MSX

Melbourne House

Toshiyuki Sato
Paul Mapstone
Isabella Muriel

· THE · COMPLETE · MSX ·

Programmers Guide

· THE · COMPLETE · MSX ·

Programmers Guide

OTHER MSX COMPUTER TITLES

MSX Games Book (Lacey)
MSX Exposed (Pritchard)
MSX Machine Language for the Absolute Beginner (Pritchard)
Ultra High-Performance MSX Programs (Sato, Muriel, Mapstone)
Z80 Reference Guide (Tully)

· THE · COMPLETE · MSX ·

Programmers Guide

Toshiyuki Sato
Paul Mapstone
Isabella Muriel

Melbourne House Publishers

IN THE UNITED KINGDOM —
Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

IN THE UNITED STATES OF AMERICA —
Melbourne House Software Inc.
347 Reedwood Drive
Nashville TN 37217

IN AUSTRALIA —
Melbourne House (Australia) Pty Ltd
70 Park Street
South Melbourne, Victoria 3205

Cataloguing in Publication

Sato, Toshiyuki.
Complete MSX Programmer's Guide.
Includes index.
ISBN 0 86161 173 X.
[1]. MSX Basic (Computer program language).
2. Microcomputers — Programming. I. Muriel, Isabella.
II. Mapstone, Paul. III. Title.

001.64'24

Edition 7 6 5 4 3 2 1
Printing F E D C B A 9 8 7 6 5 4 3 2 1
Year 90 89 88 87 86 85 84

CONTENTS

Section 1:
Introduction to MSX Basic

Section 2:
Advanced Programming Guide

Section 3:

Reference Section

Section 4:

The Operating System 495

SECTION 1

INTRODUCTION TO MSX BASIC

CHAPTER 1

SETTING UP

When you unpack your MSX computer you should find a video connecting lead packed with it. This lead has a PHONO connector on either end and is used to connect the computer to a television.

For the moment, you only need to connect the computer to a television. Any television will do. A colour one is nice as you can then use the MSX colour command to full advantage, but a B/W set is perfectly adequate.

To connect the TV to the computer, insert one of the PHONO plugs into the socket marked "TV" at the back of the computer; the other end of the lead plugs into the aerial connection of the TV. If your TV has two aerial sockets, marked UHF and VHF, use the UHF one (UK).

Switch on both TV and computer at the mains. First turn on your TV at the set and allow it to warm up, then turn on your computer. If the TV is correctly tuned you will first see this screen . . .

```
MSX  system
version 1.0
Copyright 1983 by Microsoft
```

. . . and, after a pause, this screen:

```
MSX BASIC version 1.0
Copyright 1983 by Microsoft
28815 Bytes free
OK
■


 color   auto   goto   list   run
```

If you do not see this second screen, adjust your UHF tuner until you do. (The number of bytes free will depend on how much memory is installed in your machine.) This happens every time the MSX is turned on.

If your TV has a continuously variable tuning control, adjust this until you get a clear picture. If you have a push button for each station, it is a good idea to tune in on an unused one.

Now you are ready to use the keyboard. Experiment as much as you like, you will not damage the computer! Everything you type will apear on the TV screen. You will probably get several Error messages from the computer as well, but do not worry, you will learn how to avoid these later.

If you look at the screen, on first switching on, you will see a small white square directly underneath the "OK" message. This is called the Text Cursor, and shows you where the next character to be printed on the screen will appear. Try typing some letters in. It's best to get into the habit of using both hands when typing. Pressing any of the Character Keys will result in the character appearing on the screen. Type:

The quick brown fox jumps over the lazy dog.

To get spaces, hit the long bar at the bottom of the Keyboard. To get upper case characters you press the <SHIFT> Key and the Character Key you want simultaneously.

On the LHS of the keyboard there is a key marked <CAPS LOCK>. If you press this key, everything you type afterwards will automatically be

printed in upper case, and the green light next to it will light up. To get out of this, press the key again.

Now look at the numbers. See what happens if you press <SHIFT> and each number in turn from 1 to 9.

At the very top of the keyboard there are five elongated Keys with F1 to F10 written on them. These are the Function Keys. The display at the bottom of the screen is due to these keys. Forget about them for the moment, as we will come to their use in CHAPTER 8.

Now find the <STOP> Key. You will use this quite a lot with the Control Key, marked <CTRL>. The <ESC> key does nothing at the moment, but may be used for controlling a printer.

The <TAB> key moves the cursor across the screen. It deletes anything in its way. If the cursor was originally on the LHS, it is moved to the 9th character position. On pressing <TAB> again it is moved to the 17th position, then the 25th.

The <GRAPH> Key enables you to print the Graphics Characters, which are accessed via the Character Keys. On pressing <GRAPH> with any Character Key, you will get one of the Graphics Characters. You will get a different Graphics Character if you now press <SHIFT> <GRAPH> and one of the Character Keys. Find the Graphics Characters ♥, ♦, ♣ and ♠.

On pressing <CODE> and a Character Key, you can get the European and Greek characters, e.g. α, β, $\ddot{a}$, $\ddot{\mu}$ etc. Experiment with this key. Try pressing <SHIFT> as well. Most of the keys will now give you a different CODE Character..

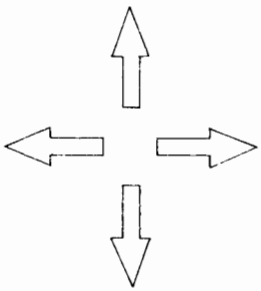
A very important key is the <RETURN> Key. Whenever you want the computer to read something you have written on the screen, you must press <RETURN>. If you press it now, you will probably get an Error message. Do not worry, it just means that the computer has not understood what you have typed in.

By now the screen must be rather full. Notice that when you get to the end of the screen, all the text automatically moves up one line and you lose your top line. This is called "scrolling".

To clear the screen, press <SHIFT> and <HOME> simultaneously. <HOME> is one of the four editing Keys on the top RHS of the keyboard. Notice that after doing this, the cursor automatically jumps back to the top LHS of the screen. If ever the screen gets very messy, you can always clear it in this way. Pressing this key without <SHIFT>, causes the Cursor to Home back to the top LHS of the screen without deleting the text.

Now type in a whole line of characters. To delete these you can press the Back Space Key, <BS>. This will delete anything on the LHS of the cursor. Like any of the other keys, it repeats itself if you keep your finger on it, so by now you have probably deleted the complete line!

Type in a lot more characters and experiment with the Cursor Keys, which are marked



and move the cursor about in the marked direction without deleting what is underneath.

Try to delete a particular character, using <BS> and the Cursor Keys. When you think you have got the idea, see what happens when you press <DEL>, which is another of the four Editing Keys on the RHS.

This key DELetes the character underneath the cursor and moves everything to the right of the Cursor back one space. If you keep your finger on <DEL> you will delete everything which was originally on the RHS of the Cursor.

Supposing you had the following typed in:

abcfg

You might wish to insert "d" between "c" and "f". To do this, move the Cursor until it is on top of the letter "f" and then press the INSert Key, <INS>. The Cursor will be replaced by a white line. If you now type "d", it will appear to the left of the line and be inserted between the "c" and "f". To get the Cursor back, press <INS> again. Practice using the Cursor and Editing Keys.

The fourth Editing Key, <SELECT> does nothing at the moment.

EXERCISES

1 Type out both Graphics Characters associated with each key, and the character itself. Now delete the alphanumeric characters, leaving just the Graphics Characters on the screen.

2 Print the number 0 then the letter O; notice the difference between the two. You must use 0 when typing a number, otherwise the computer will report an Error. Also compare 1 with the lower case letter l. Once again, do not confuse the two.

CHAPTER 2

COMMAND MODE

Once you have become familiar with the keyboard and the use of <BS> and the EDIT keys, you are ready to give the computer commands. First of all clear the screen, by pressing the <SHIFT> and <CLS> keys simultaneously. Do this whenever the screen gets rather full or messy.

Now carefully type in:

PRINT "Welcome to the MSX"

then press the <RETURN> key.

You should see "Welcome to the MSX" displayed directly underneath the line you have just typed in, with OK under this. Perhaps you have an error message instead. Do not worry, have another go, as you have probably made a typing mistake.

On pressing <RETURN>, the computer carries out or executes a command. It only recognises certain words, called Keywords, of which PRINT is one. On seeing PRINT, followed by some words in quotation marks, it knew that it had to PRINT out everything included within the quotation marks.

The computer does not mind if you use upper or lower case letters and ignores spaces. (Do not, however, put spaces between the letters in a KEYWORD, as this will not be accepted by the computer.)

print "Welcome to the MSX" <RETURN>

would produce the same result.

Everything included within quotation marks will be printed exactly as you type it. The line:

"Welcome to the MSX"

is called a "string". When the computer sees a PRINT statement followed by a string, it strips the quotation marks off the string and PRINTs the rest of the string exactly as it stands. More on strings later!

