pattern is made up of either 8×8 dots or 16×16 dots. Each can be displayed in either a standard size or a magnified size. The magnified size is twice the size, both vertically and horizontally, of the standard size.

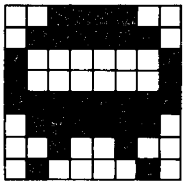
8×8 dot standard



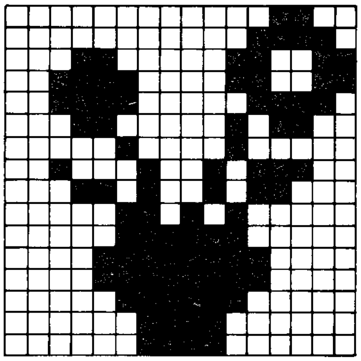
16×16 dot standard



8×8 dot magnified



16×16 dot magnified



Specifying Sprite Size—SCREEN Statement

The second parameter of the SCREEN statement selects the sprite pattern size.

SCREEN Mode, Sprite Size

Sprite size	Size selected
0	8×8 dot, standard
1	8×8 dot, magnified
2	16×16 dot, standard
3	16×16 dot, magnified

For example,

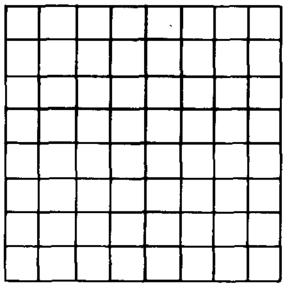
```
SCREEN 2,3
```

would specify the SCREEN 2 mode and select the 16×16 dot magnified sprite size. Once the sprite size is selected with the SCREEN statement, the sprites in all sprite planes will be displayed in that size.

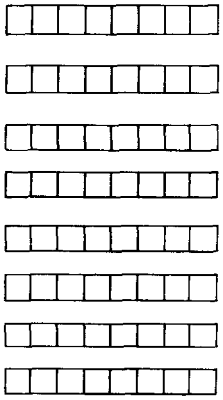
SPRITE PATTERN DEFINITION SPRITE$ variable

The 8×8 Dot Sprite Pattern

To define an 8×8 dot pattern, the pattern is first separated into 8 horizontal lines. For instance, an arrow pattern is defined as shown in the following figure.



Each of the 8 horizontal lines is divided into a small pattern of 8 dots.



Next a 1 is assigned to a marked dot, and a 0 to an unmarked dot to produce a binary number. For example, the top line would be 00011000, and the second line would be 00111100.

□□□■■□□□ ➡ 0 0 0 1 1 0 0 0

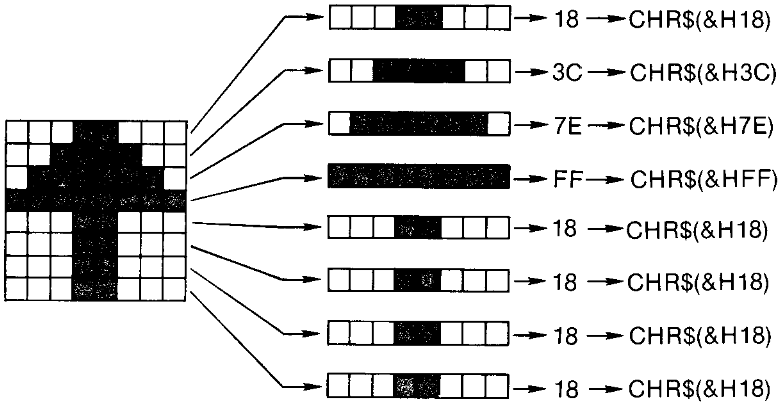
□□■■■■□□ ➡ 0 0 1 1 1 1 0 0

The binary numbers are then converted to hexadecimal or decimal. The top line becomes: 00011000 (binary) = 18 (hexadecimal) or 24 (decimal). The second line becomes 00111100 (binary) = 3C (hexadecimal) or 60 (decimal).

The following table is used to convert binary numbers to hexadecimal.

Pattern	Hexadecimal	Pattern	Hexadecimal
□□□□	0	■□□□	8
□□□■	1	■□□■	9
□□■□	2	■□■□	A
□□■■	3	■□■■	B
□■□□	4	■■□□	C
□■□■	5	■■□■	D
□■■□	6	■■■□	E
□■■■	7	■■■■	F

First the eight dot pattern is divided into 4 dots on the left side of the pattern and the 4 dots on the right side. Then the above table is used to determine the hexadecimal equivalents.
For the □□□■■□□□ pattern, the left 4 dots are □□□■, which would be hexadecimal 1 in the above table, and the right 4 dots are ■□□□, which is 8. Therefore the hexadecimal equivalent would be 18.
The final step is to obtain the character for which the hexadecimal (or decimal) is the character code using the CHR$ function, as shown below.



The character data obtained by the 8×8 dot sprite pattern is added sequentially from the top, and is assigned to the SPRITE$ variable to define the sprite pattern. The arrow pattern in the above example would be defined as follows:

```
SPRITE$(1)=CHR$(&H18)+CHR$(&H3C)+CHR$(&H
7E)+CHR$(&HFF)+CHR$(&H18)+CHR$(&H18)+CHR
$(&H18)+CHR$(&H18)
```

The number of the defined sprite pattern is 1, as indicated by the number 1 inside the parentheses of SPRITE$ (1).
(The use of functions is explained in Chapter 7).

SPRITE$ (sprite number) = character string

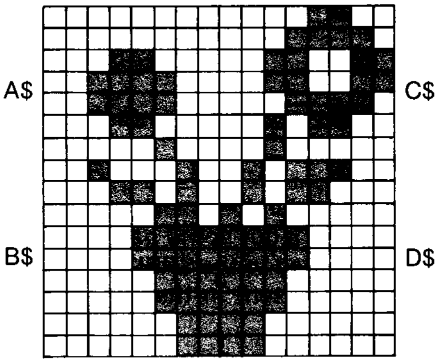
Also, if there is a character that can be obtained with the CHR$ function, it can be used directly in the character string. In the above example, since CHR$ (&H3C) is "<", and CHR$ (&H7E) is "~", these two characters can be used directly as follows:

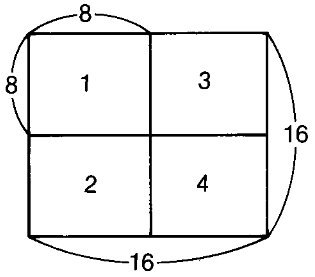
```
SPRITE$(1)=CHR$(&H18)+"<"+"~"+CHR$(&HFF)
+CHR$(&H18)+CHR$(&H18)+CHR$(&H18)+CHR$(&
H18)
```

(For the conversion to characters, refer to the character code table in the BASIC Programming Reference Manual)

The 16×16 Dot Sprite Pattern

The 16×16 dot sprite pattern is defined in a similar way. However, a 16×16 dot sprite pattern is considered to be made up of four separate 8×8 dot sprite patterns. These four patterns are defined in the following sequence.





```
A$=CHR$(&H0)+CHR$(&H0)+CHR$(&H18)+CHR$(&
H3C)+CHR$(&H3C)+CHR$(&H18)+CHR$(&H4)+CHR
$(&H22)
B$=CHR$(&H1A)+CHR$(&H6)+CHR$(&HF)+CHR$(&
HF)+CHR$(&H7)+CHR$(&H7)+CHR$(&H3)+CHR$(&
H3)
C$=CHR$(&HC)+CHR$(&H1E)+CHR$(&H33)+CHR$(
33)+CHR$(&H1E)+CHR$(&H2C)+CHR$(&H20)+CHR
$(&H5C)
D$=CHR$(&H58)+CHR$(&HA0)+CHR$(&HF0)+CHR$
(&HF0)+CHR$(&HE0)+CHR$(&HE0)+CHR$(&HC0)+
CHR$(&HC0)
SPRITE$(2)=A$+B$+C$+D$
```

The Number of Sprite Patterns That Can Be Defined

A maximum of 256 8×8 dot sprite patterns can be defined, using the numbers from 0 to 255, and a maximum of 64 16×16 dot sprite patterns can be defined, using the numbers from 0 to 63.

SPRITE PATTERN DISPLAY PUT SPRITE

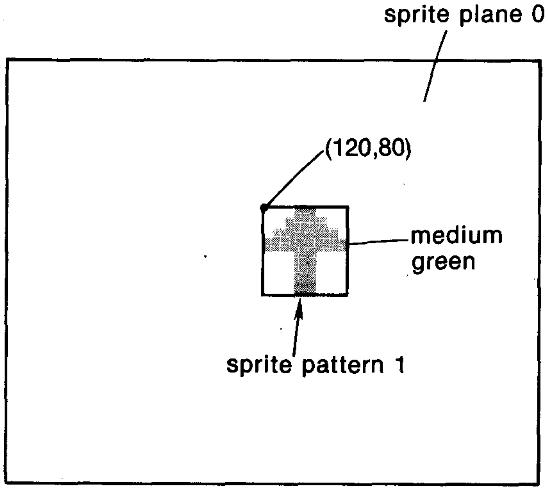
The PUT SPRITE statement is used to display a defined sprite pattern on a sprite plane.

PUT SPRITE sprite plane number, [(X,Y)], [color], [sprite pattern number]

The PUT SPRITE statement displays the sprite pattern with the specified number on the specified sprite plane in the specified color.
To display the sprite pattern defined above (pattern no. 1) on sprite plane 0 in medium green (color code 2) at the (120,80) location, you would execute:

```
        PUT SPRITE 0,(120,80),2,1
```

The specified display location corresponds to the dot at the top left of the sprite pattern frame. The X, Y coordinates are specified in a coordinate system on the graphic screen which takes (0, – 1) as the origin (0,0).



Sprite Pattern Display Rules (for SCREEN 1, 2, 3 modes)

- Only one sprite pattern can be displayed on one sprite plane.
- When sprite patterns overlap on different sprite planes, the sprite pattern on the sprite plane at the back (the large numbered plane) is hidden by the sprite pattern in front.
- When 5 or more sprite patterns are lined up on the same horizontal line, only the 4 sprite patterns with the higher priority (the ones on the sprite planes with the lowest numbers) will be displayed.
- When the display location is omitted, the location specified by the last graphic command will be taken as the specified location.
- When the color code is omitted, the foreground color is taken as the specified color.
- When the sprite pattern number is omitted, the sprite plane number is taken as the specified number.

ANIMATING SPRITE PATTERNS

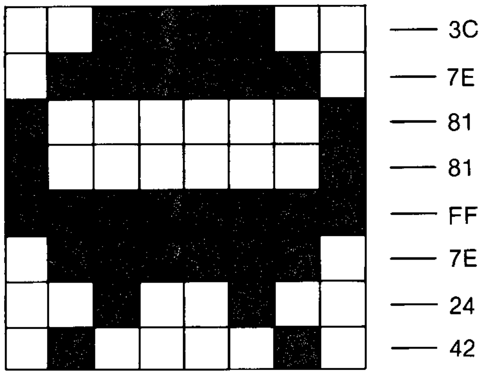
A sprite pattern is animated by repeatedly executing the PUT SPRITE statement while changing the display location specified in the statement. Each time the PUT SPRITE statement is executed, the previous sprite in the same sprite plane disappears, so it is not necessary to erase the previous sprite each time in a program. Also, the display location is changed in units of 1 dot, so the movement of the sprite pattern is smooth.

The following program moves a UFO-shaped pattern diagonally around the screen.

```
10 SCREEN 2,0
20 SPRITE$(0)=CHR$(&H3C)+CHR$(&H7E)+CHR$
(&H81)+CHR$(&H81)+CHR$(&HFF)+CHR$(&H7E)+
CHR$(&H24)+CHR$(&H42)
30 COLOR ,1,1:CLS
40 X=120:Y=50:VX=1:VY=1
50 PUT SPRITE 0,(X,Y),5,0
60 X=X+VX
70 IF X>240 OR X<0 THEN VX=-VX
80 Y=Y+VY
90 IF Y>180 OR Y<0 THEN VY=-VY
100 GOTO 50
```

The SCREEN mode is 2, and the sprite size is 8×8 dot standard (line 10).

The following illustration shows the sprite pattern defined in line 20.

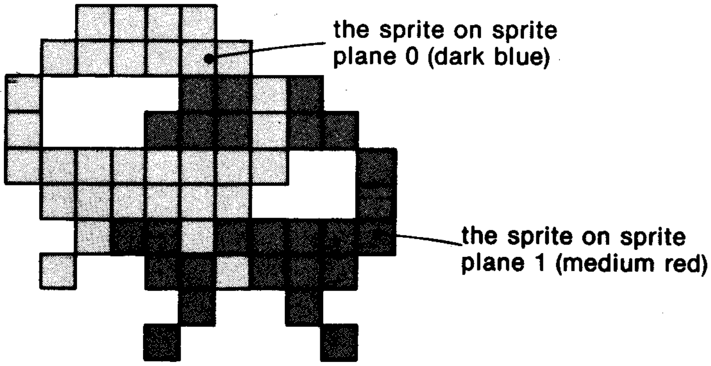


This sprite pattern is displayed by the PUT SPRITE statement in line 50. The initial values of (X,Y), which show the display location, are set at (120,50). Then the (X,Y) values are changed in lines 60 to 80, and the program returns to line 50. This causes the sprite pattern to move on the screen.

The following program shows what happens when two sprite patterns overlap.

```
10 SCREEN 2,1
20 SPRITE$(0)=CHR$(&H3C)+CHR$(&H7E)+CHR$
(&H81)+CHR$(&H81)+CHR$(&HFF)+CHR$(&H7E)+
CHR$(&H24)+CHR$(&H42)
30 COLOR ,1,1:CLS
40 FOR X=0 TO 117
50 PUT SPRITE 0,(X,80),4,0
60 PUT SPRITE 1,(240-X,84),8,0
70 NEXT X
80 GOTO 80
```

The 8×8 dot magnified sprite size is used so that the overlap can be clearly seen. The sprite pattern is the same as the pattern used in the previous program. This sprite pattern is displayed on sprite planes 0 and 1 by the two PUT SPRITE statements in lines 50 and 60. The pattern on sprite plane 0 is dark blue and moves from left to right. The pattern on sprite plane 1 is medium red and moves from right to left. When the two sprite patterns overlap, the sprite on sprite plane 0 (dark blue) appears on top of the other sprite.


the sprite on sprite plane 0 (dark blue)
the sprite on sprite plane 1 (medium red)

USING ENHANCED SPRITE FUNCTIONS

- The Enhanced Sprite Functions
- Changing the Color of a Sprite—COLOR SPRITE
- Specifying the Color for Each Line of a Sprite—COLOR SPRITE$
- Sprite Definition Technique

THE ENHANCED SPRITE FUNCTIONS

Sprites with enhanced functions can be used in the SCREEN 4 through SCREEN 8 modes.
The following table shows the enhancement of the sprite functions in the SCREEN 4 through 8 modes, in comparison to the SCREEN 1 through 3 modes.

Function	SCREEN 1—3	SCREEN 4—8
Number of sprites displayed on one horizontal line	A maximum of 4	A maximum of 8
Sprite color	1 color for 1 sprite	A maximum of 8 (8×8 dot) colors, or 16 (16×16 dot) colors for 1 sprite
To change the sprite color	Specified with the PUT SPRITE statement	Specified with the PUT SPRITE statement or the COLOR SPRITE statement

CHANGING THE COLOR OF A SPRITE
COLOR SPRITE

The color of a sprite is specified with the PUT SPRITE statement, but in the SCREEN 4 to 8 modes the color of a sprite after it has been displayed can be changed with the COLOR SPRITE statement.

COLOR SPRITE (sprite plane number) = palette number

The COLOR SPRITE statement changes the color of the sprite pattern in a specified plane to the specified color.

For example, to change the color of the sprite pattern displayed in sprite plane 2 to medium red (color code 8), you would execute

```
COLOR SPRITE(2)=8
```

When the following program is executed, 6 UFOs of different colors land, and then their colors are changed. The last color they become is transparent, and then the landing routine is repeated.

```
10 SCREEN 5,1
20 SPRITE$(0)=CHR$(&H3C)+CHR$(&H7E)+CHR$
(&H81)+CHR$(&H81)+CHR$(&HFF)+CHR$(&H7E)+
CHR$(&H24)+CHR$(&H42)
30 COLOR ,1,1:CLS
40 X1=0:Y1=118
50 FOR L=1 TO 14
60 READ X2,Y2
70 LINE (X1,Y1)-(X2,Y2),3
80 X1=X2:Y1=Y2
90 NEXT L
100 PAINT (1,119),3
110 P=0:X=5:YE=101:C=3:GOSUB 250
120 P=1:X=45:YE=122:C=4:GOSUB 250
130 P=2:X=83:YE=87:C=5:GOSUB 250
140 P=3:X=133:YE=143:C=6:GOSUB 250
150 P=4:X=168:YE=127:C=7:GOSUB 250
160 P=5:X=218:YE=96:C=8:GOSUB 250
170 FOR S=1 TO 10
180 FOR SP=0 TO 5
190 SC=S+3:IF S=10 THEN SC=0
200 COLOR SPRITE (SP)=SC
210 NEXT SP
220 FOR T=0 TO 300:NEXT T
230 NEXT S
240 GOTO 110
250 FOR Y=-8 TO YE
260 PUT SPRITE P,(X,Y),C,0
270 NEXT Y
280 RETURN
290 DATA 26,118,40,139,67,139
300 DATA 79,104,103,104,128,160
310 DATA 154,160,161,144,192,144
320 DATA 200,113,252,113,252,212
330 DATA 0,212,0,118
```

draws the background

moves the sprites

changes the color of the sprites

sprite movement subroutine

background drawing data

The COLOR SPRITE statement in line 200 changes the color of the sprites. The sprite plane numbers are assigned to the variable SP. The color code (variable SC) changes from 3 to 13, but with the final repetition of the loop (S = 10), the color code becomes 0 (transparent).

SPECIFYING THE COLOR FOR EACH LINE OF A SPRITE COLOR SPRITE$

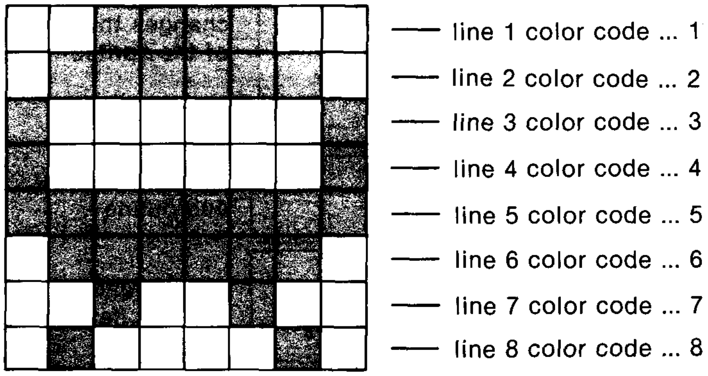
An 8×8 dot sprite is composed of eight horizontal lines, while a 16×16 dot sprite has sixteen horizontal lines. In SCREEN modes 4 to 8, it is possible to specify the color of each of these lines separately.

COLOR SPRITE$ (sprite plane number) = character string

The COLOR SPRITE$ statement uses a character string to specify the color of each line of a sprite pattern on the specified sprite plane. The character string in the COLOR SPRITE$ statement is composed of

CHR$ (color code)

character strings connected by the + sign. The first CHR$ (color code) specifies the color for the first line of the sprite, the second CHR$ (color code) the color of the second line, and so on.



The following COLOR SPRITE$ statement will specify the colors shown in the above illustration for a sprite pattern on sprite plane 0.

```
COLOR SPRITE$(0)=CHR$(1)+CHR$(2)+CHR$(3)
+CHR$(4)+CHR$(5)+CHR$(6)+CHR$(7)+CHR$(8)
```

If only seven or less CHR$ (color code) character strings are specified, the color of the lines not specified will not be changed.
In the statement

```
COLOR SPRITE(0)=CHR$(1)+CHR$(2)
```

the color of line 1 will become color code 1 and the color of line 2 will become color code 2, but the color of lines 3 and below will not be changed.