To PRINT a number you do not need the quotation marks. Type in:

PRINT 12345 <RETURN>

You will see 12345 displayed directly below the last previous line on the screen. Here again it does not matter how many spaces you have between the PRINT statement and the number. Spaces within the number to be PRINTed will be ignored.

PRINT 1 2 3 4 5 <RETURN>

gives the same result.

Now if you type in:

PRINT 3+4 <RETURN>

the computer will PRINT out the value 7. It has evaluated the expression 3+4 and given you the answer. If you wish to PRINT out 3+4 as it stands, you must put it in a string by using the quotation marks:

PRINT "3+4" <RETURN>

The computer now sees 3+4 as a string, and so PRINTs it exactly as you wrote it.

To subtract a number, use the "−" sign.

To multiply two numbers, the "*" sign is used, instead of the more usual multiplication sign, "×". This is done to prevent confusion with the letter "×".

To divide, use the "/" sign, instead of "÷".

Type in the following in turn. N.B. Don't forget to press <RETURN> after each line. From now on this will be assumed.

```
PRINT 4*3
PRINT "3+4=";3+4
```

In the last example you will see:

```
3+4= 7
```

When the computer sees a semicolon it realises that there is another item to be PRINTed, in this case the expression 3+4. It therefore evaluates this and PRINTs the answer straight after the string.

You will notice that there is always a blank space PRINTed before and after a positive number.

```
PRINT "aa" ; 3;4;−5 "bb""cc"
```

gives:

```
aa 3  4 −5 bbcc
```

As you can see no spaces are left when PRINTing strings. When PRINTing a negative number, the negative sign is put in to the preceding blank space. When no punctuation is used between strings, the computer assumes a semicolon was intended. You must put punctua-

tion between numbers, otherwise the computer will see this as one long number.

The computer also recognises the comma in a PRINT statement, in which case items are PRINTed on the same line, the second item being PRINTed 15 character positions along from the first item PRINTed.

Type in the next two lines:

```
LET A=3
PRINT A
```

The number 3 is displayed on the screen.

The LET command assigns a value to the variable, which in this case is called A. When the computer receives the PRINT command it is expecting a number, as there are no quotation marks. It therefore checks its memory and finds that a number has been assigned to the variable A and so PRINTs this number.

You can have more complicated expressions in a LET command:

```
LET B=31*72*4
PRINT B
```

The number 8928 is displayed.

Now type:

```
LET B=B+1000
```

You get:

```
9928
```

The above LET statement looks incorrect mathematically, but is perfectly acceptable in computing. The expression on the RHS of an "=" sign is evaluated and put into the variable on the LHS. In this case, the variable B is now equal to its old value, plus 1000.

Variable names must begin with a letter. After that, numbers may be used. Blanks are ignored in variable names so:

A A is equivalent to AA

Upper or lower case letters can be used, but the computer will not distinguish between the two.

ABBACUS is equivalent to abbacus

The computer only reads the first two characters of a variable name, so:

ab is equivalent to abbacus

It is important not to include a KEYWORD in the variable name, so:

MONTH

is invalid as ON is a KEYWORD.

Some words have been reserved, and may be used as KEYWORDs in a later version of the MSX. These words must not be used within variable names, e.g.

```
NAME
```

This is invalid, as "NAME" is a reserved word.

Here are some permitted variable names:

```
number 2
n2
NUM
```

These names would not be allowed:

2n	The first character must be a letter.
num?	"?" is an invalid character in a variable name.

Strings can also be assigned to variables:

```
LET A$="Hello"
PRINT A$
```

Hello is displayed, as PRINT A$ is equivalent to typing PRINT "Hello" in this example.

String Variable names must end with the "$" sign to distinguish them from Numeric Variables. Otherwise they must conform to exactly the same specifications as Numeric Variable names.

Here are some valid String Variable names:

```
YEAR$
INCOME$
L1$
```

These are invalid:

A	No $ sign
A.B$	"." is an invalid character.
1FIRST$	The first character must be a letter.
ON$	ON is a KEYWORD
TONE$	ON is a KEYWORD
NAME$	NAME is a reserved word.

Another useful command is INPUT. Type:

```
INPUT A
```

When the computer sees this it displays a question mark and waits for you to type in a number. This number is then assigned to the variable A. Type in a number. You will see it appear after the question mark. If you now type:

PRINT A

the number you have just typed in will be PRINTed. If you type in a string, you will get an error.

EXERCISE

1. Find the value of the A, where:
   A=49*38.07/13

CHAPTER 3

WRITING A PROGRAM

So far you have been working in Command Mode. You have typed a command on screen, pressed <RETURN> and the computer has executed the command straight away. If you wanted to repeat the command, you have had to type it in again, which is rather time consuming.

If you want the computer to remember a command, you must give it a line number, for example:

```
10 PRINT "THIS IS A TEST"      <RETURN>
```

On pressing <RETURN> the computer sees the line number, in this case 10, so it realises that this line is part of a program. It therefore stores the complete line in memory. When it has done this, it displays the "Ok" prompt on the screen.

To get the computer to execute the line, the command RUN must be used. Type:

```
RUN.                <RETURN>
THIS IS A TEST      is now displayed.
```

To write a program, each command must be preceded by a number. This can be any value up to 65529, but it is a good idea to start with 10 and increase this in steps of 10, i.e. have line numbers 10,20,30,40, etc. This way, when you have finished typing your program, there is room for you to insert new lines between these lines, as you can use line numbers 15, 25, 36 etc. as necessary. The computer executes a program, line by line, starting from the line with the lowest line number.

Before you type in the next program, enter the following:

```
NEW     <RETURN>
```

This command erases any old programs and variables in the computer's memory. It is advisable to use this command before typing in a new program.

Now type in the following program. Do not forget, the <RETURN> key must be pressed after typing each line:

```
10  PRINT "WHAT IS YOUR NAME?"                    <RETURN>
20  INPUT N$                                      <RETURN>
30  PRINT "HELLO ";N$" WELCOME TO MSX"            <RETURN>
```

RUN this program. If you have not made any typing errors you will see a display as shown below:

```
WHAT IS YOUR NAME?
?
```

The INPUT statement, in line 20, is waiting for a string variable to be entered. Type in your name and press <RETURN>. The computer now PRINTs:

```
HELLO name WELCOME TO MSX
```

After you have typed in your name, the computer proceeds to line 30 and gives you the Welcome message. When the computer runs out of program lines to execute, it realises that it has reached the end of the program and finishes off by giving you the "Ok" prompt. You are now back in Command Mode.

If you got an error message instead of the Welcome message, you have made a typing mistake in one of your lines. Before looking for errors, it's a good idea to get a new listing of your program. To do this type:

```
LIST
```

This command will LIST your program out neatly, starting from the lowest line number. All Keywords and Variable Names will be LISTed in upper case, even if you typed them in using lower case. You should see:

```
10  PRINT "WHAT IS YOUR NAME?"
20  INPUT N$
30  PRINT "HELLO ";N$;" WELCOME TO MSX"
```

Now check the line referred to in the error message for typing mistakes. Once you have found the error, edit this line using the Editing and Cursor Keys as described in CHAPTER 1. Then, with the cursor still at the corrected line, press <RETURN>. This tells the computer to replace the old line with the corrected version.

It's a good idea to get the cursor completely out of the program listing before typing RUN again. To do this, keep the < ↓ > key depressed until the cursor is at the bottom of the screen. Now type RUN again, followed

by <RETURN>. Repeat the editing process until your program works!

If the screen gets very messy, press <SHIFT><CLS> before typing LIST.

If you have made lots of mistakes in typing a line, instead of editing with the cursor key, it is sometimes quicker to retype the whole line, e.g. to edit:

```
10  PRONT WHst os yourname"
```

it might be quicker to retype the whole line:

```
10  PRINT "Hello, what is your name"
```

The old line 10 is then deleted and replaced by the new one.

If you wish to remove a line from your program, all you have to do is type in the line number and press <RETURN>. After you have done this you cannot get the line back, so be careful!

```
10      <RETURN>
```

deletes line 10.

Type in the next program:

```
10  REM This program repeats itself
20  CLS
30  INPUT "WHAT IS YOUR NAME"; N$
40  PRINT "HELLO "; N$ ", WELCOME TO MSX"
50  GOTO 30
```

The REM statement tells the computer to ignore everything that follows in that line. The rest of the line is used as a REMinder, for you, as to what the program does. It is a good idea to include REM statements in your program, otherwise it is very easy to forget how a program works when you come back to it later.

The second line Clears the Screen. You have already found that one of the keys does this, <SHIFT> <HOME>, but if you want to incorporate this command into a program you must type in the command CLS.

The command in line 30 should be familiar by now. This example shows you how to PRINT out a message before the "?". All you have to do is put the message in quotation marks, follow the message by the semicolon and then the Variable Name. The order is important here.

Line 50 just tells the computer to go back to line 30, and continue execution from there. This is called an 'infinite loop', as the computer loops back to line 30, every time it reaches line 50. This will continue until you break out of the program.