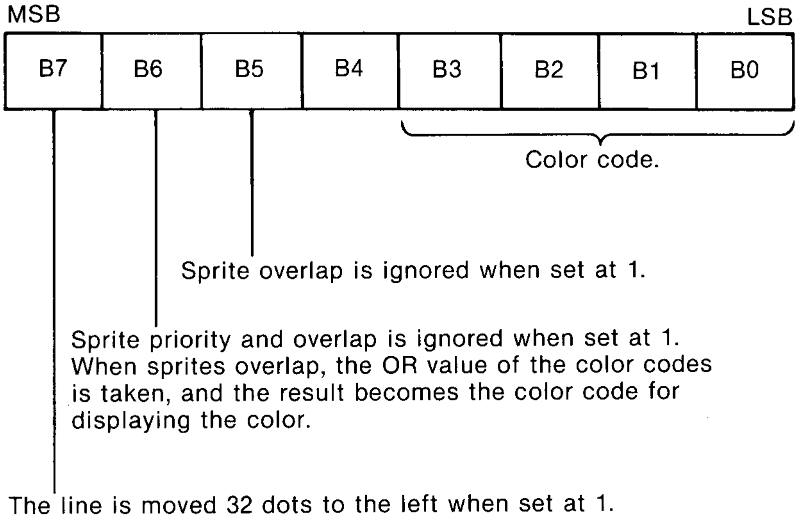
The following program is a revision of the previous program to display UFOs which have a medium red and dark blue striped pattern.

```
10 SCREEN 5,1
20 SPRITE$(0)=CHR$(&H3C)+CHR$(&H7E)+CHR$
(&H81)+CHR$(&H81)+CHR$(&HFF)+CHR$(&H7E)+
CHR$(&H24)+CHR$(&H42)
30 COLOR ,1,1:CLS
40 X1=0:Y1=118
50 FOR L=1 TO 14
60 READ X2,Y2
70 LINE (X1,Y1)-(X2,Y2),3
80 X1=X2:Y1=Y2
90 NEXT L
100 PAINT (1,119),3
110 P=0:X=5:YE=101:C=3:GOSUB 250
120 P=1:X=45:YE=122:C=4:GOSUB 250
130 P=2:X=83:YE=87:C=5:GOSUB 250
140 P=3:X=133:YE=143:C=6:GOSUB 250
150 P=4:X=168:YE=127:C=7:GOSUB 250
160 P=5:X=218:YE=96:C=8:GOSUB 250
170 FOR SP=0 TO 5
180 COLOR SPRITE$(SP)=CHR$(8)+CHR$(4)+CH
R$(8)+CHR$(4)+CHR$(8)+CHR$(4)+CHR$(8)+CH
R$(4)
190 NEXT SP
200 GOTO 200
250 FOR Y=-8 TO YE
260 PUT SPRITE P,(X,Y),C,0
270 NEXT Y
280 RETURN
290 DATA 26,118,40,139,67,139
300 DATA 79,104,103,104,128,160
310 DATA 154,160,161,144,192,144
320 DATA 200,113,252,113,252,212
330 DATA 0,212,0,118
```

changed

Line 180 in the part of the program that was changed specifies the red and blue striped pattern for the sprite.

Character String Data
Data that can be used with the CHR$ function in the COLOR SPRITE$ statement is not limited only to color codes. When the data is displayed in binary numbers, each bit has the following function:



Sprite overlap is ignored when set at 1.

Sprite priority and overlap is ignored when set at 1. When sprites overlap, the OR value of the color codes is taken, and the result becomes the color code for displaying the color.

The line is moved 32 dots to the left when set at 1.

Sprite overlap is explained in Chapter 8.

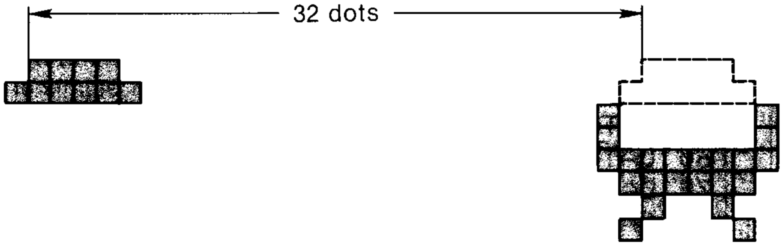
When B7 is set at 1, and the color code (B3—B0) is set at 0010 (= 2, medium green), the data becomes 10000010. Since 10000010 is &H82 in hexadecimal, when

```
COLOR SPRITE$(0)=CHR$(&H82)
```

is executed, the first line of the sprite pattern on sprite plane 0 becomes medium green and the line is moved 32 dots to the left. When the following program is executed, the hatch of the UFO opens, and a spaceman appears.

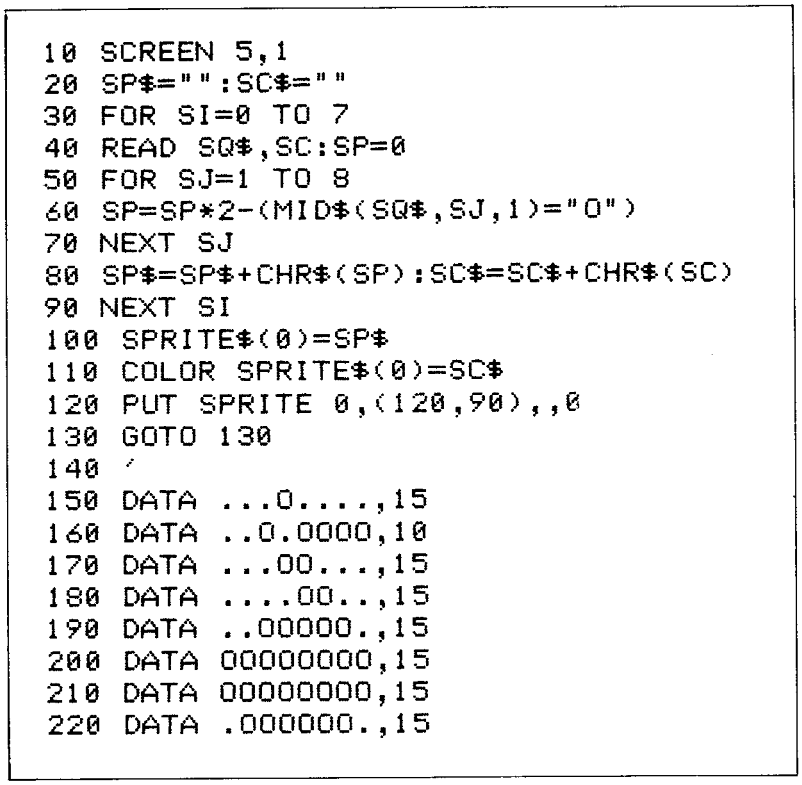
```
10 SCREEN 5,1
20 SPRITE$(0)=CHR$(&H3C)+CHR$(&H7E)+CHR$
(&H81)+CHR$(&H81)+CHR$(&HFF)+CHR$(&H7E)+
CHR$(&H24)+CHR$(&H42)
30 SPRITE$(1)=CHR$(&H58)+CHR$(&H58)+CHR$
(&H7E)+CHR$(&H1A)+CHR$(&H18)+CHR$(&H18)+
CHR$(&H0)+CHR$(&H0)
40 CLS
50 PUT SPRITE 0,(120,100),1,0
60 FOR T=0 TO 1000:NEXT T
70 COLOR SPRITE$(0)=CHR$(&H81)+CHR$(&H81
)
80 FOR T=0 TO 1000:NEXT T
90 PUT SPRITE 1,(120,96),14,1
100 GOTO 100
```

Line 70 moves the first and second lines of the sprite pattern (pattern no. 0) displayed on sprite plane 0 32 dots to the left.



SPRITE DEFINITION TECHNIQUE

The basic method of defining a sprite pattern is to use the CHR$ function to assign the pattern data to the SPRITE$ variable. In the following program the sprite pattern data is written in DATA statements and assigned to the SPRITE$ variable by the READ statement. The sprite pattern data is written in the DATA statements using periods ( ) and the capital letter O in order to give a visual presentation of the shape of the pattern.



```
10 SCREEN 5,1
20 SP$="":SC$=""
30 FOR SI=0 TO 7
40 READ SQ$,SC:SP=0
50 FOR SJ=1 TO 8
60 SP=SP*2-(MID$(SQ$,SJ,1)="O")
70 NEXT SJ
80 SP$=SP$+CHR$(SP):SC$=SC$+CHR$(SC)
90 NEXT SI
100 SPRITE$(0)=SP$
110 COLOR SPRITE$(0)=SC$
120 PUT SPRITE 0,(120,90),,0
130 GOTO 130
140 '
150 DATA ...O....,15
160 DATA ..O.OOOO,10
170 DATA ...OO...,15
180 DATA ....OO..,15
190 DATA ..OOOOO.,15
200 DATA OOOOOOOO,15
210 DATA OOOOOOOO,15
220 DATA .OOOOOO.,15
```

In the DATA statement in lines 150 to 220, the sprite pattern to be defined is expressed with periods and O. A period indicates an unmarked dot, and an O indicates a marked dot. The number at the end of each DATA statement is the color code which specifies the color of that line of the sprite pattern. Lines 20 through 110 assign the data in the DATA statements to the SPRITE$ variable and specify the colors with the COLOR SPRITE$ statement. In line 60 a function is used (explained in Chapter 7) to change the data to hexadecimal. Then lines 100 and 110 define the pattern and the colors. The following pattern is defined.

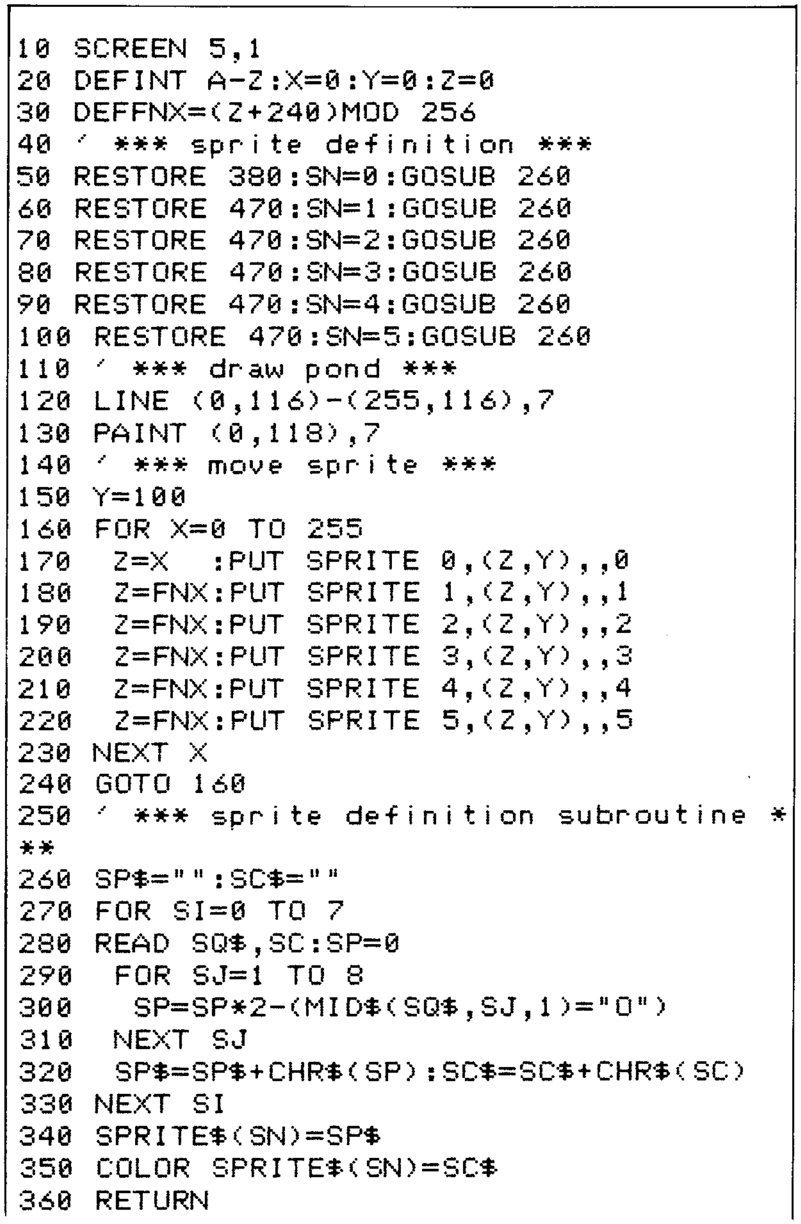
			▨					— white (15)
		▨		▨	▨	▨	▨	— dark yellow (10)
			▨	▨				— white (15)
				▨	▨			— white (15)
		▨	▨	▨	▨	▨		— white (15)
▨	▨	▨	▨	▨	▨	▨	▨	— white (15)
▨	▨	▨	▨	▨	▨	▨	▨	— white (15)
	▨	▨	▨	▨	▨	▨		— white (15)

The method used to define the sprite pattern in this program might seem to involve a somewhat advanced programming technique, but it is simple if you think of it as a formula for defining the pattern. Using this method makes the program a little longer, but it has the advantage of giving a visual representation of the sprite in the program list so that you can check it, and you do not have to convert each piece of data to hexadecimal when you are writing the program. Also, it is very easy to revise the sprite pattern simply by changing a period into an O or an O into a period.

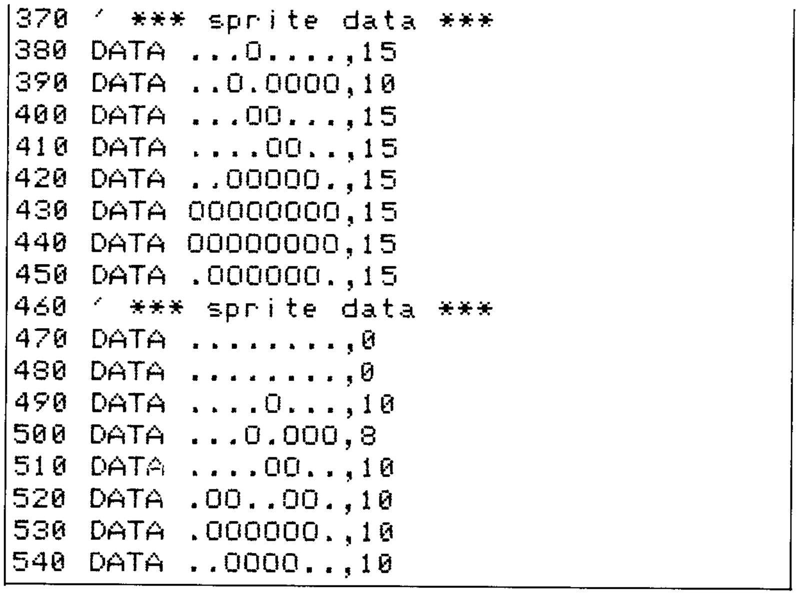
Sample Program

The following program uses the DATA statement technique to define sprites. This program uses two statements (DEFINT and DEFFN) which are not explained in this book. For an explanation of these statements you can refer to the Programming Reference Manual.

The program displays a mother duck and her baby ducks swimming on a pond.



```
10 SCREEN 5,1
20 DEFINT A-Z:X=0:Y=0:Z=0
30 DEFFNX=(Z+240)MOD 256
40 ' *** sprite definition ***
50 RESTORE 380:SN=0:GOSUB 260
60 RESTORE 470:SN=1:GOSUB 260
70 RESTORE 470:SN=2:GOSUB 260
80 RESTORE 470:SN=3:GOSUB 260
90 RESTORE 470:SN=4:GOSUB 260
100 RESTORE 470:SN=5:GOSUB 260
110 ' *** draw pond ***
120 LINE (0,116)-(255,116),7
130 PAINT (0,118),7
140 ' *** move sprite ***
150 Y=100
160 FOR X=0 TO 255
170  Z=X   :PUT SPRITE 0,(Z,Y),,0
180  Z=FNX:PUT SPRITE 1,(Z,Y),,1
190  Z=FNX:PUT SPRITE 2,(Z,Y),,2
200  Z=FNX:PUT SPRITE 3,(Z,Y),,3
210  Z=FNX:PUT SPRITE 4,(Z,Y),,4
220  Z=FNX:PUT SPRITE 5,(Z,Y),,5
230 NEXT X
240 GOTO 160
250 ' *** sprite definition subroutine *
**
260 SP$="":SC$=""
270 FOR SI=0 TO 7
280 READ SQ$,SC:SP=0
290  FOR SJ=1 TO 8
300   SP=SP*2-(MID$(SQ$,SJ,1)="O")
310  NEXT SJ
320  SP$=SP$+CHR$(SP):SC$=SC$+CHR$(SC)
330 NEXT SI
340 SPRITE$(SN)=SP$
350 COLOR SPRITE$(SN)=SC$
360 RETURN
```



```
370 ' *** sprite data ***
380 DATA ...O....,15
390 DATA ..O.OOOO,10
400 DATA ...OO...,15
410 DATA ....OO..,15
420 DATA ..OOOOO.,15
430 DATA OOOOOOOO,15
440 DATA OOOOOOOO,15
450 DATA .OOOOOO.,15
460 ' *** sprite data ***
470 DATA ........,0
480 DATA ........,0
490 DATA ....O...,10
500 DATA ...O.OOO,8
510 DATA ....OO..,10
520 DATA .OO..OO.,10
530 DATA .OOOOOO.,10
540 DATA ..OOOO..,10
```

Chapter 7
Using Functions

NUMERIC TYPE FUNCTIONS

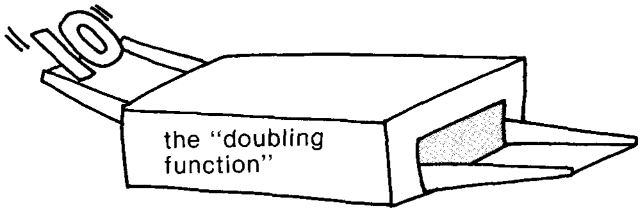
- What are Functions?
- Various Numeric Type Functions
- The Random Function

WHAT ARE FUNCTIONS?

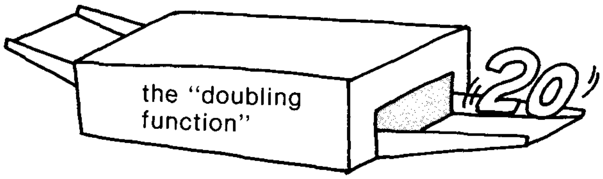
The BASIC language is divided into commands and functions. Up to now this book has focused primarily on explaining the use of commands. We will now concentrate on functions.

A BASIC function can be thought of as a box that processes a value and gives the result.

For example, suppose we had a function which doubled a number. (Actually, such a function is not included in MSX2-BASIC). If the number 10 were put into this function, it would come out as 20. If 100 were input, it would become 200. 450 would become 900 and so forth. Any number put into the function would be doubled.



If 10 is put in



20 comes out

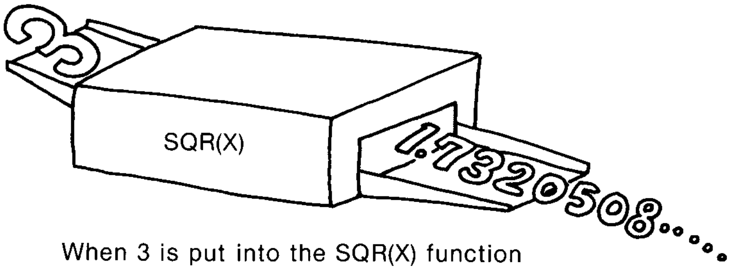
NUMERIC TYPE FUNCTIONS

In MSX2-BASIC, functions for which the input is a number and the output is a number are called numeric type functions. There are a total of 17 numeric type functions in MSX2-BASIC.

ABS(X) CDBL(X) CINT(X) CSNG(X)
FIX(X) INT(X) SQR(X) SGN(X)
ATN(X) COS(X) SIN(X) TAN(X)
LOG(X) EXP(X) RND(X)
ERR ERL

THE SQUARE ROOT FUNCTION SQR(X)

Let's use the SQR(X) function as an example to see how functions work. This function gives the square root of the number X. In talking about functions, instead of using the word "gives," the word "returns" is normally used. Rephrased in this way, we can say the SQR(X) function returns the square root of X.
For instance, if X is 3, then the square root is 1.7320508... . If X is 5 then the square root would be 2.2360679... .



When 3 is put into the SQR(X) function

How to use Functions
When 3 is put into the SQR(X) function, 1.7320508... is returned. When a value is entered into a function, it is entered in place of X.

```
SQR(3)
```

Now 3 has been entered into the SQR(X) function. You can also assign 3 to a variable, and enter the variable into the function instead of the number.
If you first execute

```
A=3
```

with

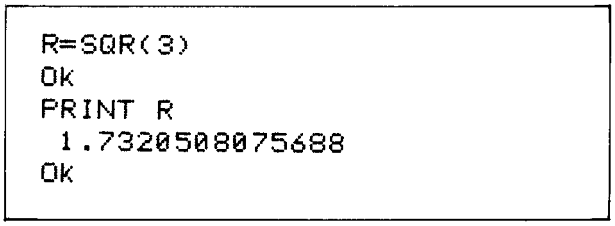
```
SQR(A)
```

the number 3 will be entered in the SQR(X) function.
In either case, whether the number is entered directly or a variable is used, the SQR(X) function now has the value 1.7320508... . Any time you call this function in the program, it will return the value 1.7320508... .