Now try running this program.

When you get bored with INPUTting your name, press <CTRL> <STOP>. You will get the message:

```
BREAK at LINE 30
```

To continue execution, type in CONT. You will find that execution restarts from where it left off, i.e. at the INPUT statement in line 30.

If you had broken into the program at any other line, i.e. one which did not contain an INPUT statement, you would find that execution starts from the next line.

EXERCISE

1. Write a program which asks for your name and age, and then PRINTs these out.

CHAPTER 4

MORE ON ARITHMETIC

So far you have met the mathematical symbols:

+ − / *

Another useful symbol is " ^ ". This raises a number to the power of another number, i.e.

2^3 is written 2 ∧ 3

Type in:

PRINT 4 ^ 5

This gives the answer 1024.

Brackets can be used in arithmetic expression. Any expression within brackets is executed first.

All the symbols listed above can be combined in one expression. The expression will then be evaluated in the following order:

( )	Expressions within ( ) are evaluated first
^	Exponentials
*,/	Multiplication and division have the same priority
+,−	These are evaluated last

For example, the computer evaluates the following expression in three stages:

	(3+4) * 7 ∧ 2
First stage	7 * 7 ∧ 2
Second stage	7 * 49
Third stage	343

You must include the "*" sign between brackets, i.e.

```
PRINT (3+4)(4-2)        IS WRONG!
```

This line should be:

```
PRINT (3+4)*(4-2)
```

In the following example program a temperature is asked for in fahrenheit and is then converted to centigrade:

```
10  REM temperature conversion
20  CLS
30  INPUT "FAHRENHEIT TEMPERATURE"; F
40  C=(F-32)*5/9
50  PRINT "EQUIVALENT CENTIGRADE TEMPERATURE IS"; C
60  GOTO 30
```

Notice in line 40 that the LET statement has been omitted. This statement is optional. As soon as the computer sees a variable next to an equals sign, it assigns the value of the expression on the RHS of the "=" to the Numeric Variable on the LHS.

When you run this program, you will find that the centigrade temperature is given to 14 significant figures! If you wish to round the temperature off to the nearest integer, the function INT may be used.

INT converts real numbers, i.e. numbers which contain decimal places, to integers, i.e. whole numbers. The argument must be enclosed in brackets.

The INT function rounds a real number down to the nearest integer:

```
 3.4     becomes     3
-2.7     becomes    -3
```

To round the real number to the nearest integer and not to the next lowest, 0.5 must be added:

```
INT(3.9)          evaluates to 3
INT(3.9 + 0.5)    evaluates to 4
```

You must let the computer know when you want it to store integers and when real numbers. To distinguish bettween the two, integer names end with the "%" sign.

Replace lines 40 and 50 of the last program by:

```
40  C%=INT((F-32)*5/9+0.5)
50  PRINT"EQUIVALENT CENTIGRADE TEMPERATURE IS";
    C%
```

The program will work if you use C instead of C%, but the number will still be stored to 14 decimal places, even if all the numbers after the decimal point are zeros. The zeros won't be PRINTed, but they are stored in

memory and so taking up valuable room. It is therefore better to use Integer Variable Names whenever possible.

EXERCISE

1. Convert the temperature program so that it INPUTs the temperature in centigrade and PRINTs it out in fahrenheit. You will need to use the equation:

   F=C*9/5+32

CHAPTER 5

THE USE OF LOOPS

This program PRINTS out any multiplication table you wish. Do not bother typing it in!

```
10  REM Multiplication Table
20  INPUT "Multiplier"; M
30  PRINT "1 times"; M ; "="; 1*M
40  PRINT "2 times"; M ; "="; 2*M
50  PRINT "3 times"; M ; "="; 3*M
```

and so on. This is a rather long and repetitive program. A much more concise way of doing the same calculation is by using a FOR. . .NEXT loop:

```
10  REM Multiplication Table with loop
20  INPUT "Multiplier"; M%
30  FOR C%=1 TO 10
40  PRINT C%;  "times";  M%;  "=";  C%*M%
50  NEXT C%
60  GOTO 20
```

Notice that integer Numerical Variables are used in this program, C% and M%.

The first time the computer reaches line 30, it sets C%=1 and then passes on to line 40. When it reaches line 50, the NEXT statement tells it to return to line 30. Back at line 30, it increments the counter by one, i.e. C%=2 now. This loop is repeated until C%=10, the limiting value; after this the computer ignores the NEXT statement in line 50 and carries on with line 60.

You can increment the counter by STEPs greater than 1 by using the STEP statement in the FOR. . .NEXT Line. Try the following:

```
10  FOR N=0 TO 12 STEP 2
20  PRINT N
30  NEXT
```

You should see:

```
0
2
4
6
8
10
12
```

The constraints on the counter name are the same as those for Numeric Variable Names. See CHAPTER 2.

If you replace the limit in the above program by 13 it will have no effect on the results. The computer will still exit from the loop when the counter reaches 12. With a STEP size of two and starting from zero, the counter will never reach 13.

You can use real numbers as well as integers to set the initial and limiting values of the counter and the step size. Negative STEPs can also be used:

```
10  FOR N=10 TO 2.5 STEP -2.5
20  PRINT N
30  NEXT N
```

gives:

```
10
7.5
5
2.5
```

Do be careful to start the counter at a higher value than the limiting value when using negative STEPs, otherwise an Error Statement will occur.

It is possible to put a FOR. . .NEXT loop inside another FOR . . .NEXT loop. This is then called a nested loop. We will use this technique in the next example to PRINT out a pattern of "*"s:

```
10 REM a nested loop
20 PRINT"*"
30 FOR I=1 TO 9
40 FOR J=1 TO I          ) THE
50 PRINT "*";            ) NESTED
60 NEXT J                ) LOOP
70 PRINT"*"
80 NEXT I
```

You should see:

```
*
**
***
****
*****
******
*******
*********
**********
```

Note the ";" at the end of the PRINT statement in line 50. This causes the next PRINT statement to PRINT on the same line, starting at the first empty character position, i.e. there is no carriage return.

Line 30 puts I=1. This sets the limit for the J counter in the nested loop, so the first time round the I loop the "*" is only PRINTed once. The next time round the main loop, I is incremented to two. J can now take the values 1 and 2 so this loop is executed twice, resulting in two "*"s being PRINTed on the same line in the loop.

Different patterns can be obtained by varying the STEP size.

EXERCISES

1. See what happens if you omit lines 20 and 70.
2. Try the previous program with STEP 4 in the main loop. Change the upper limit of this loop. With a different step size, lines 20 and 70 will need changing to keep a regular pattern.
3. Vary the STEP size in the nested loop.

CHAPTER 6

THE USE OF CONDITIONS

Sometimes you will find that you want your program to take a certain path depending on whether a certain condition holds. The IF. . .THEN format is used for this. Here is an example of its use:

```
20  IF A=B THEN GOTO 100
30  PRINT A,B
```

Line 20 tells the computer that IF the contents of variable A are equal to the contents of variable B, THEN GOTO line 100, otherwise continue with the next line of the program, which in this case PRINTs the variables A and B.

Various expressions can follow the IF statement using the signs:

>	GREATER THAN
<	LESS THAN
=	EQUAL TO
>=	GREATER THAN OR EQUAL TO
<=	LESS THAN OR EQUAL TO
<>	NOT EQUAL TO

Any command can follow the THEN. For example:

```
20  IF A=B THEN PRINT A
```

The following program INPUTs two numbers, then PRINTs them, the larger first.

```
10  INPUT"Numbers"; A,B
20  IF A>=B THEN GOTO 40
30  SWAP A,B
40  PRINT "The larger number is";A
50  PRINT "The smaller number is";B
```

In this program both variables are INPUT using the one INPUT statement. The computer displays "Numbers?" and then waits until two values have been entered. You can either type in one of the numbers, a comma, the second number and then press <RETURN>, or you can type in the first number, press <RETURN>, and then the second number and <RETURN>. If you chose the second option, the computer will display two question marks after you press the first <RETURN>. The program then continues.

In general, any number of variables can be INPUT with the one INPUT statement, as long as they are separated from the message in quotation marks by a semicolon, and from each other by commas.

The SWAP Command, in line 30, SWAPs the contents of variable A with the contents of variable B. So if the condition A>B in line 20 is not true, then, instead of jumping to line 40, the program continues with line 30. The SWAP Command ensures that the larger number is put into variable A. The next two lines PRINT the numbers out. SWAP can also be used to exchange the contents of two String Variables. The variables must be separated by commas.

The IF. . .THEN format can be extended to include the word ELSE. Everything after the word ELSE is executed IF the condition before THEN is not satisfied. The previous program could have been written.

```
10  INPUT "Numbers"; A,B
20  IF A>B THEN PRINT "The larger number is";A ELSE PRINT
    "The larger number is";B
30  IF A>B THEN PRINT "The smaller number is"; B ELSE PRINT
    "The smaller number is";A
```

Multiple conditions can be used after the IF statement using AND and OR. For example:

```
10  INPUT A,B,C
20  IF A=B AND A>C THEN PRINT A
```

The last short program will only PRINT A if it is equal to B as well as being greater than C. Compare the above to the following:

```
10  INPUT A,B,C
20  IF A=B OR A>C THEN PRINT A
```

This version of the program will PRINT A if it is either equal to B, or greater than C.