The procedure for returning the value of a function in a program is as follows. Since functions themselves are not commands, they cannot be used by themselves to control the computer. They must be combined with some command, in order to have the value returned. The LET statement and the PRINT statement can be used for this purpose.

```
R=SQR(3)
```

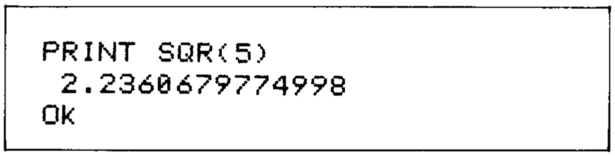
This LET statement assigns the value of SQR(3), that is, 1.7320508... to the numeric variable R. Let's check to make sure this happens.



```
R=SQR(3)
Ok
PRINT R
 1.7320508075688
Ok
```

The value of the square root of 3, to 13 decimal places (a total of 14 digits), has been assigned to the variable R.
In MSX2-BASIC all calculation results normally have a precision of 14 digits. But 6 digit precision can also be used. 14 digit precision is called double precision. 6 digit precision is called single precision.

The value of a function can be directly displayed using only the PRINT statement.



```
PRINT SQR(5)
 2.2360679774998
Ok
```

Functions can also be used in an IF—THEN conditional expression.

```
IF SQR(A)>=10 THEN END
```

The value of SQR(A) changes according to the value of A. In this IF—THEN statement, if the value of SQR(A) becomes 10 or greater, the program is ended.
As shown above, a function processes an entered value following fixed rules according to what the function is designed to do, and returns the result. The returned value is used in combination with such BASIC commands as the LET statement, the PRINT statement, and the IF-THEN statement.

TRIGONOMETRIC FUNCTIONS SIN(X)

The trigonometric function SIN(X) is one of the numeric functions. This function returns the sine of X. Other trigonometric functions used in MSX2-BASIC are COS(X) (cosine), TAN(X) (tangent), and ATN(X) (arc tangent).

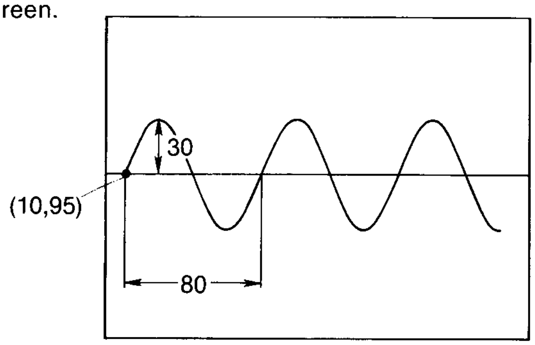
Let's use the SIN(X) function to make a program which draws a sine curve.

```
10 SCREEN 2:CLS
20 PI=3.14
30 LINE (0,95)-(252,95)
40 FOR X=10 TO 230
50 R=(X-10)*PI/40
60 S=SIN(R)
70 VY=S*30
80 Y=95-VY
90 PSET (X,Y)
100 NEXT X
110 GOTO 110
```

When trigonometric functions such as SIN(X), COS(X), and TAN(X) are used, the value of X is expressed in radian units (1 radian = π). In this program, X has values from 10 to 230. These values are converted to radian units in line 50. When the value of X is 90 it is 2 radians (2π), when it is 50 it is 1 radian, and when it is 10 it is 0 radian. (X is the X-coordinate on the screen when the sine curve is drawn.)

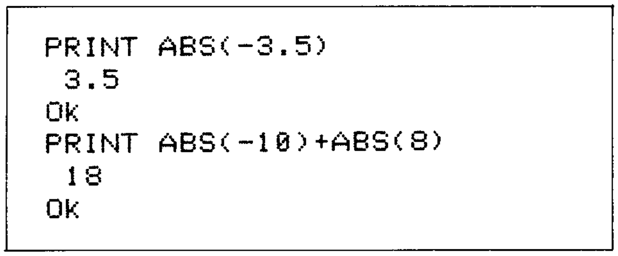
The sines of these radian values are assigned to S by the SIN(X) function in line 60. When the sine value is 1, the distance on the Y-axis is 30 (line 70). In line 80 the sine curve Y-coordinate is determined taking 95 as the Y-coordinate reference point.

When this program is executed, the following sine curve is drawn on the screen.



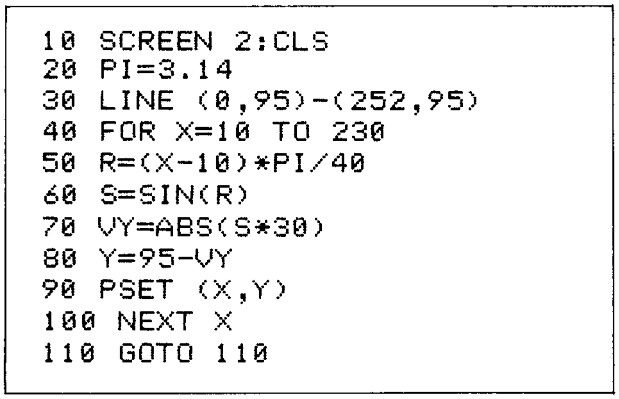
THE ABSOLUTE VALUE FUNCTION ABS(X)

The ABS(X) function returns the absolute value of X.
For example, when X is 100, ABS(X) returns the value of 100. When X is −100, ABS(X) also returns the value of 100.



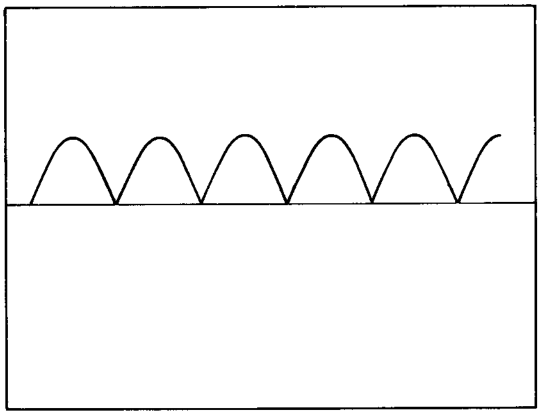
```
PRINT ABS(-3.5)
 3.5
Ok
PRINT ABS(-10)+ABS(8)
 18
Ok
```

Let's use the ABS(X) function in the previous sine curve program.



```
10 SCREEN 2:CLS
20 PI=3.14
30 LINE (0,95)-(252,95)
40 FOR X=10 TO 230
50 R=(X-10)*PI/40
60 S=SIN(R)
70 VY=ABS(S*30)
80 Y=95-VY
90 PSET (X,Y)
100 NEXT X
110 GOTO 110
```

The program has been revised so that the absolute value of SIN(R) * 30 is assigned to VY in line 70. As a result, the following figure is drawn on the screen.

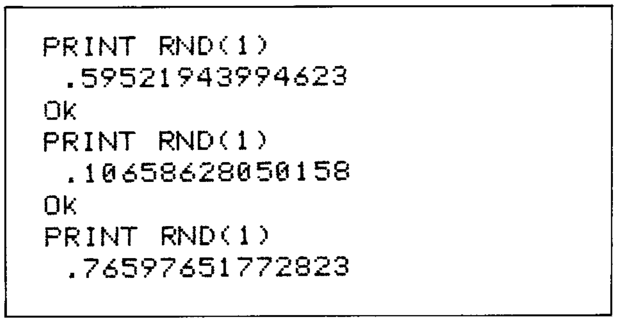


THE RANDOM FUNCTION RND(X)

The RND(X) function **returns a random number from 0 to less than 1** when X is a number greater than 0.
Execute

```
        PRINT RND(1)
```

several times in the direct mode.



```
PRINT RND(1)
 .59521943994623
Ok
PRINT RND(1)
 .10658628050158
Ok
PRINT RND(1)
 .76597651772823
```

Each time the RND(X) function is executed it returns a random number from 0 to less than 1. The RND(X) function can be used to produce random numbers within a certain range.

RND(X) AND THE INTEGER FUNCTION INT(X)

Let's consider how to use the RND(X) function to return random integers from 1 to 200. RND(1) itself will return only values from 0 to less than 1 (from 0 to 0.99999999999999). If we multiply RND(1) by 200

```
RND(1)*200
```

the value will be a number from 0 to less than 200. But the value will be a 14 digit number, including the digits after the decimal point. The numeric type function INT(X) is used to change this value into an integer.
INT(X) converts any value of X into an integer, and returns the integer.

```
PRINT INT (12.73)
 12
Ok
PRINT INT (-7.99)
-8
Ok
```

INT(X) returns the maximum integer value smaller than the value of X. Consequently, if RND(1) * 200 is used as the value of INT(X),

```
INT(RND(1)*200)
```

random numbers from 0 to less than 200 (from 0 to 199) will be returned. Therefore to return random numbers from 0 to 200 you only need to use

```
INT((RND(1)*(200+1))
```

or

```
INT (RND(1)*201)
```

```
10 FOR L=1 TO 10
20 X=INT(RND(1)*201)
30 PRINT X;
40 NEXT L
```

This program will generate and display 10 random numbers from 0 to 200, as shown below.

```
RUN
 119  21   153  116  147  37  74  190
 128  94
```

The standard format for generating a random number from 0 to N is:

X = INT(RND(1) * (N + 1))

With this format, a random number from 0 to N will be assigned to X.

The following format is used to generate a random number from M to N.

X = INT(RND(1) * (N − M + 1)) + M

For example, to generate a random number from 2 to 15, M would be 2 and N would be 15. Therefore, N − M + 1 = 14.

X = INT(RND(1) * 14) + 2

Random Boxes

Let's make a program which draws 50 squares of different sizes and colors at different display locations.

```
10 SCREEN 5
20 COLOR ,1,1:CLS
30 FOR B=1 TO 50
40 SX=INT(RND(1)*220)+5
50 SY=INT(RND(1)*180)+5
60 ST=INT(RND(1)*40)+20
70 C=INT(RND(1)*14)+2
80 LINE (SX,SY)-STEP(ST,ST),C,BF
90 NEXT B
100 GOTO 100
```

The variables SX, SY are the top left coordinates of each square, ST is the length of one side, and C is the color code. Lines 40 through 70 assign random numbers to these variables with the RND(X) function. The value range for each of the random numbers is:

SX...5 to 224
SY...5 to 184
ST...20 to 59
C...2 to 15

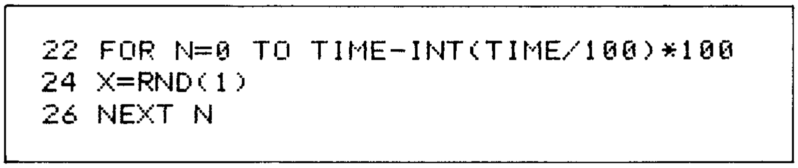
Line 80 then draws squares of different sizes and colors at different display locations.

Generating Different Random Numbers Each Time the Program is Executed

As can be seen if the above Random Boxes program is executed several times, each time the squares are positioned in the same locations. Actually, the computer contains a random number list, and each time a program that uses the RND(1) function is executed, the RND(1) function returns the values of this list in the same sequence, beginning with the first value on the list.

It is possible, however, to make a program which returns a different part of the random number list each time it is executed, and in this way obtain different results each time the program is executed.

One way to do this is to utilize the computer's internal timer by adding the following lines at the beginning of the program.



```
22 FOR N=0 TO TIME-INT(TIME/100)*100
24 X=RND(1)
26 NEXT N
```

The TIME variable used in line 22 is a special MSX-BASIC variable. The current value of the internal timer is always assigned to it. The internal timer value changes from 0 to 65535 in units of 1 approximately every 1/50 second. When the value reaches 65535 it returns to 0 and the process is repeated.
The value of the TIME variable when a program containing lines such as those shown above is executed will be any number from 0 to 65535. However, line 22 contains two TIME variables, and since the internal timer value is changing at a relative high speed, the values assigned to each of the two TIME variables will be slightly different. For instance, if an internal timer value of 42280 is assigned to the first TIME variable, a value of about 42281 will be assigned to the second TIME variable. Therefore, line 22 would be:

```
FOR N=0 TO 42280-INT(42281/100)*100
```

or

```
FOR N=0 TO 80
```

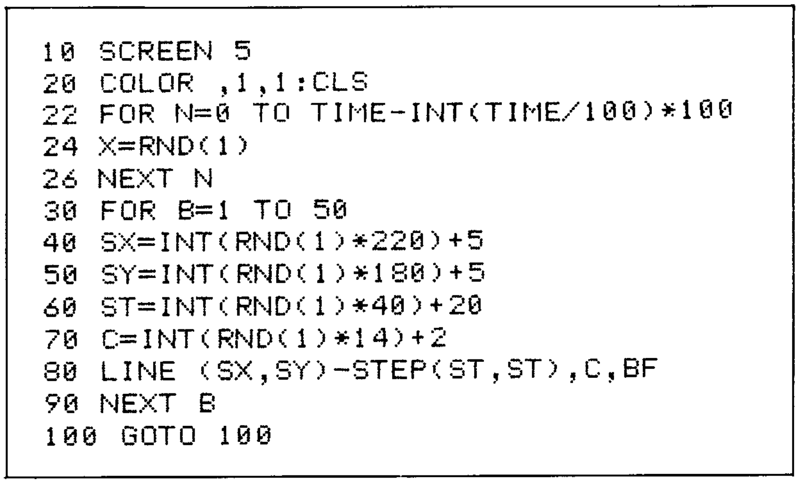
Or if the value assigned to the first TIME variable is 10900, and the value assigned to the second TIME variable is 10901, then the statement would becomes:

```
FOR N=0 TO 10900-INT(10901/100)*100
```

or

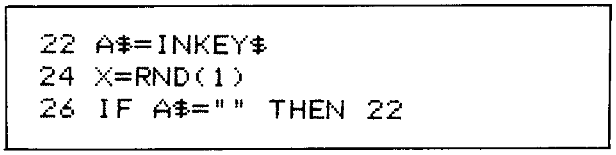
```
FOR N=0 TO 0
```

In this way, the final value of N in the FOR—NEXT statement when the program is executed will change depending on the value of the internal timer. When the final value of N is 80, X=RND(1) in line 24 will be executed 81 times by the FOR—NEXT loop. Consequently, the next time the RND(1) function is used in the program it will return the 82nd value on the internal random number list in the computer.
Let's add these three lines to the previous Random Box program.



```
10 SCREEN 5
20 COLOR ,1,1:CLS
22 FOR N=0 TO TIME-INT(TIME/100)*100
24 X=RND(1)
26 NEXT N
30 FOR B=1 TO 50
40 SX=INT(RND(1)*220)+5
50 SY=INT(RND(1)*180)+5
60 ST=INT(RND(1)*40)+20
70 C=INT(RND(1)*14)+2
80 LINE (SX,SY)-STEP(ST,ST),C,BF
90 NEXT B
100 GOTO 100
```

There is a second method for obtaining different random numbers each time a program is executed. This method utilizes the INKEY$ function which inputs data from the keyboard. To use the INKEY$ function the following three lines would be added to the program.



```
22 A$=INKEY$
24 X=RND(1)
26 IF A$="" THEN 22
```

The INKEY$ function is explained in detail on page 200. By means of the above three lines, X = RND(1) will continue to be executed until a key on the keyboard is pressed.

Add these three lines to the Random Box program and check the results.

STRING TYPE FUNCTIONS

- ●What Are String Type Functions?
- ●Processing Character Strings

WHAT ARE STRING TYPE FUNCTIONS?

A string type function processes a character string and returns the character string result.

There are seven string type functions in MSX2-BASIC.

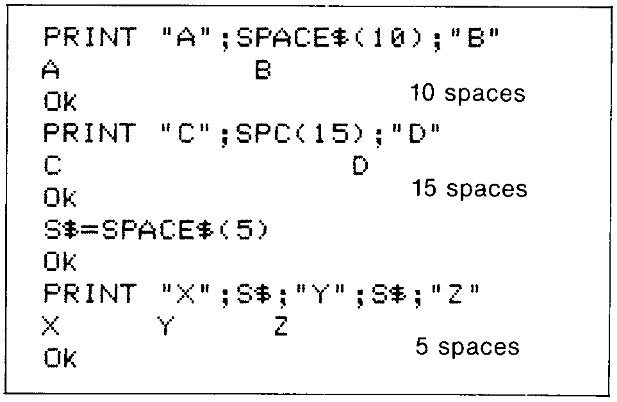
LEFT$(X$,N), MID$(X$,M[,N]), RIGHT$(X$,N)
SPACE$(N), STRING$(N,J) or STRING$(N,X$),
TAB(N), SPC(N)

Several of these functions take a numeric value input and return a character string, but they are classified as string type functions.

SPECIFYING SPACES SPACE$(N), SPC(N)

The functions SPACE$(N) and SPC(N) return N number of blank character spaces. The results are the same using either of the functions, but SPC(N) can only be used together with a PRINT statement.

Let's see how these functions work in the direct mode.



```
PRINT "A";SPACE$(10);"B"
A          B
Ok                           10 spaces
PRINT "C";SPC(15);"D"
C               D
Ok                           15 spaces
S$=SPACE$(5)
Ok
PRINT "X";S$;"Y";S$;"Z"
X     Y     Z
Ok                           5 spaces
```

A string type function always returns characters. Therefore, when the value is assigned with the LET statement, the variable also must be a string type variable.
In the above example,

```
S$=SPACE$(5)
```

assigns five spaces to the string type variable S$.

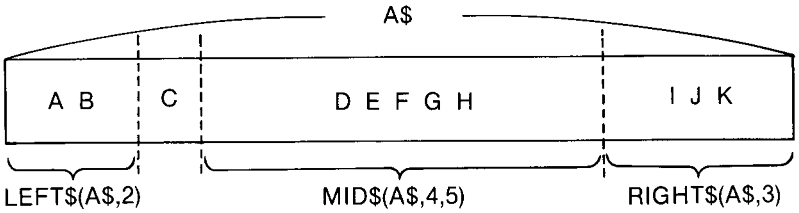
PROCESSING CHARACTER STRINGS
LEFT$(X$,N), MID$(X$, M, N), RIGHT$(X$, N)

The LEFT$(X$, N), MID$(X$, M, N), and RIGHT$(X$, N) string type functions return one part of a specified character string.
LET$(X$,N) returns N number of characters from the left side of the character string X$.
RIGHT$(X$,N) returns N number of characters from the right side of the character string X$.
MID$(X$,M,N) returns N number of characters beginning with the Mth character in the character string.

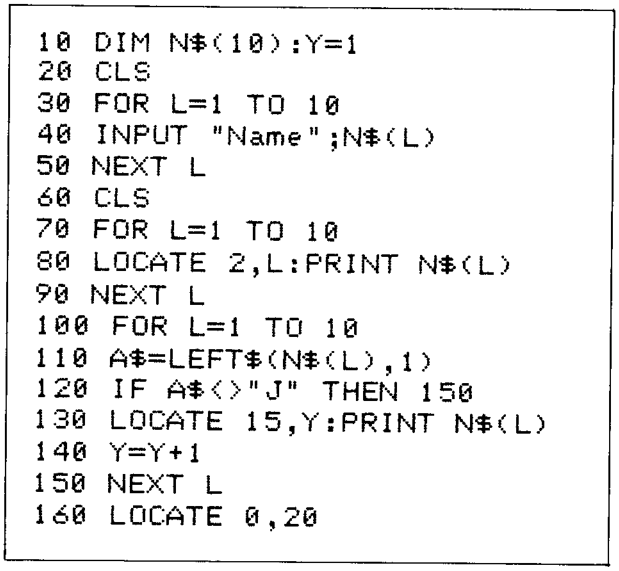
The following direct mode PRINT statements show the operation of these three functions.

```
A$="ABCDEFGHIJK"
Ok
PRINT LEFT$(A$,2)
AB
Ok
PRINT RIGHT$(A$,3)
IJK
Ok
PRINT MID$(A$,4,5)
DEFGH
Ok
```

As shown above, the LEFT$(X$,N), MID$(X$M,N), and RIGHT$(X$,N) functions are used to return a specified part of a character string.



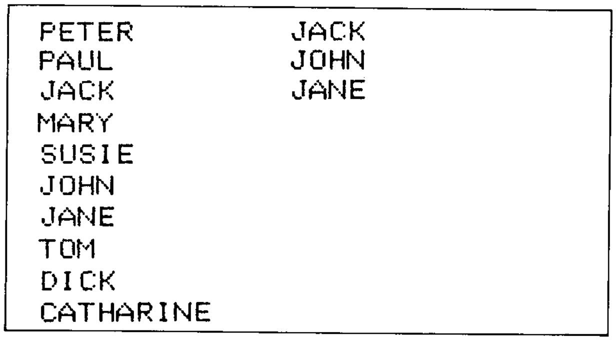
In the following program all of the character strings (personal names) that are entered with the INPUT statement are displayed, and then the LEFT$(X$,N) function is used to display separately all of the names which begin with J, that is, all of the character strings in which J is the first character on the left.



```
10 DIM N$(10):Y=1
20 CLS
30 FOR L=1 TO 10
40 INPUT "Name";N$(L)
50 NEXT L
60 CLS
70 FOR L=1 TO 10
80 LOCATE 2,L:PRINT N$(L)
90 NEXT L
100 FOR L=1 TO 10
110 A$=LEFT$(N$(L),1)
120 IF A$<>"J" THEN 150
130 LOCATE 15,Y:PRINT N$(L)
140 Y=Y+1
150 NEXT L
160 LOCATE 0,20
```

The INPUT statement in line 40 assigns personal names to the array variable N$(1)—N$(10). For example, the following names could be assigned in order beginning with N$(1);
PETER, PAUL, JACK, MARY, SUSIE, JOHN, JANE, TOM, DICK, CATHARINE

After all of the names are displayed by lines 70 to 90, then lines 100 to 150 display only the names that begin with J—JACK, JOHN, JANE—on the right side of the screen.



```
 PETER        JACK
 PAUL         JOHN
 JACK         JANE
 MARY
 SUSIE
 JOHN
 JANE
 TOM
 DICK
 CATHARINE
```

The LEFT$(X$, N) function is used in line 110 to display only the names which begin with J. Here only the first character at the left side of the character string assigned to the array variable N$ is assigned to A$. Then the IF—THEN statement in line 120 checks to see if this character is "J". If it is J, then the content of N$ is displayed at the right side of the screen.

FUNCTIONS WHICH CONVERT NUMERIC AND STRING TYPE DATA

- The Conversion Functions
- The Character Code

THE CONVERSION FUNCTIONS

Conversion functions are classified according to whether they return string type data when numeric type data is entered, or return numeric type data when string type data is entered. There are nine conversion functions in MSX2-BASIC.

ASC(X$), CHR$(X)
VAL(X$), STR$(X)
LEN(X$), INSTR([N,]X$, Y$)
BINS(X), OCT$(X), HEX$(X)

CHANGING THE TYPE OF NUMBERS
VAL(X$), STR$(X)

In BASIC, numbers can be treated either as numeric type numbers or as string type numbers. For example, if 123 is assigned to a numeric type variable, as in

```
A=123
```

then 123 is treated as the number one hundred twenty-three. But if 123 is assigned to a string variable, as in

```
A$="123"
```

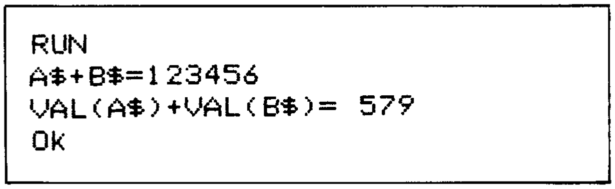
then 123 is treated as the characters one two three.

VAL(X$) changes a number treated as characters to a numeric type number.
STR$(X) changes a number treated as a number to a string type number.

Input the following program:

```
10 A$="123":B$="456"
20 X=VAL(A$):Y=VAL(B$)
30 PRINT "A$+B$=";A$+B$
40 PRINT "VAL(A$)+VAL(B$)=";X+Y
```

First the numbers 123 and 456 are assigned to the variables A$ and B$ as string type values. Then in line 20 they are converted to numeric type values. Lines 30 and 40 display the addition of the string type numbers and the numeric type numbers. The following illustration shows the display when the program is executed.



```
RUN
A$+B$=123456
VAL(A$)+VAL(B$)= 579
OK
```

STR$(X) performs the opposite function of VAL(A$). Execute the following program:

```
10 A=123:B=456
20 X$=STR$(A):Y$=STR$(B)
30 PRINT "A+B=";A+B
40 PRINT "STR$(A)+STR$(B)=";X$+Y$
RUN
A+B= 579
STR$(A)+STR$(B)= 123 456
Ok
```

When a numeric type number is changed to a string type number, the space before the numeric type number (where the + or − sign appears) is included as part of the string type value.

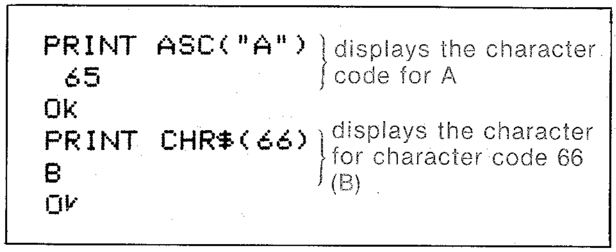
CHARACTER CODES AND FUNCTIONS

ASC(X$), CHR$(X)

Just as there are color codes, there are character codes for the characters used in BASIC. For instance, the character code for the capital A is 65 (decimal number). There are two functions in MSX2-BASIC which use the character codes.

ASC(X$) returns the character code of a character that is entered.
CHR$(X) returns the character of a character code that is entered.

These functions can be checked in the direct mode as follows:

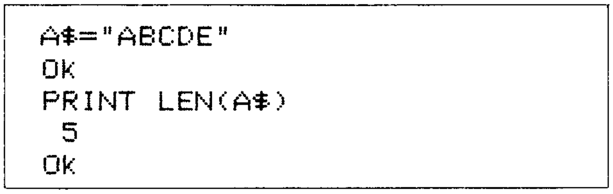


```
PRINT ASC("A")  } displays the character
 65             } code for A
Ok
PRINT CHR$(66)  } displays the character
B               } for character code 66
Ok              } (B)
```

Refer to the MSX2-BASIC Programming Reference Manual for a list of the character codes.

RETURNING THE LENGTH OF A CHARACTER STRING LEN(X$)

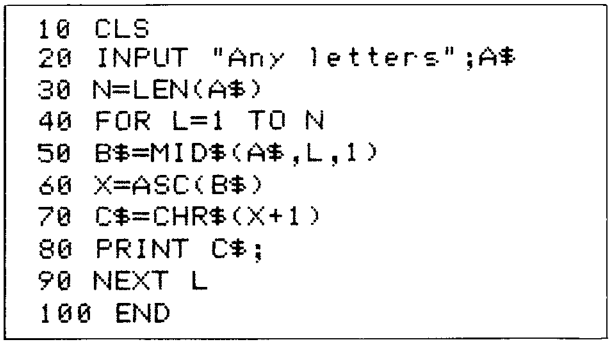
The LEN(X$) function returns the number of characters in character string X$ as numeric value data.



```
A$="ABCDE"
Ok
PRINT LEN(A$)
 5
Ok
```

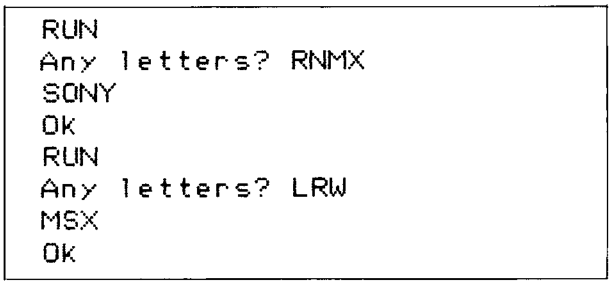
The number of characters, 5, in the character string "ABCDE" is returned by LEN(A$).

The following program uses the LEN(X$), MID$(X$, M, N), and CHR$(X) functions to convert each character assigned to the variable A$ by the INPUT statement into the character with a character code number one number higher than that of the input character.



```
10 CLS
20 INPUT "Any letters";A$
30 N=LEN(A$)
40 FOR L=1 TO N
50 B$=MID$(A$,L,1)
60 X=ASC(B$)
70 C$=CHR$(X+1)
80 PRINT C$;
90 NEXT L
100 END
```

Try inputting your own choice of letters, and see the results.



```
RUN
Any letters? RNMX
SONY
Ok
RUN
Any letters? LRW
MSX
Ok
```

DATA INPUT FUNCTIONS

- What Are Data Input Functions?
- Data Input from the Keyboard
- Inputting the Cursor Key Status

THE OPERATION OF DATA INPUT FUNCTIONS

The functions we have discussed up to now have all been functions which process an entered value and return the result. The functions discussed in this section are of a slightly different type. Values are not entered in these functions to be processed directly. The values entered in these functions are like signs which have special meanings. The signs tell the data input functions to input the status of input devices connected to the computer and the functions then return the status as data. (Some of these data input functions require no input value).

These functions make it possible to write programs which will perform an action based on the status of an input device—for example, to have a sprite pattern move in the direction of a cursor key that is pressed.

The Data Input Functions

MSX2-BASIC has 22 data input functions:

- Input from the screen
  CSRLIN, POS(X), POINT(X,Y)
- Input from the printer
  LPOS(X)
- Input from memory
  FRE(X), FRE(" "), PEEK(N), VARPTR(variable), VPEEK(N)
- Input from the keyboard
  INKEY$, INPUT$(X)
- Input from an I/O port
  INP(N)
- Input from the joystick, space bar, mouse, track ball, paddle, touch pad, or light pen
  STICK(N), STRIG(N), PDL(N), PAD(N)
- Input from a data file
  EOF(file number), INPUT$(N,[#]file number)
- Input from a disk
  DSKF(drive number), LOC(file number), VARPTR(#file number)
- Input from a machine language subroutine
  USR[X](I)

DATA INPUT FROM THE KEYBOARD INKEY$

When a key is pressed on the keyboard, the INKEY$ function returns the character of the pressed key as string type data.

The following simple program illustrates the operation of the INKEY$ function.

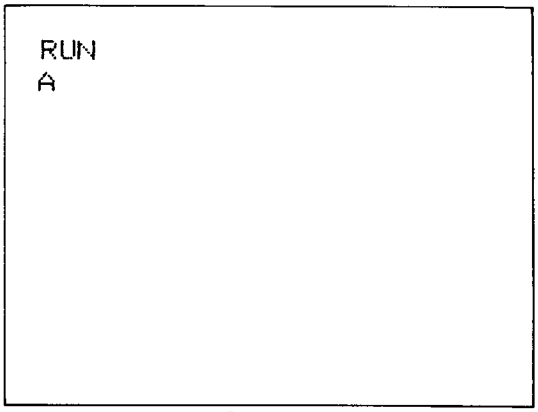
```
10 K$=INKEY$
20 PRINT K$;
30 GOTO 10
```

When this program is executed,

```
K$=INKEY$
```

in line 10 will be executed over and over. If no key is pressed when line 10 is executed, the INKEY$ function returns a character string with no data. This is called a **null string**. In the above program, when a null string is returned it is assigned to K$. But since there is no data in a null string, when the PRINT statement in line 20 displays the null string, the effect on the screen is the same as if nothing has happened.

When a key such as the A key is pressed while line 10 is being executed, the INKEY$ function will return the character A. (If the small a is pressed, it will return the small a.) This character is assigned to K$, and displayed by line 20, so the character A is displayed on the screen.



```
RUN
A
```

When A is pressed

When additional keys are pressed, the characters for these keys are also displayed. The screen resembles the command-wait condition, but the cursor is not displayed and this indicates that a program is being executed. Also, if the ↵ key is pressed, the next character is displayed at the beginning of the same line, instead of being displayed on the next line.
CTRL + STOP is used to stop the program.

Let's make another program which uses the INKEY$ function.

```
10 CLS
20 LOCATE 5,4
30 PRINT "┌───────────────┐"
40 LOCATE 5,14
50 PRINT "└───────────────┘"
60 FOR Y=5 TO 13 STEP 2
70 FOR X=6 TO 20
80 K$=INKEY$
90 IF K$="" THEN 80
100 LOCATE X,Y:PRINT K$
110 NEXT X
120 NEXT Y
130 LOCATE 0,22:END
```

This program illustrates one of the important uses of the INKEY$ function.
Take a look at the combination of the INKEY$ statement in line 80 with the IF—THEN statement in line 90.

```
80 K$=INKEY$
90 IF K$="" THEN 80
```

If no key is pressed when line 80 is executed, a null string is assigned to K$. If the content of K$ is a null string, then line 90 returns the program to line 80. Therefore, as long as no key is pressed, the program continues to repeat this loop.
No space between the double quotation marks (" ") in

```
K$=""
```

indicates a null string.

When a key is pressed, the program advances to line 100, and the character of the key that was pressed is displayed.

The INKEY$ function is used in this way very frequently in programs to have a program advance to the next step when a key is pressed.

The INKEY$ function can also be used to have the program advance to the next step only if a special key, such as the space bar, is pressed, as in the following program.

```
10 CLS
20 INPUT "Any letters";A$
30 K$=INKEY$
40 IF K$="" OR K$<>" " THEN 30
50 PRINT A$
```

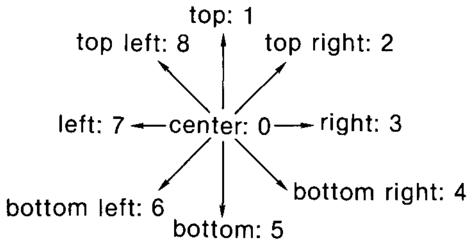
The IF—THEN statement in line 40 returns the program to line 30 if: 1) K$ is a null string; 2) if K$ is not the space bar. The program advances to line 50 only when the space bar is pressed.

INPUTTING THE STATUS OF THE CURSOR KEY STICK(N)

The STICK(N) function returns a numeric value which indicates the direction of the cursor key, the joy stick, the mouse, the track ball, or the touch pad.
The value of N determines whether the status of the cursor key or one of the other input devices is returned.

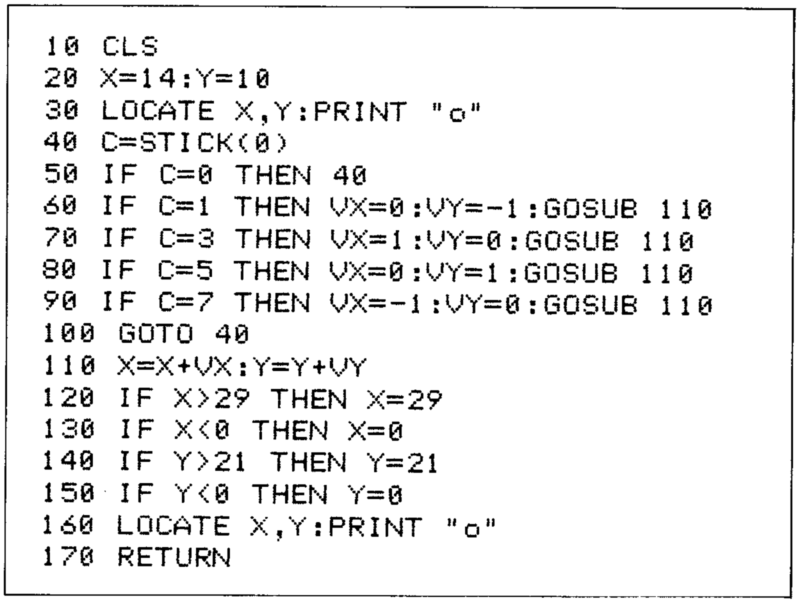
N	Device
0	cursor key
1	device connected to CONTROLLER A
2	device connected to CONTROLLER B

STICK(N) returns values from 0 to 8, which indicate the direction of the cursor key or other devices.



For example, for STICK(0), 0 will be returned when no cursor key is pressed.
When ↑ (top) is pressed 1 is returned, when ↓ (bottom) is pressed 5 is returned, and when ↑ and → are pressed together 2 is returned.

Let's use the STICK(N) function to make a program in which the cursor key controls the screen display.



```
10 CLS
20 X=14:Y=10
30 LOCATE X,Y:PRINT "o"
40 C=STICK(0)
50 IF C=0 THEN 40
60 IF C=1 THEN VX=0:VY=-1:GOSUB 110
70 IF C=3 THEN VX=1:VY=0:GOSUB 110
80 IF C=5 THEN VX=0:VY=1:GOSUB 110
90 IF C=7 THEN VX=-1:VY=0:GOSUB 110
100 GOTO 40
110 X=X+VX:Y=Y+VY
120 IF X>29 THEN X=29
130 IF X<0 THEN X=0
140 IF Y>21 THEN Y=21
150 IF Y<0 THEN Y=0
160 LOCATE X,Y:PRINT "o"
170 RETURN
```

In this program, the small letter "o" is displayed according to the direction the cursor key is moved. The "o" is first displayed at the 14, 10 location (lines 20,30). The STICK(0) function in line 40 returns the status of the cursor key. Lines 50 to 90 determine the values of the VX and VY variables according to the value returned by the STICK(0) function in order to specify the location where the next "o" will be displayed.
For instance, when → is pressed, STICK(0) returns the value 3, and in line 70 VX becomes 1 and VY becomes 0. Then the program jumps to the subroutine beginning at line 110.

In the subroutine, the next "o" display location is assigned to X and Y and the "o" is displayed. Lines 120 to 150 in the subroutine limit the size of the display area.

Chapter 8 Interrupts

MAKING INTERRUPTS

- ●What is an Interrupt?
- ●MSX2-BASIC Interrupts
- ●Making Interrupts

WHAT IS AN INTERRUPT?

An interrupt suspends program flow when a specific condition occurs during program execution and then performs a separate processing routine, called an interrupt processing program or an interrupt processing routine.
An interrupt is similar to a subroutine, except a subroutine is performed only when a GOSUB statement in the program is executed. In contrast, an interrupt processing routine is executed by an external condition, such as when the F1 key is pressed. Regular program execution is resumed when the execution of an interrupt processing routine is completed, just as in the case of a subroutine.

MSX2-BASIC INTERRUPTS

There are five ways to provide an interrupt in MSX2-BASIC.

- ●When a function key (F1—F10) is pressed.
- ●When the space bar is pressed, or when a mouse, joystick, track ball, or touchpad button is pressed.
- ●When CTRL+STOP is pressed.
- ●When there is sprite overlap.
- ●When a specified period of time has passed (using the internal timer).

MAKING INTERRUPTS

The following five interrupt declaration statements are used to provide for an interrupt of the main routine during execution of an MSX2-BASIC program.

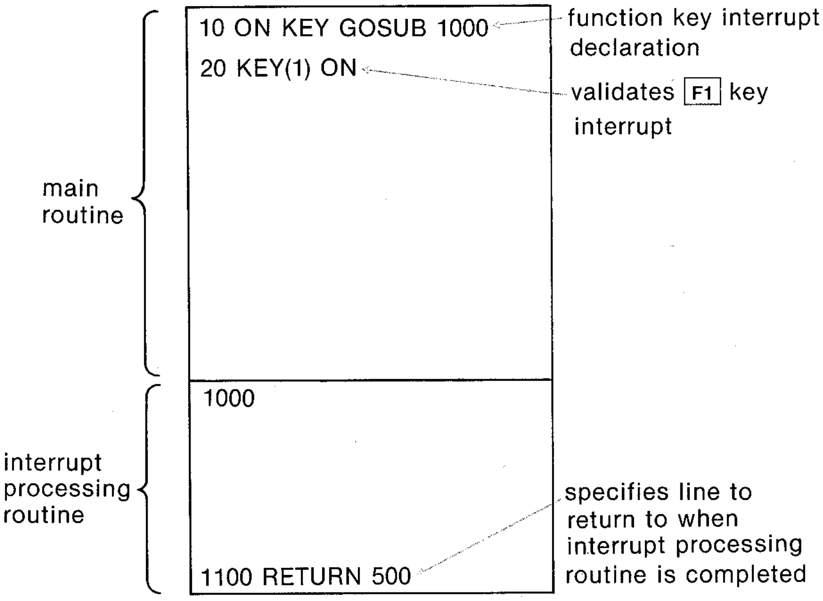
Interrupt	Interrupt Declaration Statement
by a function key	ON KEY GOSUB line number [, line number]...
by the space bar, or joystick, mouse, track ball, or touch pad button	ON STRIG GOSUB line number [, line number]...
by CTRL + STOP	ON STOP GOSUB line number
by sprite overlap	ON SPRITE GOSUB line number
by the internal timer	ON INTERVAL = interval time GOSUB line number

An interrupt declaration statement declares what will cause the interrupt and specifies the line number of the first line of the interrupt processing routine.

A statement which validates the interrupt is executed immediately after an interrupt declaration statement. There are five validation statements.