The inequalities can also be used with strings. A string is stored as a number in the computer's memory, each character being represented by a different number. To compare two strings, the computer really compares the two numbers representing the first character of each string. If the first two characters are the same, then it compares the next two, and so on. Lower and upper case characters are represented by different numbers:

A-Z by the numbers 65 to 90
a-z by the numbers 97 to 122

Other characters such as "*" can also be compared. More on this later! (See CHAPTER 20)

The following strings are sorted into alphabetical order, with the "largest" first:

"zebra"
"aa"
"aA"
"a"
"Hello"
"AND"

To summarise what you have learnt in this chapter, here is a short program which makes full use of the IF, THEN, and SWAP statements.

This program sorts out three numbers given and displays them in order, the largest first:

```
10 REM SORT PROGRAM
20 INPUT "Three numbers"; A,B,C
30 IF B>A THEN SWAP A,B
40 IF C>B THEN SWAP B,C
50 IF A>B THEN SWAP B,A
60 PRINT A,B,C
```

Have a go at this program by INPUTting various numbers. Can you work out the logic of this program?

EXERCISES

1. Using the above program, write a program which sorts three strings into alphabetical order.
2. The example program can only cope with three numbers. Can you write a better one which can handle more than three numbers? One is given in CHAPTER 13.

CHAPTER 7

USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

A useful time saving command is AUTO. On typing this in, you will find that line number 10 AUTOmatically appears on the screen. Type REM after this and press <RETURN>. Now line number 20 will appear. This process will go on indefinitely. To STOP the AUTOmatic Line Numbering, press <CTRL><STOP>.

Before typing in a program it is advisable to use the Command NEW, which deletes any old program lines and resets the variables. If you do not do this, you may well find that some of your old program lines are mixed in with the new program, especially if the last program was longer than the present one. This can result in chaos!

As you are already aware, the LIST statement is used to LIST out your program on to the screen. If you just wish to LIST one line of your program (you might have an Error Message referring to this line), type in LIST followed by the Line Number:

```
LIST  30
```

This just results in line 30 being displayed on the screen.

On LISTing your program, you will probably have noticed that all the lower case characters in the keywords and variable names are replaced by upper case characters.

It is always a good idea to title your program using a REM statement, and to sprinkle these liberally throughout the program to explain each part. This makes life a lot easier when you come to edit your program, especially if you are doing this some time after you originally wrote it.

The REM statements can always be removed at a later date if you find

you are running out of memory. As you might well want to remove a REM statement later on, NEVER have a GOTO command referring to a REM statement.

The simplest way to delete a single line from a program is simply to type the line number in and then <RETURN>. If you wish to delete a large chunk of a program, however, it is quicker to use the DELETE command:

```
DELETE 20-80
```

This will delete all the lines with line numbers between 20 and 80 inclusive. Be careful not to delete the wrong lines by accident, as once you have used this command, the only way to get the lines back is by typing them in again!

More than one command can be included in one line but the commands must be separated by the colon. For example:

```
20 PRINT"a":GOTO 10
```

This is a valid line. The letter "a" will be printed and then the computer will jump to line 10. Any number of commands can be linked together in this way. This will save on computer memory as a certain amount of memory is necessary to save a line number. A program is generally more readable, however, if you avoid the use of multiple statements. There is also a limit on the number of characters which can be included in one line — up to 255 characters may be typed in, including spaces. The computer ignores anything over this.

To demonstrate this, type in the following:

```
10 PRINT"**************....."
```

filling the screen with "*"s. Now press <SHIFT><CLS> and RUN the program; only 247 "*"s will be PRINTed. The computer has only executed the line up to the 255th character. Notice that if you miss out the final quotation marks of a string, the computer automatically assumes their presence. However, do not get into the habit of doing this as it can produce confusing results.

One more point about PRINT — it has an abbreviated form, the question mark. Type in:

```
10 ?
```

Now type LIST. The question mark will have been replaced by PRINT.

On inspecting your listed program, you might find that you have missed a line out, or even a number of lines, by accident when typing your program in. It is possible that there is not enough room to insert the missing lines due to inadequate spacing of the line numbers. If this happens, you can RENUMber your program using the RENUM Command. For example:

```
10  REM
11  REM
12  REM
 -   -
 -   -
```

To insert a line between lines 10 and 11, type RENUM. Now LIST your program, you should see:

```
10  REM
20  REM
30  REM
 -   -
```

You can now easily insert new lines between lines 10 and 20, using line numbers 11 to 19.

It is possible to store more than one program in memory. To do this you could store the first program between line numbers 10 and 100, for example, and the second between 500 and 1000. When you type in RUN, you must prevent the computer from executing the first program and then going straight on to the second. To do this, use the END statement. This must be the very last statement in the first program. When the computer encounters END it stops execution and puts you back into Command Mode.

To continue execution, which in this case means RUNning the second program stored in memory, you can type CONT followed by <RETURN>. Alternatively, to RUN the second program before the first, you can go straight to it using:

```
RUN  500
```

This starts execution at line 500. The same result is obtained using:

```
GOTO  500
```

The STOP Command can also be used in a program. This command, like the END Command, puts you back into Command Mode. However, it produces the message "BREAK in LINE ..", giving the line number of the STOP statement. After a STOP statement, as you are in Command Mode, you can continue execution by typing in CONT.

You can STOP a running program at any point by pressing <STOP>. To continue execution press this key again. You will not be able to use the command CONT in this case, as this can only be used with STOP statements within programs. If you press <CTRL> <STOP>, however, this STOPs the program and puts you back into Command Mode. Once in Command Mode you can restart execution from the next line by using CONT. Type in the following short program:

```
10  REM
20  PRINT "Goodbye"
30  GOTO 20
```

RUN this program. Now Press < STOP> . This will stop the program. The only way to restart execution is to press the< STOP> key for a second time. RUN the program again, this time STOP it using <CTRL> <STOP>. Restart the program using CONT.

```
10  REM
20  PRINT "HELLO"
30  STOP
40  PRINT "GOODBYE"
50  END
60  GOTO 20
```

Restart the program, using CONT after both the END and STOP statements.

A rather different command, which you will not need to use at the moment but which you should be aware of, is CLEAR. This tells the computer to forget all the variables currently stored in memory. It can also be used to make more room for a variable. On turning the computer on, strings can only be 200 characters long. You can increase this to 255 characters by typing:

```
CLEAR  255
```

Another command which deals with the computer's memory is FRE. This tells you how much memory you have FREe in either the BASIC program area, or in the string storage area:

PRINT FRE(0)	Gives the memory left in the BASIC programming area.
PRINT FRE("")	Gives the memory left in the string area.

Finally, two commands which are designed to help you TRace errors in your program. TRON, which stands for TRace ON, makes the computer display the line number of each line as it executes it. Type in:

```
10  REM USE OF TRON
20  PRINT "Hello, are you still awake?"
30  INPUT "yes or no"; A$
40  IF A$="no" THEN BEEP : GOTO 20
50  PRINT "GOOD"
```

Now type TRON and RUN the program. You should get the following display:

```
[10][20]  Hello, are you still awake?
[30]      yes or no?
```

Here the program waits for an INPUT. If you INPUT yes, you get:

```
[40][50]GOOD
```

If you INPUT no, you get:

```
[40][20]  Hello are you still awake?
[30]      yes or no?
```

The BEEP Command, in line 40, is the easiest command to use to get a noise out of the computer.

To stop the tracing process, use TROFF.

CHAPTER 8

THE FUNCTION KEYS

These are the five keys at the top of the keyboard. Each key has two functions, one of which is displayed on pressing the key directly, the other on pressing <SHIFT> and the key SIMULTANEOUSLY, i.e. the functions F1, F2, F3, F4 and F5 are used by pressing the relevant key, while F6, F7, F8, F9 and F10 by pressing <SHIFT> and the corresponding key at the same time.

On turning on the computer, the keys are already programmed for certain functions. A list of these functions may be displayed by typing the command:

KEY LIST

The contents of the first five keys are displayed on the last line of the screen. The other five are displayed on pressing <SHIFT>. To get rid of this display type KEY OFF, then, to bring it back, KEY ON. The last three commands can be included in a program if so desired.

For the moment, forget KEYs F1, F6 and F7. There will be more said about these KEYs later. The functions of the other seven keys should be familiar to you by now, and you will probably find them useful:

KEY	DISPLAY	DESCRIPTION
F1	color	See CHAPTER 22
F2	auto	This is programmed to display AUTO on the screen, followed by a space, i.e. pressing this key is equivalent to typing "AUTO".
F3	goto	Pressing this key is equivalent to typing in GOTO followed by a space.
F4	list	This is equivalent to typing LIST followed by a space.

KEY	DISPLAY	DESCRIPTION
F5	run	The function of this key is slightly different from the ones you have already met. It is equivalent to typing RUN followed by <RETURN>, so this key will RUN a program, if one exists in memory, as soon as it is pressed.
F6	color 15,4,7	See CHAPTER 22
F7	cload"	See CHAPTER 11
F8	cont	This is equivalent to CONT followed by <RETURN>. If you have stopped a program running by pressing <CTRL><STOP>, or by an END or STOP statement in a program, you can restart execution, from the next line, by pressing <SHIFT> and F8
F9	list.	F9 will straight away LIST the program from the last line number typed in, and move the cursor to the beginning of that line.
F10	run	This function key is useful as it CLearS the screen and then RUNs any program in memory.

It is possible to redefine the action of any function key yourself by using the command, KEY, followed by the key number, a comma, then a string containing the new function. Type in:

```
KEY 1, "PRINT"          <RETURN>
```

This will result in the statement PRINT being displayed on the screen whenever KEY 1 is depressed.

Notice that the list at the bottom of the screen has changed. COLOR has been replaced by PRINT.

```
KEY 6, "NEW" + CHR$(13)
```

programs KEY 6 to carry out the command NEW. CHR$(13) is equivalent to pressing <RETURN> (see Chapter 20 for greater detail).

When defining a KEY, you may only use a maximum of 15 characters in the string. Note that CHR$(13) only takes up one character position.

EXERCISE

1. Program KEY 7 to RENUMber your program immediately the KEY is pressed.

CHAPTER 9

MORE ON PRINT AND THE SCREEN

So far you have used the PRINT statement to PRINT strings and variables, with either a comma or semicolon separating the list of items to be PRINTed. The semicolon has resulted in the items being PRINTed next to each other; the comma has resulted in two items being PRINTed on the same line, the first at the beginning of the line, the second at the 15th character space.

A semicolon at the end of a PRINT statement suppresses the carriage return, so the next PRINT statement will start PRINTing on the same line:

```
10  PRINT "To be, or not to be—";
20  PRINT "that is the question;"
```

This is all PRINTed on one line:

```
To be. or not to be --- that is the question;
```

You will probably find that at some stage you want to print a table of results, or perhaps a list of options neatly tabulated. Two commands to help you to do this are TAB and LOCATE.

To understand how to use these commands you should be aware of the different Screen Modes.

There are two text screens, SCREEN 0 and SCREEN 1. On switching on the computer, you are automatically in one of these SCREENs. For this Chapter it will be assumed that the default screen is SCREEN 1. To make sure you are in this screen, type:

```
SCREEN 1
```

This screen is 29 character positions wide by 24 long. Each character position occupies a square which is made up of 8x8 points.

The other text screen, SCREEN 0, is 39 character positions wide by 24 long. The character positions in this screen are slightly smaller, each occupying a rectangle of 6x8 points. In this screen the border is the same colour as the background, so the whole screen is blue on initially switching on. To access this screen type:

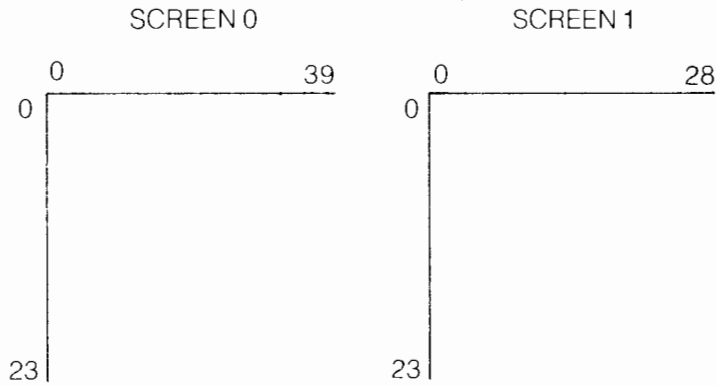
```
SCREEN 0
```

Now return to screen 1, by typing:

```
SCREEN 1
```

The origin of both the TEXT SCREENS is taken to be at the top left hand corner of the screen. Notice that the first row and column on each screen is called Row 0, and Column 0, respectively. So, the last column on Screen 0 is Column 39 and on Screen 1 is Column 28. The last row on both screens is Row 23.

You cannot PRINT on the last row unless you first use KEY OFF to get rid of the Function Key list.



You can change the maximum number of characters on each line from the default values by using the WIDTH statement, followed by a number. This number sets the maximum WIDTH of the screen. In Screen 0, the maximum number of characters you can set across the screen is 40, in Screen 1, it is 32. You can decrease the width of both screens down to just 1 character wide if you wish!

```
SCREEN  0
WIDTH 20
```

These commands set the maximum WIDTH of SCREEN 0, to 20 characters. You will find now that you can only write in the 20 most central columns. To get back to the default width and screen, type:

```
SCREEN 0
WIDTH 29
```

Now back to tabulation! The TAB command is always used in conjunction with PRINT. This statement causes the computer to start PRINTing at the specified column (or X co-ordinate) given the TAB statement. For example:

```
PRINT TAB(10) "This starts Printing at the 11th character position."
```

Don't forget the first character position on the screen is at Column 0, Row 0. Alternatively you can think of it as being at the X, Y character co-ordinate position (0,0).

You can use more than one TAB statement in a PRINT statement:

```
PRINT TAB(3)4 TAB(7)8
```

This will PRINT the number 4 at the fourth column. The blank space which always precedes a positive number is PRINTed at the third column. The number 8 is PRINTed on the same line, the preceding blank being at the seventh column. No punctuation is needed between TAB statements, but do not forget the brackets.

You can only specify the column in a TAB statement, so if you wish to PRINT at a particular row as well as column, you must use the LOCATE Command just before the PRINT Command. This command sets the Cursor position; the first number in the LOCATE Command is the column number (or X co-ordinate), the second the row number (or Y co-ordinate). For example:

```
10  LOCATE 6,2
20  PRINT "NAME"
30  LOCATE 20,2
40  PRINT "SCORE"
```

which PRINTs:

```
NAME        SCORE
```

on the third line of the screen.
You need a LOCATE Command with each PRINT statement.

Alternatively, you can just use LOCATE to set the row, and then use PRINT TAB. The following program gives exactly the same result as the last one:

```
10  CLS
20  LOCATE ,2
30  PRINT TAB (6) "NAME" TAB(20) "SCORE"
```

When the X co-ordinate is left out of a LOCATE statement, the computer assumes you wish to stay at the present X position. The CLS Command

in line 10 has set X=0, so this value of X is used when locating the Cursor in line 20. Note that although you may leave out the X co-ordinate, you must not leave out the comma.

LOCATE cannot be used in Command Mode, because as soon as a command is executed in this mode, the cursor is set to the LHS of the next line, ready for a new command to be typed in.

You can also use LOCATE to switch the cursor off within a program, i.e. to get rid of the little white square. To do this, put a comma after the Y co-ordinate and follow this with a zero. To switch the cursor back on, replace the 0 with a 1. For example:

```
10  LOCATE , ,0
```

disables the Cursor. While:

```
10  LOCATE 10,10,1
```

enables the cursor and moves it to co-ordinate position (10,10). This effect cannot be observed in Command Mode.

Now for another command, SPC. This command is used within a PRINT statement to specify the number of SPaCes to be PRINTed. The short program you typed in earlier, to PRINT out the headings "NAME" and "SCORE", could be replaced by:

```
10  CLS
20  LOCATE  ,2
30  PRINTTAB(6)"NAME" SPC(10) "SCORE"
```

Any integer number between 1 and 255 may follow the SPC statement; the number must be enclosed within brackets. Whenever you wish to PRINT a string containing a lot of spaces, it is always advisable to use the SPC statement, as this saves on memory.

There is an alternative method to PRINT spaces, using SPACE$. This is a rather more versatile command than SPC as it can also be used outside a PRINT statement, whereas SPC cannot. SPACE$ is used to form strings made up entirely of spaces. For example:

```
A$=SPACE$(10)
```

is equivalent to:

```
A$="          "
```

Using this command, we can assign our heading to a variable and PRINT this out as often as we wish:

```
10  H$="NAME" + SPACE$(10) + "SCORE"
20  CLS
30  LOCATE  ,2
40  PRINTTAB(5) H$
```

Don't forget you can't do this with SPC:

```
A=SPC(10)        IS WRONG!
```

EXERCISE

1. Write a program which INPUTs today's date and your date of birth, then PRINTs out your age. Tabulate the messages printed on the screen using LOCATE. Turn the cursor off before the INPUT statement.

CHAPTER 10

INTERACTIVE PROGRAMMING

A program is called "interactive" if at some point during its execution it accepts an input from the keyboard. This chapter is therefore going to deal with the INPUT statement, INKEY$ and INPUT$.