Interrupt	Interrupt Validation Statement
by a function key	KEY(N) ON (N is 1—10. 1 = F1 key)
by the space bar, or joystick, mouse, track ball, or touch pad button	STRIG(N) ON (N is 0—4, 0 = space bar)
by CTRL + STOP	STOP ON
by sprite overlap	SPRITE ON
by the internal timer	INTERVAL ON

The following program will execute the interrupt processing routine beginning in line 1000 when the F1 key is pressed.



```
10 ON KEY GOSUB 1000
20 KEY(1) ON

1000

1100 RETURN 500
```

function key interrupt declaration
validates F1 key interrupt
main routine
interrupt processing routine
specifies line to return to when interrupt processing routine is completed

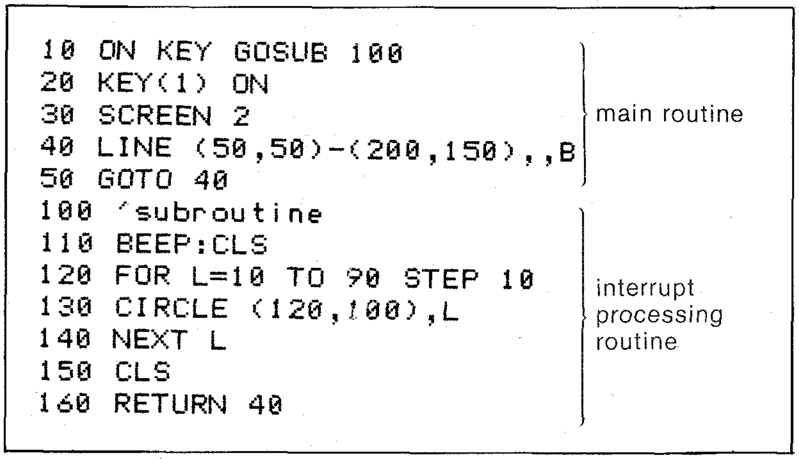
The main routine is executed when a program like this is run, but when the F1 key is pressed during execution, the program jumps to line 1000 and executes the interrupt processing subroutine. When execution is completed, RETURN 500 at the end of the processing routine returns the program to line 500 of the main routine.

PROGRAMS USING INTERRUPTS

- Function Key Interrupts
- Invalidating an Interrupt
- Interrupt Hold
- Sprite Overlap Interrupt

A FUNCTION KEY INTERRUPT PROGRAM

The following program illustrates the use of the F1 key to make an interrupt.



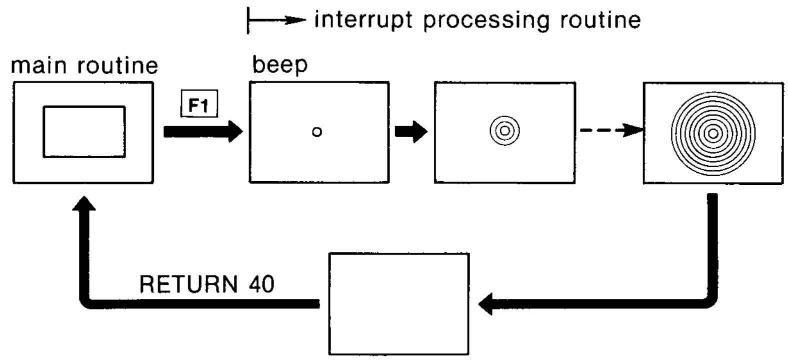
```
10 ON KEY GOSUB 100
20 KEY(1) ON
30 SCREEN 2
40 LINE (50,50)-(200,150),,B
50 GOTO 40
100 'subroutine
110 BEEP:CLS
120 FOR L=10 TO 90 STEP 10
130 CIRCLE (120,100),L
140 NEXT L
150 CLS
160 RETURN 40
```

main routine

interrupt processing routine

Lines 10 and 20 of this program specify that the program will jump to the interrupt processing routine beginning at line 100 when the F1 key is pressed.

A rectangle is continuously displayed by lines 40 and 50 in the main program when the program is executed. An interrupt occurs when the F1 key is pressed, and the program jumps to line 100. A beep is produced and the rectangle disappears (BEEP:CLS). Then 9 concentric circles are drawn. The screen is cleared after the final circle is drawn and the program returns to line 40.


interrupt processing routine
main routine
beep
F1
RETURN 40

INVALIDATING AN INTERRUPT KEY(N) OFF

Add the following line to the above program.

```
105 KEY(1) OFF
```

When the program is executed with this line added, an interrupt occurs the first time the F1 key is pressed, but there is no interrupt when F1 is subsequently pressed. This is because

```
KEY(1) OFF
```

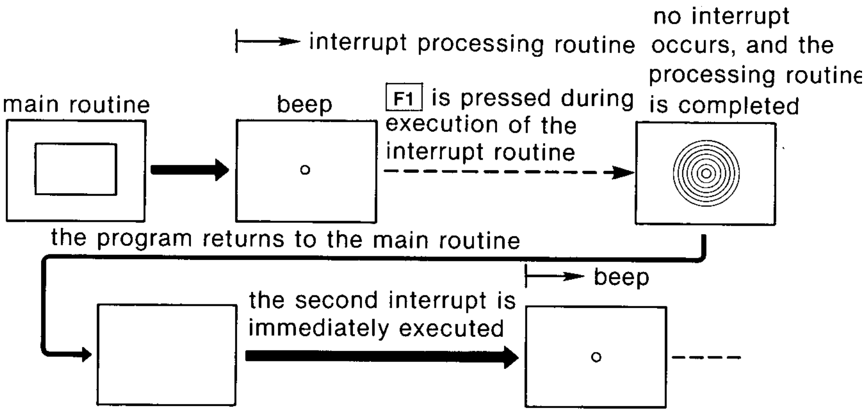
in line 105 is executed when the interrupt processing routine is executed and this invalidates the F1 key interrupt.
The KEY(N) OFF statement invalidates a function key interrupt. N specifies the function key number.

The following table shows the five MSX2-BASIC statements which invalidate interrupts.

Interrupt	Interrupt Invalidation Statement
by a function key	KEY(N) OFF
by the space bar, or joystick, mouse, track ball, or touch pad button	STRIG(N) OFF
by CTRL + STOP	STOP OFF
by sprite overlap	SPRITE OFF
by the internal timer	INTERVAL OFF

INTERRUPT HOLD

Additional interrupts are placed in a hold condition when the execution of a program is shifted to an interrupt processing routine by an interrupt. If another interrupt is attempted during the hold condition, it will not be executed immediately. Instead, the program will first be returned to the main routine by the RETURN statement at the end of the interrupt processing routine. Upon return to the main routine, the —ON statement is automatically executed and the second interrupt then occurs in place of execution of the main routine.
Therefore, when an interrupt is attempted during a hold condition, the interrupt is remembered by the computer but it is not executed until the initial interrupt processing routine is completed.
In the program on page 209, nine circles are drawn by the interrupt processing routine when the F1 key is pressed. If the F1 key is pressed again before the last circle is drawn, a second interrupt will not be executed immediately. But the second interrupt will be executed as soon as the program returns to the main routine after the last circle is drawn, and instead of drawing a rectangle, the program will again draw the circles.

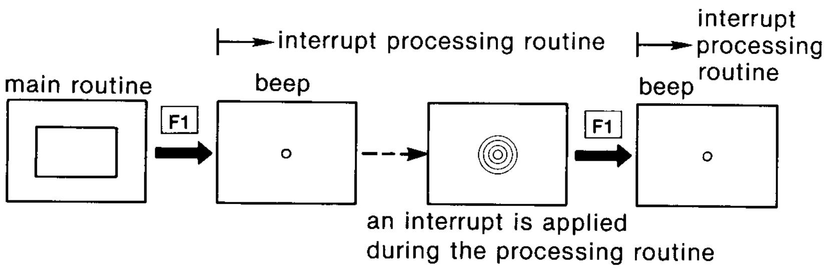

interrupt processing routine
no interrupt occurs, and the processing routinε is completed
main routine
beep
F1 is pressed during execution of the interrupt routine
the program returns to the main routine
beep
the second interrupt is immediately executed

RE-VALIDATING AN INTERRUPT DURING AN INTERRUPT PROCESSING ROUTINE KEY(N) ON

A statement such as KEY(1) ON can be included in a interrupt processing routine to re-validate the interrupt when the processing routine is executed. Then if the interrupt is again applied after the interrupt processing routine has begun, the processing routine will be executed again from the beginning.'

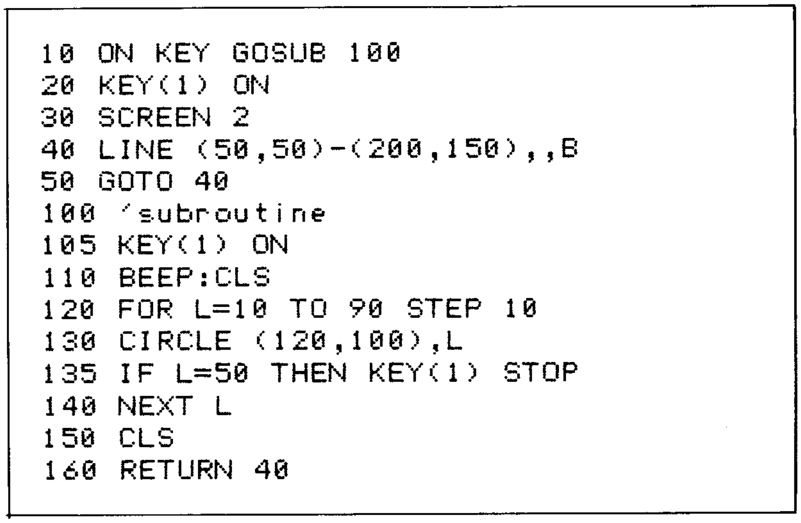
```
10 ON KEY GOSUB 100
20 KEY(1) ON
30 SCREEN 2
40 LINE (50,50)-(200,150),,B
50 GOTO 40
100 'subroutine
105 KEY(1) ON
119 BEEP:CLS
120 FOR L=10 TU 90 STEP 10
130 CIRCLE (120,100),L
140 NEXT L
150 CLS
160 RETURN 40
```

KEY(1) ON has been added to the previous program as line 105. This will cause an interrupt to be executed when the F1 key is pressed while the processing subroutine is drawing the circles. The program will immediately return to line 100 and execute the processing subroutine again from the beginning.


interrupt processing routine
interrupt processing routine
main routine
beep
beep
F1
F1
an interrupt is applied during the processing routine

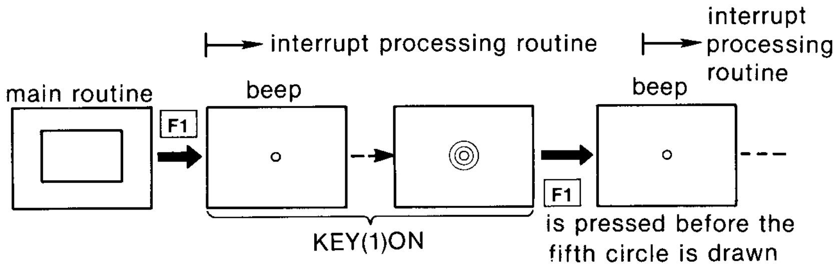
HOLDING AN INTERRUPT IN A PROGRAM KEY(N) STOP

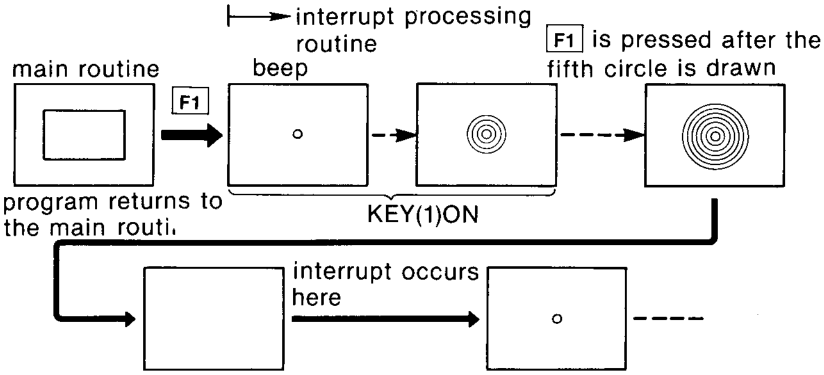
The KEY(N) STOP statement is used to reinstate an interrupt hold condition after an interrupt has been validated by a KEY(N) ON statement in a interrupt processing routine.



```
10 ON KEY GOSUB 100
20 KEY(1) ON
30 SCREEN 2
40 LINE (50,50)-(200,150),,B
50 GOTO 40
100 'subroutine
105 KEY(1) ON
110 BEEP:CLS
120 FOR L=10 TO 90 STEP 10
130 CIRCLE (120,100),L
135 IF L=50 THEN KEY(1) STOP
140 NEXT L
150 CLS
160 RETURN 40
```

Line 135, which executes KEY(1) STOP when the value of L reaches 50, has been added to the previous program. Another interrupt will occur immediately if the F1 key is pressed before the fifth circle drawn. But the interrupt hold condition is reinstated by line 135 after the fifth circle is drawn, and subsequently an interrupt will not occur immediately when the F1 key is pressed.


interrupt processing routine
interrupt processing routine
main routine
beep
beep
F1
F1
is pressed before the fifth circle is drawn
KEY(1)ON

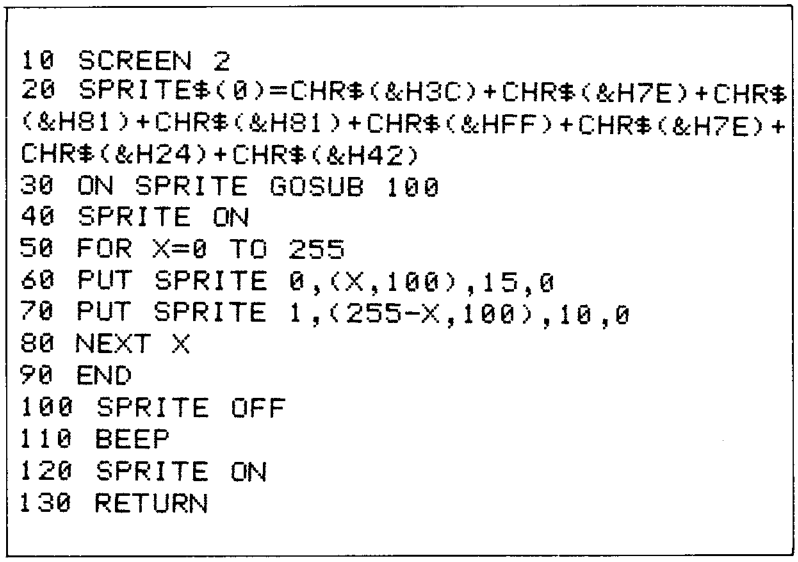

interrupt processing routine
F1 is pressed after the fifth circle is drawn
main routine
beep
F1
program returns to the main routi.
KEY(1)ON
interrupt occurs here

The KEY(N) STOP statement places a function key in the hold condition. N specifies the function key number.
The following table shows the five MSX2-BASIC statements which hold interrupts.

Interrupt	Interrupt Hold Statement
by a function key	KEY(N) STOP
by the space bar, or joystick, mouse, track ball, or touch pad button	STRIG(N) STOP
by CTRL + STOP	STOP STOP
by sprite overlap	SPRITE STOP
by the internal timer	INTERVAL STOP

SPRITE OVERLAP INTERRUPT

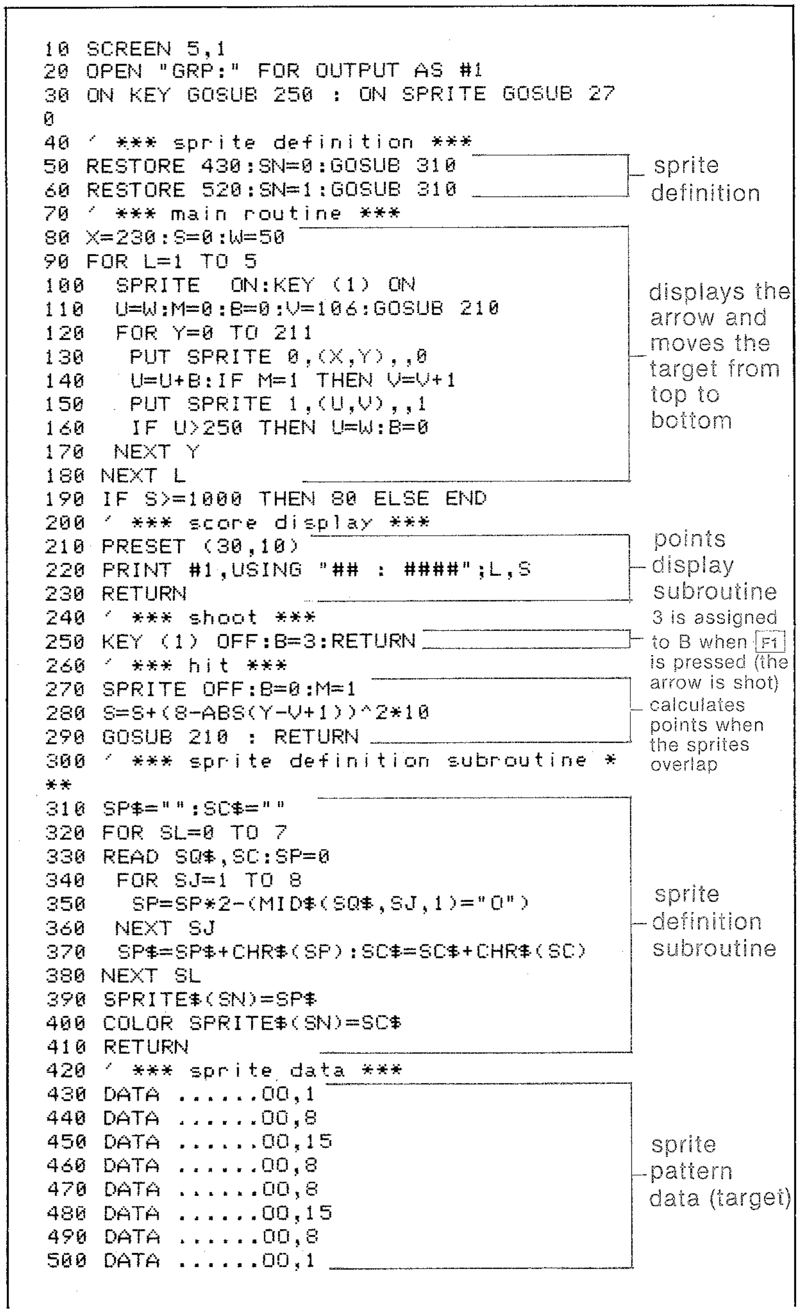
The ON SPRITE GOSUB and the SPRITE ON statements generate an interrupt when two or more sprite patterns overlap by one dot or more. In the following program UFOs fly from the left and right, and a beep occurs when the UFOs overlap.



```
10 SCREEN 2
20 SPRITE$(0)=CHR$(&H3C)+CHR$(&H7E)+CHR$
(&H81)+CHR$(&H81)+CHR$(&HFF)+CHR$(&H7E)+
CHR$(&H24)+CHR$(&H42)
30 ON SPRITE GOSUB 100
40 SPRITE ON
50 FOR X=0 TO 255
60 PUT SPRITE 0,(X,100),15,0
70 PUT SPRITE 1,(255-X,100),10,0
80 NEXT X
90 END
100 SPRITE OFF
110 BEEP
120 SPRITE ON
130 RETURN
```

Sample Program

The following program is an archery game that generates interrupts by the F1 key and by sprite overlap.
An arrow is shot when the F1 key is pressed. Points are awarded depending on what part of the target the arrow hits. Five arrows can be shot. When total points exceed 1000, a free game is awarded.



```
10 SCREEN 5,1
20 OPEN "GRP:" FOR OUTPUT AS #1
30 ON KEY GOSUB 250 : ON SPRITE GOSUB 27
0
40 ' *** sprite definition ***
50 RESTORE 430:SN=0:GOSUB 310
60 RESTORE 520:SN=1:GOSUB 310
70 ' *** main routine ***
80 X=230:S=0:W=50
90 FOR L=1 TO 5
100  SPRITE  ON:KEY (1) ON
110  U=W:M=0:B=0:V=106:GOSUB 210
120  FOR Y=0 TO 211
130   PUT SPRITE 0,(X,Y),,0
140   U=U+B:IF M=1 THEN V=V+1
150   PUT SPRITE 1,(U,V),,1
160   IF U>250 THEN U=W:B=0
170  NEXT Y
180 NEXT L
190 IF S>=1000 THEN 80 ELSE END
200 ' *** score display ***
210 PRESET (30,10)
220 PRINT #1,USING "## : ####";L,S
230 RETURN
240 ' *** shoot ***
250 KEY (1) OFF:B=3:RETURN
260 ' *** hit ***
270 SPRITE OFF:B=0:M=1
280 S=S+(8-ABS(Y-V+1))^2*10
290 GOSUB 210 : RETURN
300 ' *** sprite definition subroutine *
**
310 SP$="":SC$=""
320 FOR SL=0 TO 7
330 READ SQ$,SC:SP=0
340  FOR SJ=1 TO 8
350   SP=SP*2-(MID$(SQ$,SJ,1)="O")
360  NEXT SJ
370  SP$=SP$+CHR$(SP):SC$=SC$+CHR$(SC)
380 NEXT SL
390 SPRITE$(SN)=SP$
400 COLOR SPRITE$(SN)=SC$
410 RETURN
420 ' *** sprite data ***
430 DATA ......OO,1
440 DATA ......OO,8
450 DATA ......OO,15
460 DATA ......OO,8
470 DATA ......OO,8
480 DATA ......OO,15
490 DATA ......OO,8
500 DATA ......OO,1
```

sprite definition

displays the arrow and moves the target from top to bottom

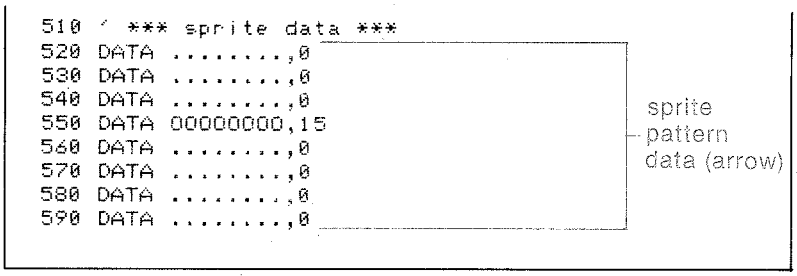
points display subroutine

3 is assigned to B when F1 is pressed (the arrow is shot)

calculates points when the sprites overlap

sprite definition subroutine

sprite pattern data (target)



```
510 ' *** sprite data ***
520 DATA ........,0
530 DATA ........,0
540 DATA ........,0
550 DATA OOOOOOOO,15
560 DATA ........,0
570 DATA ........,0
580 DATA ........,0
590 DATA ........,0
```

sprite pattern data (arrow)

The OPEN statement in line 20 is required in order to display characters on the graphic mode screen. This is explained in Chapter 9.