Although TAB can only be used with PRINT statements, LOCATE can be used with INPUT statements as well:

```
10  LOCATE(10,22)
20  INPUT "Article, Price"; A$, P
```

You will see:

```
Article, Price?
```

appear on Row 22 of the screen. The computer will wait until you have INPUT a string and then a number. To do this, as you already know, you can press <RETURN> after INPUTting the String Variable and then INPUT the number, or INPUT them together separating the string and number by a comma. If you use the statement LINE INPUT, however, you could INPUT the whole line of text into one string variable:

```
10  LOCATE(10,22)
20  LINE INPUT "Articles and price?" L$
```

You will see:

```
Articles and price?
```

on the 22nd Row of the screen. Now if you typed in:

```
Bananas, peanuts and tea bags      £25.00
```

the whole line, including the blanks, would be put into the variable L$. If you replace the LINE INPUT with an INPUT statement, everything after the comma is ignored.

LINE INPUT can only be used with String Variables, so:

10 LINE INPUT A IS WRONG!

The line you type in can be any length of up to 200 characters. The length of the line can be extended up to 255 characters, by using the CLEAR statement (see CHAPTER 7).

Unlike INPUT, LINE INPUT does not automatically produce the question mark prompt, so if you want this you must include it in the message after the KEYWORD. More than one line can be INPUT in one LINE INPUT statement, but then the String Variables must be separated from each other by commas and from the message, if any, by a semi-colon.

At some point in your program, you might want the computer to stop and ask you which of several options you want to take. For example, in a games program you might want the computer to ask the player if he wants the instructions displayed, or at the end of the game whether he wants to play again. You can use the INKEY$ Statement to do this. This function tests the keyboard to see if a key is being pressed. If one is, the key character is entered into the string Variable INKEY$. If not, an empty string is returned into the Variable INKEY$ and the computer carries on with the next program line.

The next part of a program could be put at the end of a game:

```
100 CLS
110 LOCATE ,10
120 PRINT "Do you wish to play again?"
130 PRINT "Press Y for Yes"
140 PRINT "Press N for No"
150 A$=INKEY$
160 IF A$="N" OR A$="n" THEN END
170 IF A$="Y" OR A$="y" GOTO 10
180 GOTO 150
```

The first time line 150 is reached, INKEY$ either contains an empty string or the character of any of the character keys being pressed. If any of the keys N,n,Y or y were pressed while the computer was at line 150, then the program either ENDs or starts again from line 10. The GOTO statement in line 180 makes the computer loop back to line 150 until one of the four character keys recognised by the computer is pressed.

Do not forget that INKEY$ is a string and can only be equated to a String Variable, so:

A=INKEY$ IS WRONG!

and will result in a Type Mismatch Error.

Another statement, similar to INKEY$, is INPUT$. Like INKEY$, this reads the keyboard directly, but instead of just reading one character, it reads the number of characters specified in the INPUT$ statement, i.e.:

```
10  A$=INPUT$(5)
20  PRINTA$
```

The computer will wait at line 10 until five keys have been pressed. Line 20 then PRINTs these characters out. The number following INPUT$ should be an integer in the range of 1 to 200. If a real number is used in this range it will be truncated to the nearest lower integer.

EXERCISE

1. Use an INPUT$ statement at the beginning of a program to input a password. The first few lines of the program should ask for the password, and execution of the rest of the program should only start once the correct password has been entered.

CHAPTER 11

SAVING YOUR PROGRAM ON TAPE

You have reached the stage where you can write fairly long programs. Instead of typing them in each time you wish to RUN one, you'll find it much more convenient to store your program on tape.

Firstly, connect your cassette recorder to the computer. Most cassette recorders will do; a cheap monoplayer is okay. It is very useful to have a tape counter to note where you have stored your program. The recorder must have an input socket for use with a microphone, and an output socket for use with earphones. These sockets must be 3.5mm jack sockets to take the jack plugs.

The connector lead must have an 8 pin DIN plug on one end. If the lead is not included with your MSX computer, please buy one. You must specify that it is for an MSX computer. Plug this into the computer — in the socket marked "CASSETTE". The other end of the lead divides into three leads, with red, black and white jack plugs on their ends.

Plug the white lead into the socket marked "EAR" on your recorder and the red lead into the socket marked "MIC". Forget the black lead for the moment.

Put an empty cassette into your cassette recorder. Rewind it if necessary and then play it until you reach the end of the transparent leader at the beginning of the tape. Reset the tape counter.

Now type a short program into your computer. Then type.

CSAVE"name" <RETURN>

You can give your program any name you wish as long as it does not exceed six characters. The characters may be upper or lower case

letters of the alphabet, or numbers. The first character of the name must be a letter.

Now depress both the RECORD and PLAY buttons on your cassette recorder, then press <RETURN> on the computer.

When the computer has finished recording the program, it will come up with the "Ok" message. Stop the tape and note the tape counter reading.

To check if you have saved your program successfully, switch off the computer to get rid of the program in memory. Rewind the tape to the beginning or the zero setting of the counter. Now switch the computer back on and type:

```
CLOAD"name"      <RETURN>
```

or just:

```
CLOAD      <RETURN>
```

If the program name is not specified, the computer will load the first program it finds on the tape. Now depress the PLAY button on the cassette recorder. You should get the following message on the screen once the computer has found the program:

```
FOUND: name
```

When the program has been loaded you get the "Ok" message. Once that appears, stop the tape.

Notice that the function key, F7, is programmed to display CLOAD" on the screen. Using this, you just need to type in the program name and the final quotation marks.

You can verify that the program in the computer is the same as the program on the tape, i.e. that no saving or loading errors have occurred, by using the command CLOAD?. To use this, rewind the tape to the beginning of the program and type:

```
CLOAD?"name"      <RETURN>
```

Start the tape.

If the program in the memory is the same as the program on the tape you will get the "Ok" message. If you do not get this, you have not loaded the program successfully — see the problems section at the end of this chapter.

When storing several programs on one tape, it is a good idea to space them out well and note down the tape counter readings at the beginning of each program. If you record a program on top of another program, obviously you will lose the first program.

Supposing you have two programs stored on tape, the first called "plot", the second called "point". If you enter:

```
CLOAD"point"
```

and start the tape from the beginning you will get the following display:

```
CLOAD"point"
skip :plot
Found:point
Ok
```

The computer tells you it has passed over the program called "plot", and has then loaded "point".

Now for the function of the black jack plug. You can only use this if you have a remote control socket on your cassette player, marked "REMOTE". If you have, plug in the black plug and type:

MOTOR ON <RETURN>

If you now do a CLOAD/CLOAD? or a CSAVE, the motor in the cassette recorder will be automatically switched off by the computer when it has finished reading from or writing to the cassette recorder. This is very useful, as it is easy to forget to turn the cassette recorder off after using CLOAD or CSAVE.

To stop remote control of the cassette recorder, type:

MOTOR OFF <RETURN>

Just typing:

MOTOR <RETURN>

will have the same effect as this switches control off if it is on, and on, if it is off.

Saving and Loading Errors

If you've been having problems saving and loading your programs, check:

1. You have plugged the cassette recorder in!
2. You have got the leads correctly plugged in and that they have not come loose.
3. Try varying the volume and tone control on the cassette recorder. Usually it's best to leave the tone on its maximum setting and just vary the volume control. Start at the midway setting and try increasing or decreasing the volume from this.
4. Check you are spelling the name correctly if you cannot load a program you have already saved.

If you still cannot save or load your programs, you have probably got an unreliable cassette recorder.

CHAPTER 12

READING DATA INTO ARRAYS

So far you have only INPUT a limited number of items at any one time. There is bound to come a point when you will want to record a lot of data, perhaps a long list of names or numbers. Supposing you wanted to keep a telephone directory of all the people you know. You could try the following:

```
10  REM DIRECTORY VERSION 1
20  INPUT " Name and Number";N1$,NUM1
30  INPUT " Name and Number";N2$,NUM2
40  .....
```

and so on, but with only ten names this is going to be a long and repetitive program. It would be much better to use a loop, executing the loop once for each entry in the directory. To do this, you will need to use an Array.

An Array is a whole group of variables which all have the same name but differ from each other by a subscripted number.

A(n) is a Numeric Array. If n=10, then this Array can hold eleven variables called A(0), A(1), A(2) ... A(10). You can think of A as being a street name, in which case the following number gives you the house number in the street.

If you are going to use arrays with more than 11 elements, you must let the computer know you are going to do this by telling it the size of the array and its name. This is done using a DIM statement:

```
DIM NUM(12)
```

sets up what is called a One Dimensional Numeric Array called NUM, which has thirteen elements, NUM(0), NUM(1), ... NUM(12). The array is

called One Dimensional as it can be represented by a One Dimensional diagram.

NUM(0)
NUM(1)
NUM(2)
NUM(3)
NUM(4)
NUM(5)
NUM(6)
NUM(7)
NUM(8)
NUM(9)
NUM(10)
NUM(11)
NUM(12)

The first character of an array name must be a letter of the alphabet, the rest can be letters or numbers. As with simple variables, KEYWORDs or punctuation marks must not be included in the name. Only the first two characters are recognised by the computer, and lower case letters are not distinguishable from upper case. Therefore, the following all look the same to the computer:

ab(10)
AB(10)
Abbacus(10)

If you wish to read strings into an array, you must use a String Array. The only difference in appearance between this and a Numeric Array is that the name must end with the "$" sign (as indeed a String Variable Name must).

Permitted names include:

Game$(20)
Address($10)
A1$(5)

These would not be allowed:

TOWN$(12)	Includes Command TO
Question?$(6)	Punctuation marks are not allowed.
name$(200)	NAME is a reserved word.

You can dimensionalise all the arrays you are going to use, in one statement at the beginning of the program:

DIM A$(12),B(13),C(20)

Do not DIMensionalise more array space than you need, as on receiving a DIM statement the computer sets aside sections of memory for

each array, so if you then don't use all of the array you are wasting memory.

You can save on memory by using integer arrays, rather than real number arrays, i.e. NUM%(12), will hold 13 integer variables, in elements NUM%(0), NUM%(1)... NUM%(12).

Multi-dimensional Arrays are dealt with in the next Chapter.

You can now read your names and numbers into the telephone directory using a loop:

```
10 REM Telephone Directory Version 2
20 REM Input entries into Arrays N$ and NUM$
30 CLS
40 INPUT "Number of entries", E
50 N=E-1
60 DIM N$(N), NUM(N)
70 FOR I=0 TO N
80 CLS
90 LOCATE ,10
100 INPUT "Name    "; N$(I)
110 INPUT "Number"; NUM(I)
120 NEXT I
```

The variable N is used in the DIM statement in line 40, to ensure that there are the same number of elements in the arrays N$ and NUM as entries in the directory. N is one less than E, as the first element of each array is element(0) and not element(1).

All the names and telephone numbers have been entered into two arrays. The next part of the program displays the directory. The program now looks like this:

```
10 REM Telephone Directory Version 2
20 REM Input entries into Arrays N$ and NUM$
30 CLS
40 INPUT "Number of entries", E
50 N=E-1
60 DIM N$(N), NUM(N)
70 FOR I=0 TO N
80 CLS
90 LOCATE ,10
100 INPUT "Name    "; N$(I)
110 INPUT "Number"; NUM(I)
120 NEXT I
130 REM display directory
140 CLS
150 PRINT TAB(2) "NAME" TAB(18) "NUMBER"
160 PRINTTAB(2) "************************"
170 FOR I=0TO N
```

```
180 LOCATE 2,4+I*2
190 PRINT N$(I)
200 LOCATE 18, 4+I*2
210 PRINT NUM(I)
220 NEXT I
```

When the program finishes, you have lost all your names and numbers! It is possible to save data at the end of a program. You can then store the data on tape, with the program.

To do this use the READ and DATA Commands.

A DATA statement can be followed by a long list of numeric or string data, the data items being separated by commas:

DATA Tom, 440 2073, Henry, 224 6607, Graham, 442 1708, ...

The only constraint on the number of data items stored is that the total number of characters in the line must not exceed 255.

You can have as many DATA statements as you like in a program. They can be stored anywhere you wish, as the computer will skip over DATA statements until it comes to a READ statement. It is often best to keep all the DATA statements at the end of the program, however, as this way it is easier to change DATA statements without hunting through the whole program.

Let us now go back to the directory program. This time, instead of INPUTting the data while the program is running, the latest version of the program already contains the data in DATA statements. We know the number of entries in the directory as we have already put them into DATA statements, so we no longer need to INPUT the number of entries. The number of entries is put into the variable E, as you might want to change the number of entries in the directory at a later date, and so need to change this number. In that case, only line 30 will have to be altered. The program now looks like this:

```
10 REM Telephone Directory Version 3
20 REM Read Data into an Array
30 E=4
40 N=E-1
50 DIM N$(N),NUM(N)
60 CLS
70 REM display DATA
80 PRINTTAB(2) "NAME" TAB(18) "NUMBER"
90 FOR I =0 TO N
100 READ N$(I), NUM(I)
110 LOCATE 2,4+I*2
120 PRINT N$(I)
130 LOCATE 18, 4+I*2
140 PRINT NUM(I)
```

```
170 NEXT I
175 END
180 DATA BATMAN ,774 0303, ROBIN, 420634
190 DATA "Dennis",444 2064, Baloo, 40 235
```

When the first READ statement is encountered, the computer READs the first item of DATA from the first DATA statement into the variable in the READ statement. In this case, BATMAN is READ into the element N$(0) of the array N$.

The second time a READ statement is encountered, the second item of DATA in the DATA statement is READ. Once all the items in a DATA statement have been READ, the computer goes to the next DATA statement and READs this sequentially.

You might think that it is unnecesary to READ the DATA into an array in this program. You would be right! However, once the DATA is in an array, we can then process it, if we wish, before displaying it. The next Chapter gives the final version of this program, in which the entries are sorted into alphabetical order before they are displayed.

You may use more than one READ statement to READ a DATA statement. The next READ statement starts from the first unREAD DATA item. For example:

```
10  READ A,B
20  READ A$,B$
30  PRINT A;B;A$;B$
40  DATA 10, 20, "This is a sentence. ",hello
```

results in:

```
10  20 This is a sentence. hello
```

Do be careful when using DATA statements. You must make sure that the order of the DATA in the statement corresponds to the order of the variable types in the READ statement. If you are READing into a Numerical Variable, you must have a number at the corresponding position in the DATA statement.

You cannot put variables in DATA statements, only strings and numbers. Strings, in DATA statements, don't have to be enclosed in quotation marks unless they contain commas, colons or spaces.

Sometimes you might wish to use the same DATA statement twice. The first time round you might just wish to PRINT the DATA, the second time to do some calculations with it. To make the computer go back to a DATA statement it has already READ, you use the RESTORE command.

RESTORE on its own makes the computer re-READ the DATA from the very first DATA statement. RESTORE, followed by a line number, makes the computer re-READ from the first DATA statement at or after that line number. Make sure that you follow RESTORE by a valid line number, i.e. one that exists in the program. Run the following program:

```
10  READ A,B
20  PRINT "A=";A,"B=";B
30  RESTORE
40  READ C,D
50  PRINT "C=";C, "D=";D
60  DATA 10,20
```

You will see:

```
A=10        B=20
C=10        D=20
```

CHAPTER 13

DATA HANDLING AND SORTING

In the last Chapter you were given a telephone directory program; it would be nice to PRINT the directory out in alphabetical order. We've already considered a rather limited way of sorting lists into numerical order in CHAPTER 6. The following method, however, is more flexible. (Although by no means the best way, it is one of the easiest to follow.) This method is called BUBBLE SORTING.

Let us consider a list of numbers to be sorted:

7, 4, 5, 10, 2, 8

We want to sort these numbers into decreasing order.

Let us compare the first two numbers. If the number on the LHS is smaller than the number on the RHS, then we SWAP these two numbers and then compare the second and third numbers. If the first two numbers had been in the correct order already, then we would have left them as they were and compared the second and third numbers straight away.

So, in our example, we first compare 7 with 4. These two are in the right order so we move on to compare 4 with 5. 4 is less than 5 so we SWAP these two round. We now have:

7, 5, 4, 10, 2, 8

Now we compare the third number with the fourth, i.e. 4 with 10. Again we SWAP these two:

7, 5, 10, 4, 2, 8

Comparing the fourth with the fifth, we leave things as they are. Finally, comparing the fifth number, 2, with the last, 8, we SWAP, ending up with:

7, 5, 10, 4, 8, 2

Well, we have not finished yet! But notice that the smallest number is already at the end of the list. Let's see what happens the second time through:

COMPARE	SWAP?	RESULT
1st/2nd	no	7, 5, 10, 4, 8, 2
2nd/3rd	yes	7, 10, 5, 4, 8, 2
3rd/4th	no	7, 10, 5, 4, 8, 2
4th/5th	yes	7, 10, 5, 8, 4, 2
5th/6th	no	7, 10, 5, 8, 4, 2

At the end of the second pass through, the two lowest numbers are in order at the end of the list.

The maximum number of passes it will take to get all the numbers in order in any list is therefore one less than the total number of items in the list to be sorted, i.e. in the example given, the maximum number of passes required would be five. In fact, lists are often sorted before this as some of the items are likely to be in the correct order before you start sorting. It is therefore a good idea to check whether the list is already sorted at the end of each pass through.