Chapter 9
Processing Files

FILES AND FILE DEVICES

- ●What are Files?
- ●File Devices
- ●Operating Program Files
- ●File Management

FILES AND FILE NAMES

The computer interacts with devices connected to it in operating programs or the data in a program. This interaction can be compared with keeping a diary. Let's suppose that you have several bookcases in your room. In one of the bookcases there is a notebook with the title "Diary" written on it. When you want to read your diary or write in it, you first go to the bookcase, then search for the notebook titled "Diary," and when you find it you remove it from the bookcase.

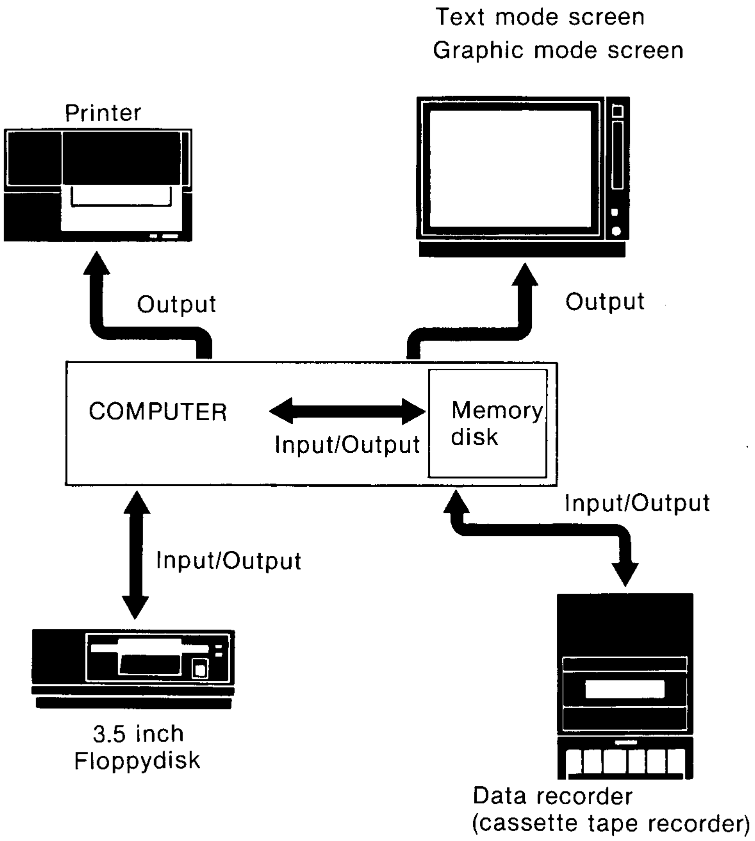
The computer performs a similar process. But instead of a notebook, a computer uses a file. A file is different from a notebook in that it is not something that you can actually pick up and touch. You can think of it as an area formed on a cassette tape or floppydisk.

File Names

The title "Diary" is used to distinguish the notebook you use for your diary from other notebooks in your bookcase. In the same way, you give names to the files you use with your computer. Such a name is called a **file name**.

FILE DEVICES AND DEVICE NAMES

Your bookcase corresponds to the file device connected to your computer. There are two types of file devices—input/output devices (cassette tape recorder, floppydisk drive, etc.), and devices which accept output only (printer, video monitor, etc.). MSX2-BASIC has special commands and statements which allows it to operate files in conjunction with the various file devices connected to the computer. The following diagram shows the file devices that can be used with MSX2-BASIC.



Floppydisks are used in the built-in floppydisk drive in your computer, or in a separate floppydisk drive unit attached to your computer.

Device Names

The device to be used must be specified in order for programs and data to interact with file device files. Therefore, each type of file device is given a device name, as shown in the following table.

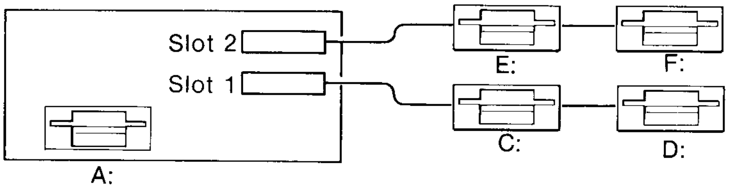
File device	Device name
Data recorder (cassette tape recorder)	CAS:
Text mode screen	CRT:
Graphic mode screen	GRP:
Printer	LPT:
Floppydisk A drive B drive ⋮ H drive	 A: B: } drive names ⋮ H:
Memory disk	MEM:

The built-in floppydisk drive in a computer is called drive A. If the computer has two built-in disk drives, they are called drive A and drive B. For computers without built-in disk drives, the external disk drive unit connected to the cartridge slot is called drive A. If additional disk drive units are connected, they are called drive B, drive C, and so forth. The device name given to a disk (such as A:, B:) is also called the drive name.

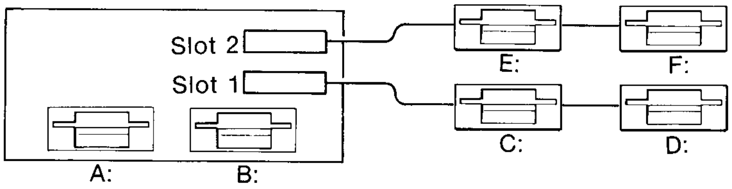
Drive Names for Disk Drives

The following illustrations show the drive names for possible combinations of built-in disk drives and external disk drive units connected to the cartridge slots.

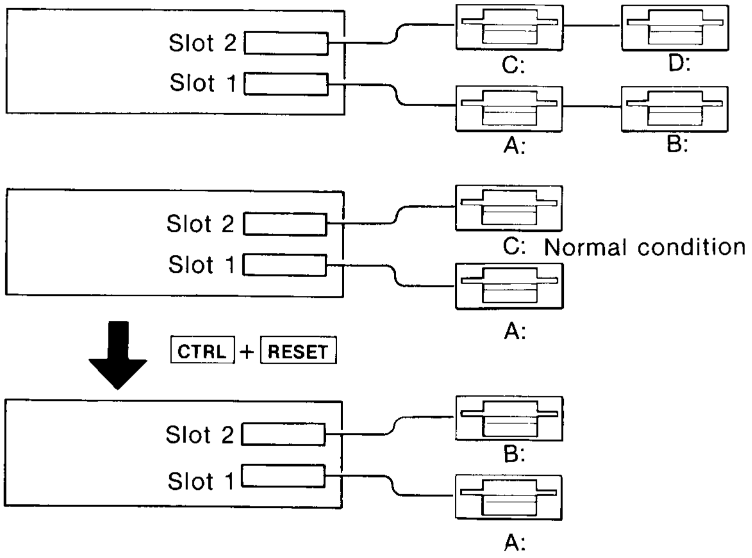
A computer with one internal disk drive:



A computer with two internal disk drives:



A computer without internal disk drives:



FILE NAME RULES AND THE TYPE NAME

A file name must begin with a letter of the alphabet, and can have up to 6 characters for a cassette tape file, and up 8 characters for other files.

A file name cannot be omitted for a floppydisk file. It can be omitted with other devices, but it is good practice to always use a file name for cassette tape or memory disk files in order to distinguish different files.

The type name is added to the file name by first writing a period (.) followed by up to three characters. The type name is used to distinguish different types of files, such as BASIC files or ASCII files, or to add characters to file names when eight characters are not sufficient to distinguish between two different files. The following examples show the two primary uses of type names.

Example 1

TEST.BAS BAS is added as the type name to indicate that the file is a BASIC program.

TEST.ASC ASC is added as the type name to indicate that the file has been saved in the ASCII format.

TEST.DAT DAT is added as the type name to indicate that the file is a data file.

Example 2

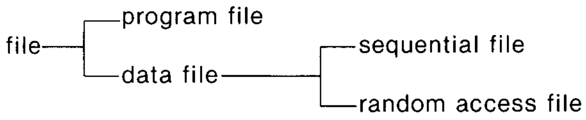
$\left.\begin{array}{l}\text{TESTPROG1}\\\text{TESTPROG2}\end{array}\right\}$ Since the TESTPROG file name is 8 characters, the computer will not distinguish between the two different files.

$\underbrace{\text{TESTPROG}}_{\text{file name}}.\underbrace{\text{001}}_{\text{type name}}$ $\underbrace{\text{TESTPROG}}_{\text{file name}}.\underbrace{\text{002}}_{\text{type name}}$

Type names can be added to make two different names for the two files.

PROGRAM FILES AND DATA FILES

When a file contains a program, it is called a program file. A file that contains data is called a data file. Data files are divided into sequential files and random access files, depending on the method used in inputting and outputting (writing in and reading out) data.



Here we will explain all of the commands used for operating program files for all devices. Data files will be covered in the next section.

OPERATING PROGRAM FILES

The following commands are used to save or load files, or to merge programs in memory.

CSAVE, CLOAD .... for cassette tape only
SAVE, LOAD, MERGE........ the device can be specified

Saving, Loading, and Merging Programs SAVE, LOAD, MERGE

A program saved with the CSAVE statement can be loaded only by the CLOAD statement, and a program saved by the SAVE statement can only be loaded by the LOAD statement. The file name that was used when a program was saved is specified to load the program. The MERGE statement loads a program and merges it with a program already stored in memory. Only programs saved in the ASCII format can be merged.

Save command	Save format	Load command
CSAVE	Intermediate language	CLOAD
SAVE (cassette tape or memory disk)	ASCII format	LOAD or MERGE
SAVE (floppydisk)	Intermediate language	LOAD
SAVE, A (floppydisk only)	ASCII format	LOAD or MERGE

For cassette tape, the CSAVE command saves a program in intermediate language, and the SAVE command saves a program in the ASCII format. Only the SAVE command can be used with floppydisks. If the A option is added to the floppydisk SAVE command, the program will be saved in the ASCII format. If it is omitted, the program will be saved in intermediate language.

Only the SAVE command can be used for the memory disk, also. Programs saved on the memory disk are always saved in the ASCII format.

Program File Operation Using Cassette Tape CSAVE, CLOAD

A program stored in memory can be saved on cassette tape in the following two ways:

`CSAVE "PROG"`—— saved in intermediate language with the file name PROG— ①

`SAVE "CAS:PROG"`— saved in the ASCII format with the file name PROG— ②

To load the program saved in ①, above, execute:

```
CLOAD "PROG"
```

To load the program saved in ②, above, execute:

```
LOAD "CAS:PROG"
```

To merge the program saved in ②, above, with another program stored in memory, execute:

```
MERGE "CAS:PROG"
```

Program File Operation Using a Floppydisk SAVE, LOAD

A program stored in memory can be saved on a disk (drive A) in the following two ways:

`SAVE "A:PROG.BAS"` saved in intermediate language with the file/type name PROG.BAS— ①

`SAVE "A:PROG.ASC",A` saved in the ASCII format with the file/type name PROG.BAS— ②

The programs saved in ① and ②, above, can be loaded by executing:

```
LOAD "A:PROG.BAS"

LOAD "A:PROG.ASC"
```

To merge the program saved in ②, above, with another program stored in memory, execute:

```
MERGE "A:PROG.ASC"
```

If only one disk drive is used, the drive name (A:) can be omitted. If the drive name is omitted when two or more disk drives are used, the disk drive in current use is selected.

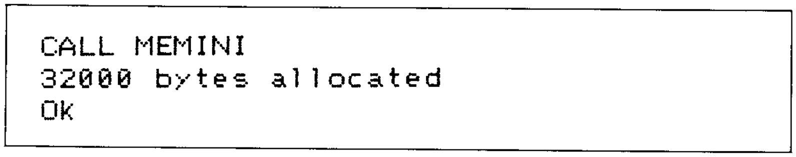
Program File Operation Using the Memory Disk

In MSX2-BASIC a part of internal memory separate from the part that stores programs can be used to save and store a program temporarily. Since this part of memory is used like a floppydisk, it is called the **memory disk function**. In contrast, the part of memory which normally remembers programs is called the program area. Programs stored in the program area can be displayed on the screen with the LIST command and executed with the RUN command. If a program stored in the program area is saved on the memory disk, the program area can be used to write another program.
However, the instant you turn off the computer, the program in the memory disk is erased.

To use the memory disk you must first initialize it with the CALL MEMINI (memory initialize) statement.

CALL MEMINI [(size)]

The amount of memory to be used by the memory disk is specified by the size in the CALL MEMINI statement. Size is specified in bytes, and can be from 1023 bytes to 32767 bytes. The default setting is 32767 bytes when the size specification is omitted.
When the CALL MEMINI statement is executed, the screen displays the size specified for the memory disk.



```
CALL MEMINI
32000 bytes allocated
Ok
```

Once the CALL MEMINI statement has been executed, a program in the program area can be saved on the memory disk with the following command:

```
SAVE "MEM:PROG.BAS"
```
—saved in the ASCII format with tr
file/type name PROG.BAS

To load a program from the memory disk to the program area, execute the following command:

```
LOAD "MEM:PROG.BAS"
```

To merge a program from the memory disk with a program in the program area, execute:

```
MERGE "MEM:PROG.BAS"
```

To Load and Execute a Program RUN

The RUN command followed by the file/type name specification is used to load a program from a floppydisk or the memory disk and execute it immediately.

RUN "[device name] file name [.type name]"

```
RUN "PROG.BAS"-----"PROG.BAS" in drive A is loaded
                                      and executed
RUN "MEM:PROG.BAS"----"PROG.BAS" in the memory disk
                          is loaded and executed
```

FILE MANAGEMENT

Displaying File Names FILES, CALL MFILES
To display the names of the files on a floppydisk, execute:

```
FILES
```

The file name/type name is specified to check if a specific file is on a disk

```
FILES "PROG.BAS"
```

To display the names of the files on the memory disk, execute:

```
CALL MFILES
```

CALL MFILES

The number of bytes remaining for use on the memory disk is also displayed after the CALL MFILES command is executed.

Erasing Files KILL, CALL MKILL
The KILL command is used to erase a file on a floppydisk. For instance, to erase a file named "TEST.DAT" you would execute:

```
KILL "TEST.DAT"
```

CALL MKILL is used to erase a file on the memory disk. To erase a file named "PROG.BAS" you execute:

```
CALL MKILL ("PROG.BAS")
```

CALL MKILL ("file name [.type name]")

Changing a File Name NAME, CALL MNAME
The NAME command is used to change the file name and/or a type name of a file on a floppydisk.

NAME "[drive name] old file name [.old type name]" AS "new file name [.new type name]"

CALL MNAME is used to change the file name and/or type name of a file on the memory disk.

CALL MNAME ("old file name [.old type name]") AS ("new file name [.new type name]")

```
NAME "OLD.BAS" AS "NEW.BAS"
```

————"OLD.BAS" in drive A is changed to "NEW.BAS"

```
CALL MNAME ("OLD.BAS" AS "NEW.BAS")
```

————"OLD.BAS" in the memory disk is changed to "NEW.BAS"

AUTO-START PROGRAM FILE

An auto-start program will automatically be loaded from a disk and executed when the computer is turned on or when the RESET button is pressed.
Specify

```
AUTOEXEC.BAS
```

as the file name/type name of the program when it is saved on a disk to create an auto-start program. Only one auto-start program can be created on one disk.

SEQUENTIAL FILE OPERATION

- What is a Sequential File?
- Writing Data in a Sequential File
- Adding Data
- Writing Characters on a Graphic Screen
- The Number of Files Which Can Be Opened at One Time

SEQUENTIAL FILES AND RANDOM ACCESS FILES

Data files are used in handling data to be processed by a BASIC program. For example, in the case of a telephone directory program, the program itself is saved as a program file, but the data for the people's names and telephone numbers which the program processes is saved as a data file.

There are two types of data files: 1) a sequential file which writes and reads data in sequence from the beginning of the file; and 2) a random access file which writes and reads data at any specified place in the file. The devices that can use sequential files and random access files are shown below.

Device Which Can Use Sequential Files	Devices Which Can Use Random Access Files
cassette tape floppydisk printer text/graphic modes screen memory disk	floppydisk

In this section, a floppydisk in drive A will be used as a representative device to explain sequential files.
The following statements are used for data input/output to a sequential file.

OPEN	opens a file
PRINT# PRINT# USING	writes data in a file
INPUT# LINE INPUT#	reads data from a file
CLOSE	close a file

WRITING DATA IN A SEQUENTIAL FILE

OPEN FOR OUTPUT

The procedure for outputting data to a sequential file is:
(1) Open the file with the OPEN statement.
(2) Write the data in the file with the PRINT# statement.
(3) Close the file with the CLOSE statement.

The format for the OPEN statement to output data is:

OPEN "[device name] [file name [.type name]]" FOR OUTPUT AS [#] file number

The OPEN statement prepares a file with the specified file name in the specified device to which data will be output. When the computer writes data in a file or reads data from a file, one part of the memory is reserved as a buffer. The buffer stores data temporarily until a fixed amount has been stored, and then outputs or inputs the data. A maximum of 16 buffers can be specified for use in MSX2-BASIC (the maximum is 7 for a floppydisk). The file number specified in the OPEN statement determines which one of the 16 buffers will be used. The initial default setting is 1.

After a file has been opened with the OPEN statement, the PRINT# statement is used to output data to the file. The format is:

PRINT# file number, [expression] [separator] [expression...]

The file number is the same file number that was specified in the OPEN statement.
Each time the PRINT# statement is executed, the return code (&H0D) and the line feed code (&H0A) are automatically added to the data that have been written by the PRINT# statement. These two codes serve as a sign which separates the data from the next data written in the file. When the data are character type data, a comma in parentheses (",") can be used to separate multiple pieces of data in one PRINT# statement, as follows:

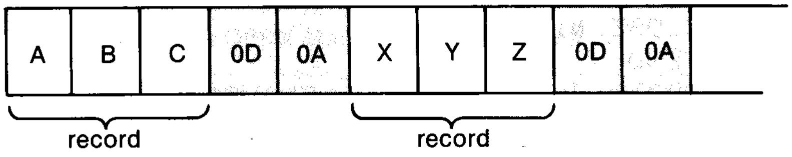
```
PRINT #1,A$;",";B$
```

Since the comma serves the function of separating the data, A$ and B$ will be treated as different pieces of data when the data is read. (If the data are numeric type data, separations are automatically made between each item of data, so no special sign is required.)

Each group of data separated by an 0D, 0A pair is called a record. For example if "ABC" is assigned to the variable A$, and "XYZ" is assigned to the variable B$, and

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

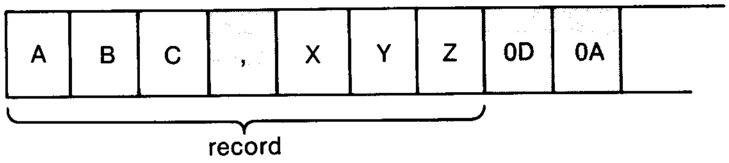
is executed, the data would be written on the disk as follows:



Or if

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

is executed, with both variables written in one PRINT# statement and separated by ",", the data will be written as follows:



In this case the data ABC and XYZ are written in one record, but the comma separates them into two sets of data.
After the data has been output, the file is closed with the CLOSE statement.

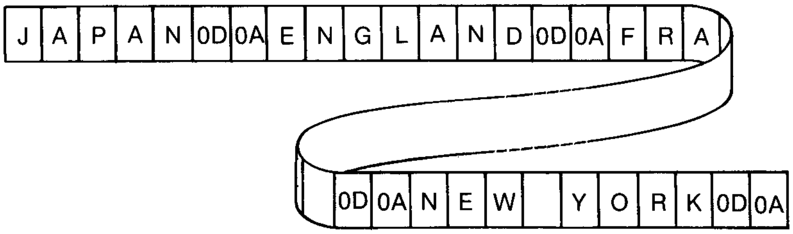
CLOSE [#] [file number]

When the CLOSE statement is executed, the file number is no longer assigned to that particular file, and the same number can be used to open a different file.

Let's write a program that writes data in a sequential file.

```
10 DIM A$(1,3)
20 OPEN "A:DATA.DAT" FOR OUTPUT AS #1
30 FOR L=0 TO 1
40 FOR M=0 TO 3
50 READ A$(L,M)
60 PRINT #1,A$(L,M)
70 NEXT M
80 NEXT L
90 CLOSE #1
100 END
110 DATA JAPAN,ENGLAND,FRANCE,U.S.A.
120 DATA TOKYO,LONDON,PARIS,NEW YORK
```

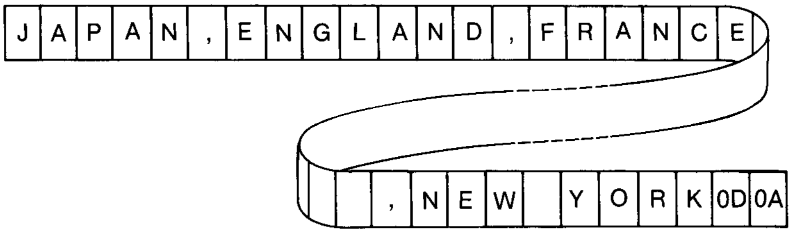
When this program is executed a data file named "DATA.DAT" is created on the disk in drive A. The character type data "JAPAN" is first written in the file, followed by an 0D, 0A pair, and then ENGLAND and the other country names and city names are written in sequence, each separated by an 0D, 0A pair.


J A P A N 0D 0A E N G L A N D 0D 0A F R A
0D 0A N E W Y O R K 0D 0A

If line 60 is changed to

```
PRINT #1,A$(L,M);",";
```

the data will be written as follows:



READING DATA FROM A SEQUENTIAL FILE OPEN FOR INPUT

The procedure for inputting data from a sequential file is:
(1) Open the file with the OPEN statement.
(2) Read the data from the file with the INPUT# statement or the LINE INPUT# statement.
(3) Close the file with the CLOSE statement.

The format for the OPEN statement to input data is:

OPEN "[device name] [file name [.type name]]" FOR INPUT AS [#] file number

The OPEN statement opens a file to input data from it. Unless specified otherwise with the MAXFILES statement (see page 243, only the number "1" can be used as the file number.
After the file is opened, the INPUT# statement is used to read the data.

INPUT[#] file number, variable

The INPUT# statement assigns one piece of data to the variable. The following table shows how data are read by the INPUT# statement.

	Numeric type data	Character type data
spaces, return code, and line code in front of data	ignored	ignored
things which separate data or when the data is separated	space comma return code line feed code	comma return code line feed code when 255 characters have been input
when data is enclosed in " "	—	data within " " are read as a set of data

The LINE INPUT# statement is used only for reading character type data, and inputs only the return code as a data separator.
The file is closed with the CLOSE statement after the data has been input, and the file number no longer has any connection with the file.

Now let's write a program which will read and display on the screen data from the "DATA.DAT" file we made previously.

```
10 DIM A$(1,3)
20 OPEN "A:DATA.DAT" FOR INPUT AS #1
30 FOR L=0 TO 1
40 FOR M=0 TO 3
50 INPUT #1,A$(L,M)
60 NEXT M
70 NEXT L
80 CLOSE #1
90 FOR M=0 TO 3
100 PRINT A$(0,M),A$(1,M)
110 NEXT M
```

The INPUT# statement in line 50 reads data in sequence to the array variable A$. Lines 90 to 110 display the data on the screen.

Let's see what would happen if we attempt to read data from the "DATA.DAT" file using the following program.

```
10 OPEN "A:DATA.DAT" FOR INPUT AS #1
20 INPUT #1,A$
30 PRINT A$
40 GOTO 20
```

When this program is executed, JAPAN, ENGLAND... and the rest of the data is assigned and displayed on the screen. But even after the last piece of data, NEW YORK, is read, the program still tries to input more data. In a case like this, when data input is attempted after the file has ended, the

Input past end

error message is displayed. The EOF function can be used to prevent this from happening.

EOF (file number)

```
10 OPEN "A:DATA.DAT" FOR INPUT AS #1
15 IF EOF(1)=-1 THEN GOTO 50
20 INPUT #1,A$
30 PRINT A$
40 GOTO 15
50 CLOSE #1
```

The EOF function returns −1 when the last data in a file has been read. In the above program this function is used each time data is read to check if there is any more data in the file.

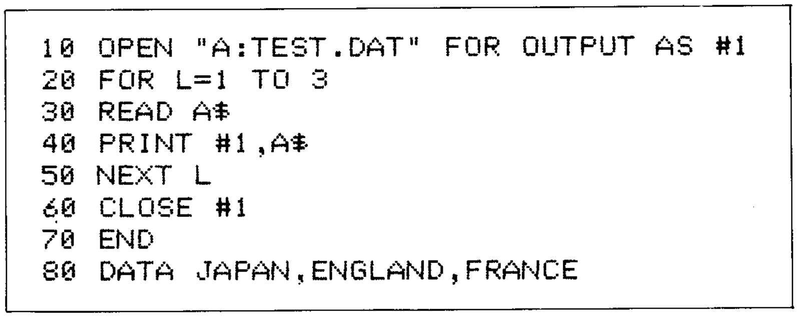
ADDING DATA OPEN FOR APPEND

Data can be added to a data file made previously only when you are operating a sequential file on a floppydisk or the memory disk. To add data, the file is first opened with the OPEN statement.

OPEN "[device name] [file name [.type name]]" FOR APPEND AS [#] file number

The following three programs write, read, and add data on a floppydisk.

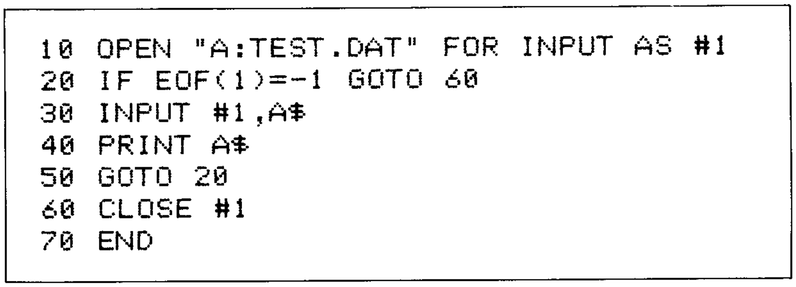
Write Data



```
10 OPEN "A:TEST.DAT" FOR OUTPUT AS #1
20 FOR L=1 TO 3
30 READ A$
40 PRINT #1,A$
50 NEXT L
60 CLOSE #1
70 END
80 DATA JAPAN,ENGLAND,FRANCE
```

This program creates a file named "TEST.DAT" on the disk in drive A, and writes the three pieces of data, JAPAN, ENGLAND, FRANCE, in it.

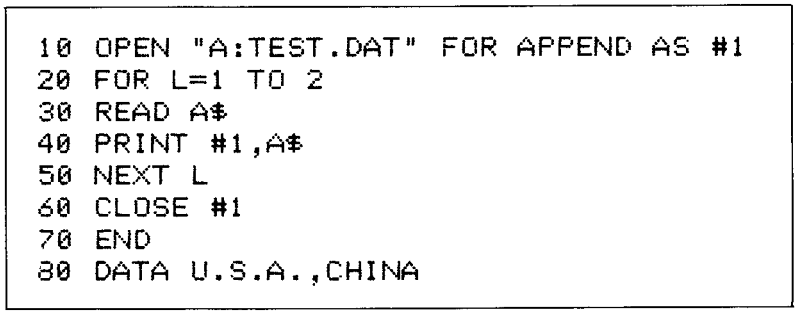
Read Data



```
10 OPEN "A:TEST.DAT" FOR INPUT AS #1
20 IF EOF(1)=-1 GOTO 60
30 INPUT #1,A$
40 PRINT A$
50 GOTO 20
60 CLOSE #1
70 END
```

The data written in the write program is read, assigned to the variable A$, and displayed on the screen.

Add Data



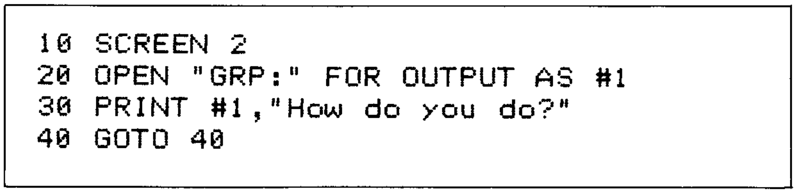
```
10 OPEN "A:TEST.DAT" FOR APPEND AS #1
20 FOR L=1 TO 2
30 READ A$
40 PRINT #1,A$
50 NEXT L
60 CLOSE #1
70 END
80 DATA U.S.A.,CHINA
```

The previous write program has already written three pieces of data in the "TEST.DAT" file opened in line 10 of this program. But since FOR APPEND is specified in the OPEN statement, the data written by the PRINT# statement in line 40 is added after the first three pieces of data. Therefore, after this program is executed the "TEST.DAT" file will contain five pieces of data—JAPAN, ENGLAND, FRANCE, U.S.A., CHINA.

If FOR OUTPUT was used instead of FOR APPEND, the added data would be written at the beginning of the file and the first three pieces of data would be erased.

WRITING CHARACTERS ON A GRAPHIC SCREEN

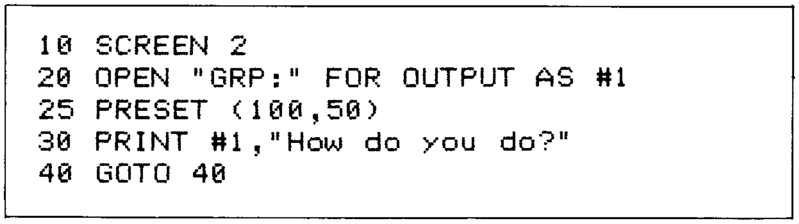
Characters cannot be displayed on graphic mode screens using the PRINT statement. To display characters on a graphic screen, the screen is treated as a file device and the characters to be displayed are output to it as a sequential file.



```
10 SCREEN 2
20 OPEN "GRP:" FOR OUTPUT AS #1
30 PRINT #1,"How do you do?"
40 GOTO 40
```

When the above program is executed the screen changes to the graphic mode and "How do you do?" is displayed.

To specify the display location, a graphic command is executed immediately before the PRINT# statement. The location specified by this command becomes the top left corner of the 8×6 dot matrix of the first character in the PRINT# character string.



```
10 SCREEN 2
20 OPEN "GRP:" FOR OUTPUT AS #1
25 PRESET (100,50)
30 PRINT #1,"How do you do?"
40 GOTO 40
```

The location (100,50) used in the PRESET command in line 25 in this program becomes the top left corner location of the character string output by line 30.

THE NUMBER OF FILES WHICH CAN BE OPENED AT ONE TIME MAXFILES

Only the number "1" can be specified as a file number in MSX2-BASIC in its initialized state. This means that in a program only one file can be opened at any given time. If you want to open more than one file at the same time, you must specify the number of files beforehand using.

MAXFILES = expression

For example, if you specify

```
MAXFILES=5
```

then a maximum of 6 files from 0 to 5 can be opened simultaneously. The number of files which can be specified are from 0 to 15 (the number is from 0 to 6 when a disk is used). File number 0 is reserved for use by the CSAVE, CLOAD, CLOAD?, LOAD, and SAVE commands. Consequently, if

```
MAXFILES=0
```

is executed, only the CSAVE, CLOAD, CLOAD?, LOAD, and SAVE commands can be used.

The MAXFILES statement should be executed either at the beginning of a program or in the direct mode.

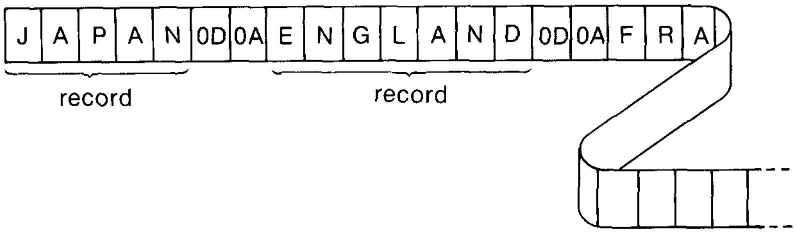
RANDOM ACCESS FILE OPERATION

- ●What is Random Access File?
- ●Writing Data in a Random Access File
- ●Reading Data from a random Access File

WHAT IS A RANDOM ACCESS FILE?

Comparison with a sequential file
A random access file can only be operated on a floppydisk. The following diagrams show how data is written in a sequential file and in a random access file.

Sequential File



Random Access File

record 1	J	A	P	A	N	.						T	O	K	Y	O						
record 2	E	N	G	L	A	N	D					L	O	N	D	O	N					
record 3	F	R	A	N	C	E						P	A	R	I	S						
record 4																						

22 bytes

As the above sequential file diagram shows, a sequential file is like data written in order on a roll of paper tape. A record is the unit for writing in the file one time, and the length of a record depends on the amount of data.

On the other hand, a random access file is like several pieces of paper tape that have been cut the same length; each one of them is a separate record with its own number, such as 1, 2, 3, etc. In the above random access file diagram, the paper tapes have been cut to a size that will hold 22 characters.

Since the records in a random access file are like these separate pieces of paper tape, it is possible to select one record only and write data in it, or read data from it. For example, if we select record 3, we can take out the data "FRANCE PARIS" from it. Or we can specify record 10 and jump over to it to write new data.

A final point which should be mentioned is that even the places where data are written in each of the records of a random access file are handled in an orderly way.

Commands and Statements Used to Operate a Random Access File

The following statements are used to make input and output to/from a random access file.

OPEN	opens a file
FIELD	specifies a record format
LSET, RSET	writes data in a record
PUT	outputs one record to a file
GET	inputs one record from a file
CLOSE	closes a file

WRITING DATA IN A RANDOM ACCESS FILE

The procedure for writing data in a random access file is as follows.
(1) Open the file with the OPEN statement.
(2) Specify the format of one record with the FIELD statement.
(3) Write data in a record with the LSET, RSET statement.
(4) Output the data with the PUT statement.
(5) Close the file.

Each step in this procedure is explained in detail below.

(1) Open the file with the OPEN statement
A file is opened with the OPEN statement in order to write or read data. The format of the OPEN statement is:

```
OPEN "[device name] file name [.type name]" AS[#] file
number [LEN = record length]
```

The device name is always the disk drive name, since a random access file can only be operated with a disk.

When the OPEN statement is executed, a file with the specified file name and type name is ready for data output on the specified disk drive. Just as with a sequential file, file numbers from 0 to 6 can be specified, but in the initialized state only the number "1" can be specified.
Unlike a sequential file, in a random access file data can be written to or read from any part of the file. The smallest unit that can be written or read is called a record. The size of a record in byte units is specified with "record length" in the OPEN statement. Any size from 1 to 256 bytes can be specified. If the size specification is omitted, 256 bytes are automatically specified.

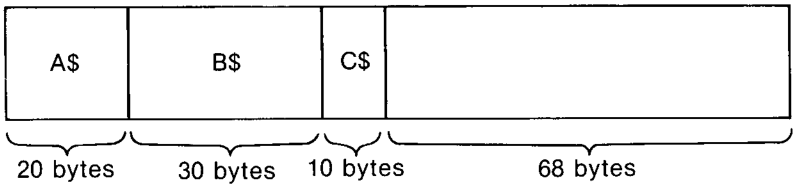
(2) Specify the format of one record with the FIELD statement

```
FIELD[#] file number, character length AS string variable
[, character length AS string variable]...
```

The FIELD statement specifies the variables used for writing and reading data, and assigns the character length of each variable in a record.
All data processed by a random access file is string type data.

```
FIELD #1,20 AS A$,30 AS B$,10 AS C$
```

In the above example, a record is divided in file #1 for input/output, and 20 bytes are assigned to the string variable A$, 30 bytes to B$, and 10 bytes to C$. The total length of all the string variables is 60 bytes, so the length of the record must be previously specified as 60 bytes or more in the prior OPEN statement. If the length has, for example, been specified as 128 bytes, the FIELD statement divides the record as follows:



In this example, no data are input/output for the remaining 68 bytes, so this results in wasted space on the disk. It would be preferable to specify only 60 bytes as the record length in the OPEN statement.

(3) Write data in a record with the LSET, RSET statements
After a record has been divided and assigned variables by the FIELD statement the output data can be set in the record. The LSET statement is used to do this with left justification of the data in the record, and the RSET statement is used for right justification.

LSET string variable = string data
RSET string variable = string data

The string variable in these statements are the variables previously assigned to the record by the FIELD statement. The string data are the data to be written in the file using the string variables.
If

```
LSET A$=X$
```

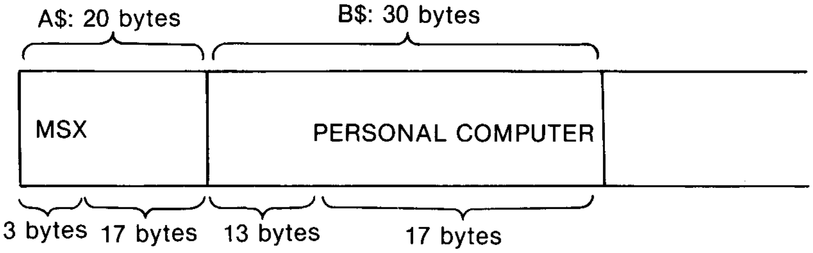
is specified, the string data X$ will be set in variable A$ in the record with left justification.
If

```
RSET B$=Y$
```

is specified, the string data Y$ will be set in variable B$ in the record with right justification.
If

```
X$="MSX"
Y$="PERSONAL COMPUTER"
```

then the data will be set in the record in the following form:
(Note: At this point the record is written in the buffer. It will be written in the file on the disk later by the PUT statement. This is an example of the orderly way in which data are handled in a random access file.)



If the data length is longer than the length of X variables in the record, the excess characters on the right side of the data will be ignored.

Character data can now be written in the file. But since only string variables can be used in records, numeric type data must be converted to string type data before it can be set in a record. The functions MKI$ (make integer dollar), MKS$ (make single precision dollar), MKD$ (make double precision dollar) are used in LSET, RSET statements to convert numeric data into string data.

LSET (or RSET) string variable = MKI$ (integer type data)
LSET (or RSET) string variable = MKS$ (single precision type data)
LSET (or RSET) string variable = MKD$ (double precision type data)

```
LSET P$ = MKI$(A%)
LSET Q$ = MKS$(B!)
LSET R$ = MKD$(C#)
```

The length of the numeric data after they are changed into string type data is 2 bytes for integer type data, 4 bytes for single precision, and 8 bytes for double precision. The number of bytes required for each variable must be defined in the FIELD statement in accordance with the size of the numeric data type.

In the above example, P$ is integer type, Q$ is single precision, and R$ is double precision. These variables would be defined in the FIELD statement as follows:

```
FIELD #1,2 AS P$,4 AS Q$,8 AS R$
```

(4) Output the data with the PUT statement

Once the data has been set in the record in the buffer, the PUT statement is used to write the data in the file on the disk.

PUT[#] file number [,record number]

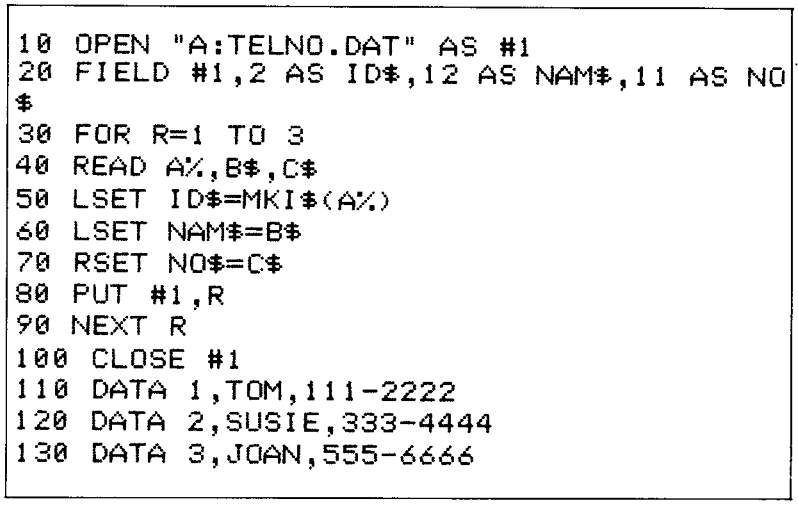
The PUT statement writes the data currently set in the record to the specified record location in the file specified by the file number. This record number is used when the data is read from the random access file at a later time.

If the record number is omitted in the PUT statement, and if no PUT statement or GET statement has been previously executed, the number "1" is automatically specified as the record number. If a PUT or GET statement has been previously executed, 1 is added to the record number of the previous statement to become the new record number.

(5) Close the file

The file is closed with
CLOSE[#] [file number]
which removes the number assigned to the file.

The following program is a complete program for writing data in a random access file.



```
10 OPEN "A:TELNO.DAT" AS #1
20 FIELD #1,2 AS ID$,12 AS NAM$,11 AS NO
$
30 FOR R=1 TO 3
40 READ A%,B$,C$
50 LSET ID$=MKI$(A%)
60 LSET NAM$=B$
70 RSET NO$=C$
80 PUT #1,R
90 NEXT R
100 CLOSE #1
110 DATA 1,TOM,111-2222
120 DATA 2,SUSIE,333-4444
130 DATA 3,JOAN,555-6666
```

A% in lines 40 and 50 is an integer type numeric variable. (The type declaration characters %, !, # are used to define variables as integer, single precision, or double precision type variables.)

The file "TELNO.DAT" is created on the disk in drive A when the above program is executed. The total length of 1 record for input/output is 25 bytes, with 2 bytes assigned to the string variable ID$, 12 bytes assigned to NAM$, and 11 bytes assigned to NO$. When the FOR—NEXT loop beginning in line 30 is repeated three times, the following data are written in the file.

	ID$: 2 bytes	NAM$: 12 bytes	NO$: 11 bytes
record 1	(1)	TOM	111-2222
record 2	(2)	SUSIE	333-4444
record 3	(3)	JOAN	555-6666

The integer type data 1, 2, 3 in this column are converted to string data based on the internal expression format, and written as 2 bytes each.

READING DATA FROM A RANDOM ACCESS FILE

The procedures for writing data and reading data in a random access file are almost the same.
(1) Open the file with the OPEN statement.
(2) Specify the format of one record with the FIELD statement.
(3) Read data from a record with the GET statement.
(4) Close the file.

Steps (1) and (2) are the same as for writing data in a file. The data are read by the (3) GET statement.

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

The GET statement reads the data in the specified record number and assigns them to each variable as defined in the FIELD statement. The numeric data were converted to string data when they were written in the file, so when they are read they are assigned to the string variables specified by the FIELD statement. It is necessary to convert this string data back to numeric data in order to display it on the screen or to process it. This is done with the CVI, CVS, CVD functions, which have exactly the opposite functions as MKI$, MKS$, and MKD$. If, for example, integer type numeric data have been assigned by the GET statement to the string variable P$, then they are changed to integer type data and assigned to the integer type variable A% by

```
A%=CVI(P$)
```

CVI(P$) can be used directly in a PRINT statement to display the data,

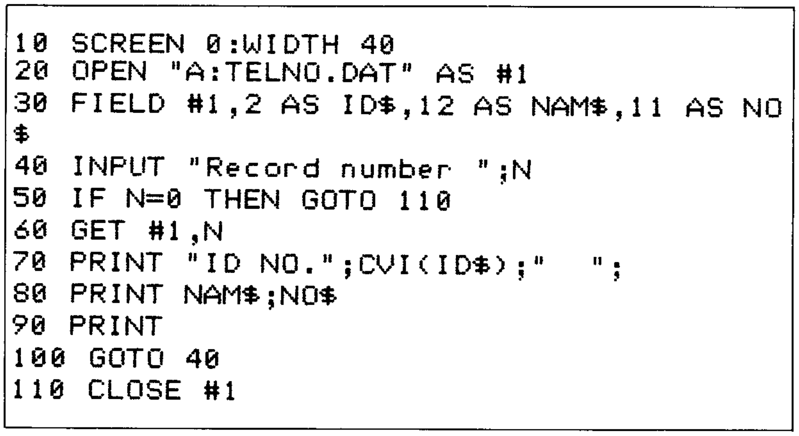
```
PRINT CVI(P$)
```

CVI (convert to integer) converts data to integer type data; CVS (convert to single precision) converts data to single precision type data; and CVD (convert to double precision) converts data to double precision type data.

numeric type variable (integer type) = CVI (string type data)
numeric type variable (single precision type) = CVS (string type data)
numeric type variable (double precision type) = CVD (string type data)

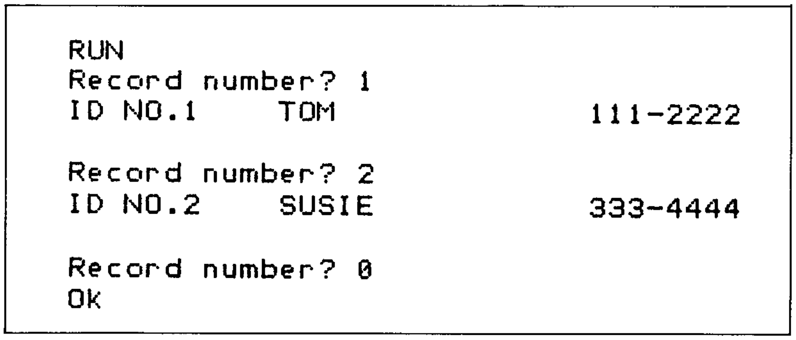
Once the data have been read, the file is closed with the CLOSE statement.

Now, let's write a program to read the data from the random access file we made in the previous program and display it on the screen.



```
10 SCREEN 0:WIDTH 40
20 OPEN "A:TELNO.DAT" AS #1
30 FIELD #1,2 AS ID$,12 AS NAM$,11 AS NO
$
40 INPUT "Record number ";N
50 IF N=0 THEN GOTO 110
60 GET #1,N
70 PRINT "ID NO.";CVI(ID$);"   ";
80 PRINT NAM$;NO$
90 PRINT
100 GOTO 40
110 CLOSE #1
```

This program would display data similar to that shown in the following illustration:



```
RUN
Record number? 1
ID NO.1    TOM                111-2222

Record number? 2
ID NO.2    SUSIE              333-4444

Record number? 0
Ok
```

Chapter 10 Machine Language Subroutines

WRITING AND EXECUTING MACHINE LANGUAGE SUBROUTINES

- ●Specifying the Area and Start Address—CLEAR, DEFUSR
- ●Writing a Machine Language Subroutine—POKE
- ●Calling a Machine Language Subroutine—USR
- ●Saving and Loading a Machine Language Subroutine—BSAVE, BLOAD

MACHINE LANGUAGE SUBROUTINES

With MSX-BASIC, the Z-80 CPU machine language can be used to write a subroutine in memory. Control can be transferred from BASIC to the subroutine, and the results of executing the subroutine returned to a variable defined by BASIC.

A maximum of 10 machine language subroutines can be defined. Also, only one value can be given to one machine language subroutine.

SPECIFYING THE AREA AND START ADDRESS OF A SUBROUTINE CLEAR, DEFUSR

To use a machine language subroutine, **the CLEAR statement should be used to specify the high-memory address of the BASIC program area**. The area following the high-memory address will be used in writing the machine language subroutine.

CLEAR [size of character area] [,high-memory address]

For example,

```
CLEAR 300,&HCFFF
```

specifies the high-memory address of the BASIC program area as &HCFFF (decimal address 53247). Therefore, the area beginning with the &HD000 address (decimal address 53248) can be used as the area for writing a machine language subroutine. In the above CLEAR statement the character area size is specified as 300 bytes. The initial default setting is 400 bytes.

Next, the subroutine start address is defined by the DEFUSR statement.

DEFUSR[X] = start address

X is an integer from 0 to 9. The start address of 10 subroutines can be defined by the USR function.

```
DEFUSR0=&HD000
```

This DEFUSR statement defines the subroutine starting at the address &HD000 (decimal address 53248) as the USR 0 function.

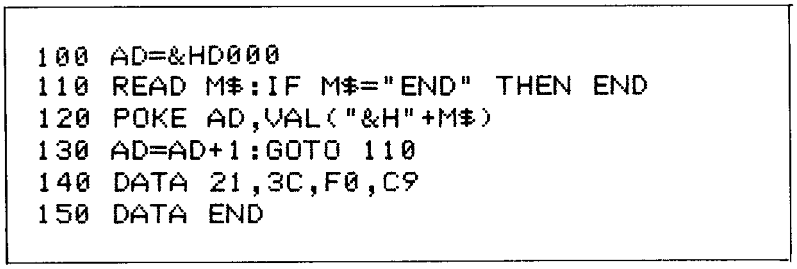
WRITING A MACHINE LANGUAGE SUBROUTINE

POKE

The POKE statement is used to write a machine language subroutine to memory.

POKE address, expression

The POKE statement writes one byte of data to the specified address in memory.
The following program writes the hexadecimal numbers 21, 3C, F0, C9 in memory beginning at the &HD000 address.



```
100 AD=&HD000
110 READ M$:IF M$="END" THEN END
120 POKE AD,VAL("&H"+M$)
130 AD=AD+1:GOTO 110
140 DATA 21,3C,F0,C9
150 DATA END
```

All that is required is to write the machine language subroutine commands (Z-80 commands) in the DATA statement in line 140. The RET command returns control from the machine language subroutine to the BASIC program.

CALLING A MACHINE LANGUAGE SUBROUTINE USR

Transferring control to a machine language subroutine is called "calling a machine language subroutine".

The USR function is used to call a machine language subroutine.

USR[X](I)

X is the USR function number defined in the DEFUSR statement. I is the value (numeric or string) given to the machine language subroutine.
For example, the subroutine defined by DEFUSR 0 is called by executing

```
X=USR0(Y)
```

After the machine language subroutine is executed, the execution result value is assigned to the variable X and the BASIC program is executed.
When the machine language subroutine is called, the value given to the subroutine (the Y variable value) is stored in the following location in memory, and data which indicates the type of Y are entered in register A. The start address for the area which stores the Y value is entered in registers HL.

Y type	Data entered in register A*	Registers HL address indication	Y value storage address
integer	2	&HF7F6	&HF7F8—&HF7F9
single precision	4		&HF7F6—&HF7F9
double precision	8		&HF7F6—&HF7FD

* the same data are also entered in the &HF663 address in memory

When Y is a string variable:

Data entered in register A	Data entered in registers DE	String descriptor
3	String descriptor start address	1st byte: character string length 2nd & 3rd bytes: start address of the character string storage area

When the machine language subroutine execution is completed, the result value is returned to the X variable by setting the registers and memory as follows:

Result value type	&HF663 memory address	Registers DE	Registers HL	Result storage address
integer	2		&HF7F6	&HF7F8—&HF7F9
single precision	4		&HF7F6	&HF7F6—&HF7F9
double precision	8		&HF7F6	&HF7F6—&HF7FD
string	3	string descriptor start address		area starting from address indicator by the 2nd and 3rd string descriptor bytes

The generated number to be returned to BASIC is stored in the &HF7F8 and &HF7F9 address. Also, since the value type is integer, 2 is entered in the address &HF663.

The following BASIC program will write the machine language subroutine to memory, call it and use the returned values (random value from 0 to 255) to display 1 to 6 on the dice.

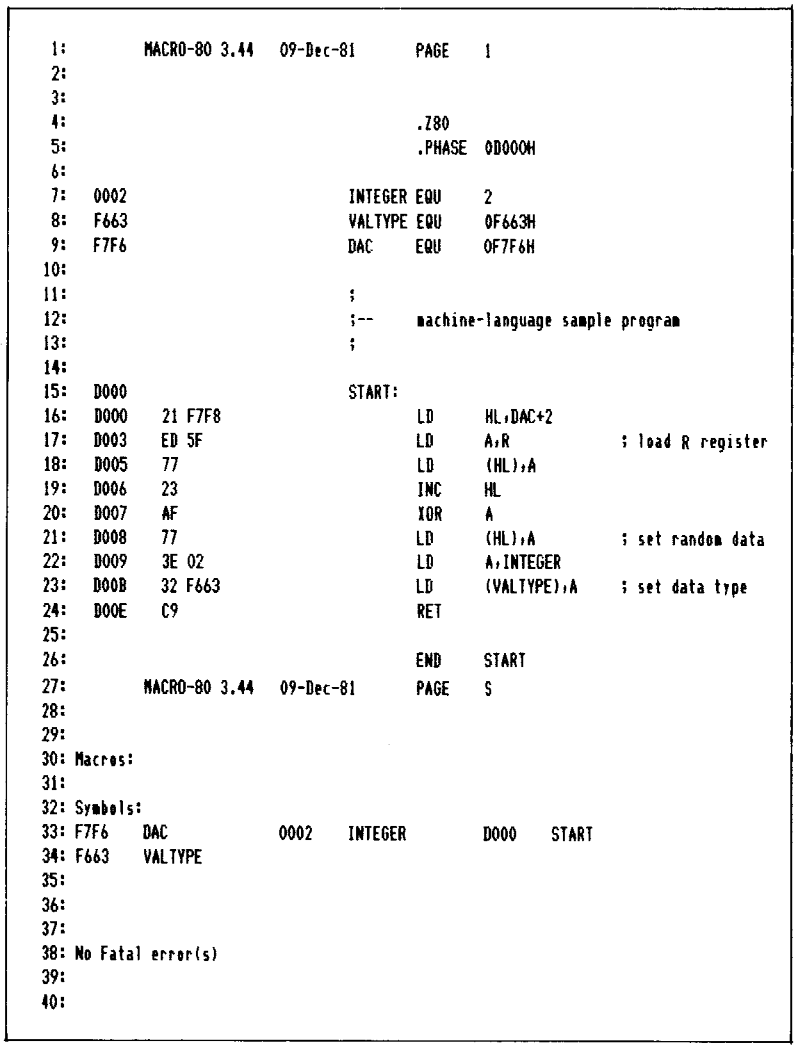
```
10 CLEAR 300,&HCFFF ——specifies the high-memory address
20 SET BEEP 1          of the BASIC program area
30 '--
40 AD=&HD000:DEFUSR0=AD ————sets the subroutine start
50 GOSUB 430                    address at &HD000
60 ' --
70 SCREEN 5
80 OPEN "GRP:" FOR OUTPUT AS #1
90 SET PAGE 1,1:CLS
100 R=5:RESTORE 360
110 FOR L=0 TO 5
120 XC=(L MOD 3)*80                               draws the
130 YC=(L \ 3)*100                                spots of the
140 LINE (XC,YC)-STEP(50,50),15,BF                dice (1 to 6)
150 FOR M=0 TO L                                  on page 1
160 READ X,Y
170 CIRCLE (XC+X,YC+Y),R,8
180 PAINT (XC+X,YC+Y),8,8
190 NEXT M
200 NEXT L
210 ' --
220 SET PAGE 0,0
230 PRESET (70,150)
240 PRINT #1,"Press any key"     calls the subroutine
250 IF INKEY$="" THEN 250 ————when a key is pressed
260 J=USR0(0) MOD 20 ——————subroutine call
270 FOR L=0 TO J
280 N=USR0(0) MOD 6 ———————subroutine call
290 X=(N MOD 3)*80                                 copies the
300 Y=(N \ 3)*100                                  spots of the
310 COPY (X,Y)-STEP(60,60),1 TO (100,70)           dice (1 to 6)
,0                                                 to page 1 in
320 BEEP:FOR W=0 TO 50:NEXT W                      accordance
330 NEXT L                                         with the value
340 GOTO 250                                       returned by
350 ' *** dice data ***                            the subroutine
360 DATA 25,25
370 DATA 25,10,25,40                               data which
380 DATA 10,10,25,25,40,40                         determines
390 DATA 10,10,10,40,40,10,40,40                   the location
400 DATA 10,10,10,40,40,10,40,40,25,25             of the spots
410 DATA 10,10,10,40,40,10,40,40,10,25,4           of the dice
0,25
```

```
420 ' *** write machine language subrout
ine ***
430 RESTORE 480
440 READ M$:IF M$="END" THEN RETURN
450 POKE AD,VAL("&H"+M$)
460 AD=AD+1:GOTO 440
470 ' *** machine codes ***
480 DATA 21,F8,F7,ED,5F,77,23,AF
490 DATA 77,3E,02,32,63,F6,C9
500 DATA END
```

writes the machine language subroutine

Sample Machine Language Subroutine

The following machine language program generates random numbers from 0 to 255 using the CPU's register R (refresh register). The source list is given below. (The format is based on MACRO80 3.44).



```
        MACRO-80 3.44   09-Dec-81       PAGE    1


                                        .Z80
                                        .PHASE  0D000H

  0002                          INTEGER EQU     2
  F663                          VALTYPE EQU     0F663H
  F7F6                          DAC     EQU     0F7F6H

                                ;
                                ;--     machine-language sample program
                                ;

  D000                          START:
  D000    21 F7F8                       LD      HL,DAC+2
  D003    ED 5F                         LD      A,R             ; load R register
  D005    77                            LD      (HL),A
  D006    23                            INC     HL
  D007    AF                            XOR     A
  D008    77                            LD      (HL),A          ; set random data
  D009    3E 02                         LD      A,INTEGER
  D00B    32 F663                       LD      (VALTYPE),A     ; set data type
  D00E    C9                            RET

                                        END     START
        MACRO-80 3.44   09-Dec-81       PAGE    S


Macros:

Symbols:
F7F6    DAC             0002    INTEGER         D000    START
F663    VALTYPE




No Fatal error(s)


```

SAVING A MACHINE LANGUAGE SUBROUTINE
BSAVE

The machine language subroutine in the above program is written in the &HD000—&HD00E address in memory.
The BSAVE command is used to save this machine language subroutine on cassette tape or on a disk.

BSAVE "[device name] [file name [.type name]]", start address, end address [,execution start address]

The BSAVE command saves the content of the specified area in memory.

```
BSAVE "CAS:RANDOM",&HD000,&HD00E
```

saves the content of the &HD000—&HD00E address (the machine language program) on cassette tape with the file name RANDOM.

```
BSAVE "A:RANDOM.BIN",&HD000,&HD00E
```

saves the same content on the disk in disk drive A with the file name/type name RANDOM.BIN.

LOADING A MACHINE LANGUAGE SUBROUTINE BLOAD

The content saved with the BSAVE command is loaded with the BLOAD command.

BLOAD "[device name] [file name [.type name]]" [,R] [,offset]

```
BLOAD "CAS:RANDOM"
```

loads the content of the file saved as RANDOM on cassette tape.

```
BLOAD "A:RANDOM.BIN"
```

loads the content of the file saved with the file name/type name RANDOM.BIN on the disk in drive A.

In both cases, the content of the area beginning at the start address location specified in the BSAVE command is loaded.

When a machine language subroutine is saved separately in the above manner, it can be loaded and executed in a BASIC program without writing the subroutine with the POKE statement. In this case, line 50 and lines 420 to 500 would not be required in the above BASIC program.

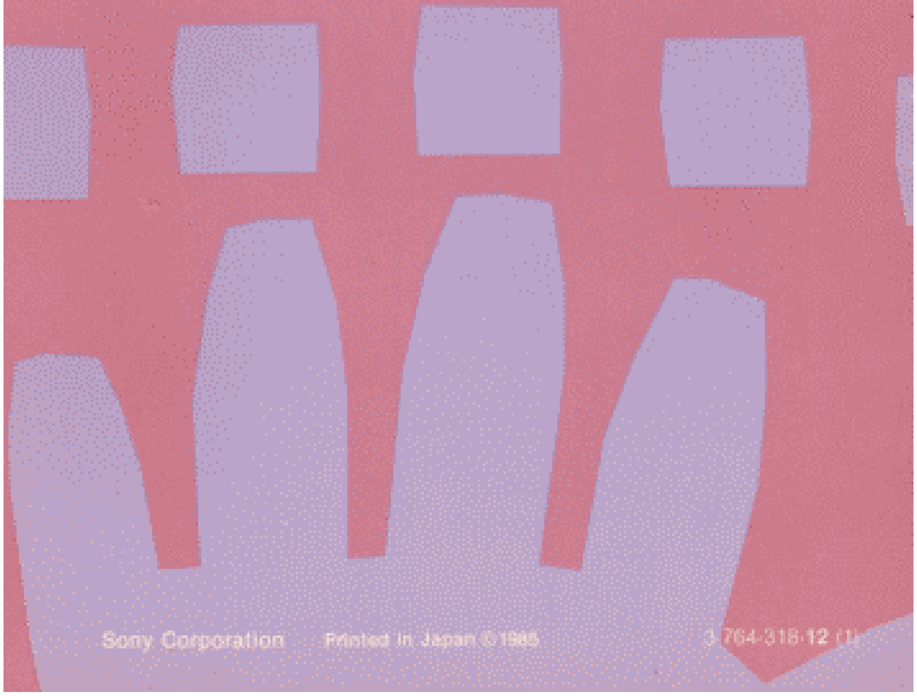
INDEX

BASIC COMMANDS, STATEMENTS, FUNCTIONS, AND ERROR MESSAGES

TERMS

USING BASIC

A GUIDE TO MSX-BASIC Version 2.0



Sony Corporation Printed in Japan ©1985 3-764-318-12 (1)