Let us now sort the telephone directory into alphabetical order. There are three parts to this program. The first part READs the DATA into an array, the second part sorts the DATA into alphabetical order, and the third part displays the sorted DATA.

```
10 REM Sorted Telephone Directory
20 REM READ DATA into two Arrays
30 E=4
40 N=E-1
50 DIM N$(N), NUM(N)
60 FOR I=0 TO N
70 READ N$(I), NUM(I)
80 NEXT I
90 REM sort into order
100 M=N-1
110 P=0
120 FOR C=0 TO M
130 IF N$(C) > N$(C+1) THEN SWAP N$(C),N$(C+1):
    SWAP NUM(C), NUM(C+1):P=P+1
140 NEXT C
150 IF P<>0 GOTO 110
160 REM display sorted DATA
170 CLS
180 PRINTTAB(2) "NAME" TAB(18) "NUMBER"
190 PRINTTAB(2) "**********************"
```

```
200 FOR I=0 TO N
210 LOCATE 2,4+I*2
220 PRINT N$(I)
230 LOCATE 18,4+I*2
240 PRINT NUM(I)
250 NEXT I
300 DATA CHRIS, 3333333, DENNIS, 4444444
310 DATA ANDREW,1111111, BELLA, 2222222
```

The first and last parts of this program should be familiar to you (see CHAPTER 12).

Lines 90 to 150 contain the sorting routine.

At the beginning of the loop the counter, P, is set to zero. Each time round the loop, an element of the array is compared to the next one along, i.e. the first time round, N$(0) is compared to N$(1). If the names in these elements are not in alphabetical order, the contents of the elements are SWAPped. The corresponding telephone number is also SWAPped. P is now set to 1, to indicate that a SWAP has taken place.

The last time round the loop, N$(2) is compared to N$(3). The contents of these elements are SWAPped if necessary, and the counter, P, incremented.

Execution then continues with line 150. Here the computer checks the value of P. If any elements have been SWAPped, P will be greater than 0. The computer assumes that the names are not yet sorted, and goes back to line 110. P is reset to 0, and the sorting loop, is executed again.

If, at line 150, P has been found to be 0, the computer knows that no elements had been SWAPped, thus indicating that the names are now in alphabetical order. The computer, therefore, continues with the next line of the program, and displays the directory.

If you RUN this program, you should see:

```
NAME          NUMBER
**************************

ANDREW        1111111
BELLA         2222222
CHRIS         3333333
DENNIS        4444444
```

If you wish to use the same array name twice in a program but with different data, you should ERASE the old array:

```
ERASE N$
```

wipes out all the names stored in the array N$. You can now use the array N$ again, and re-DIMensionalise it using a DIM statement.

If by chance you have a variable called N$, it will be totally unaffected by the ERASE statement. Try not to have variables and arrays with the same name, however, as this can be rather confusing.

You can have Multi-dimensional Arrays, anything up to 255 dimensions! For the moment just consider a two dimensional array. These are very useful when storing tables:

DATE	CREDIT	DEBIT	BALANCE
30.6.84	34.05		34.05
5.7.84		10.00	24.05
10.7.84	120.00		144.05
12.7.84		50.00	94.05
20.7.84	200.00		294.05

These four columns of numbers can be stored in the two dimensional array, BANK(4,3). This array is large enough to store a table made up of five rows and four columns and may be represented as follows:

BANK(0,0)	BANK(0,1)	BANK(0,2)	BANK(0,3)
BANK(1,0)	BANK(1,1)	BANK(1,2)	BANK(1,3)
BANK(2,0)	BANK(2,1)	BANK(2,2)	BANK(2,3)
BANK(3,0)	BANK(3,1)	BANK(3,2)	BANK(3,3)
BANK(4,0)	BANK(4,1)	BANK(4,2)	BANK(4,3)

As this is a Numeric Array, the date will have to be stored as one number, e.g. 300684. Each entry in the table is stored in a separate element of the array. The date 300684 can be stored in BANK(0,0), the final balance at BANK(4,3).

The following program shows you how to put data into a two dimensional array. The four columns of the array are accessed by taking values of C=0 to 3. The number of rows depends on the number of transactions entered, if there are T transactions, then R takes the values 0 to T−1. You enter the date of each transaction and the amount credited or debited, and the balance after each transaction is calculated and entered into column four of the array. The total Credits, Debits and Balance are then found and entered into the last row of the array.

The table headings are kept in a separate string array. This array does not need to be DIMensionalised before use, as it is a one dimensional array and has less than eleven elements. All multi-dimensional arrays must be DIMensionalised before use.

```
10 REM Calculate bank balance
15 REM Read table headings into an Array
20 CLS
30 FOR C=0 TO 3
40 READ HEADING$(C)
50 NEXT C
60 REM Input transactions into a 2-D Array
70 INPUT "Number of transactions";T
80 DIM BANK(T,3)
```

```
90 N=T-1
100 CLS
110 FOR C=0 TO 2
120 PRINTTAB (10*C) HEADING$(C);
130 NEXT C
140 FOR R=0 TO N
150 FOR C=0 TO 2
160 LOCATE 10*C, R+2
170 INPUT BANK(R,C)
180 NEXT C
190 NEXT R
200 REM Calculate balance after each transaction
210 BANK(0,3)=BANK(0,1)-BANK(0,2)
220 FOR R=1 TO N
230 X=BANK(R,1)-BANK(R,2)
240 BANK(R,3)=BANK(R-1,3)+X
250 NEXT R
260 REM To calculate total balance
270 FOR R=0 TO N
280 BANK(T,1)=BANK(R,1)+BANK(T,1)
290 BANK(T,2)=BANK(R,2)+BANK(T,2)
300 NEXT R
310 BANK(T,3)=BANK(T,1)-BANK(T,2)
320 BANK(T,0)=BANK(N,0)
330 REM To display bank balance
340 SCREEN 0
350 WIDTH 40
360 FOR C=0 TO 3
370 PRINTTAB(10*C) HEADING$(C);
380 NEXT C
390 FOR R=0 TO N
400 FOR C=0 TO 3
410 LOCATE 10*C, R+2
420 PRINT BANK(R,C)
430 NEXT C
440 NEXT R
450 PRINT"_____"
460 FOR C=0 TO 3
470 PRINTTAB (10*C) BANK(T,C);
480 NEXT C
490 END
500 DATA DATE, CREDIT, DEBIT, BALANCE
```

If you enter debits or credits greater than 99999.99, then you will mess up the table as only ten character spaces have been allocated for each entry.

EXERCISE

1) Change this program slightly so that you can INPUT the date in the form 23.6.84 etc. You will need to use another string array; DATE$(T) would be suitable.

CHAPTER 14

MANIPULATING STRINGS

There are quite a lot of ways you can manipulate strings. So far, you have only added one string to another:

```
PRINT "Good"+" Day"
```

is PRINTed as:

```
Good Day
```

By using INSTR you can look for a specified string within a string. In a program for an adventure game, you might want to ask the player where he or she is going next. The game might take different branches depending on whether the player types in North, South, East or West. This part of a program shows you how this could be done:

```
100  INPUT "Where are you going now";B$
110  IF B$="NORTH" GOTO 200
120  IF B$="SOUTH" GOTO 300
130  IF B$="EAST" GOTO 400
140  IF B$="WEST" GOTO 500
150  PRINT "TRY AGAIN!" : GOTO 100
```

The player, however, might well have typed in a whole sentence in response to the prompt in line 100, e.g. I'M GOING EAST.

As the program stands, this would mean that none of the branches would be executed and the player would be asked where he/she was going again. This would be repeated until the single word "EAST" was entered and would considerably slow the game down. If would be much better if the computer could scan the whole INPUT string and see if it recognised a word within the string. This in fact is exactly what the INSTR command does. The following example will show you how:

```
100  INPUT "Where are you going"; B$
110  IF INSTR(B$,"NORTH") <> 0 GOTO 200
120  IF INSTR(B$,"SOUTH") <> 0 GOTO 300
130  IF INSTR(B$,"EAST") <> 0 GOTO 400
140  IF INSTR(B$,"WEST") <> 0 GOTO 500
150  PRINT "TRY AGAIN!" :GOTO 100
```

In line 110, the Command INSTR searches the main string, B$, for the smaller string "NORTH". If it finds it, the number corresponding to the position of the first letter of "NORTH" in the main string, is returned. If EAST is not found, INSTR returns zero. So, if B4 was:

"I'M GOING NORTH NOW"

then, INSTR(B$, "NORTH") returns the number 11, as the N of NORTH occurs at the 11th character position. To check this, type in this short program:

```
10  A=INSTR("I'M GOING NORTH", "NORTH")
20  PRINT A
```

The number 11 should be PRINTed. INSTR always returns a number, so it must always be equated to a Numeric Variable.

Be careful to enclose strings, in INSTR statements, within quotation marks. String Variables already include quotation marks so these can be put as they stand within the INSTR statement.

When searching for a string you can start from any specified point:

INSTR(10,A$,B$)

This starts searching for B$ in A$, at or after the 10th character position.

This command would be useful in a Hangman program. The next program searches the word:

CONSTANTINOPLE

For the letter N.

```
10  A$="CONSTANTINOPLE"
20  B$="N"
30  C=0
40  C=INSTR(C+1,A$,B$)
50  IF C=0 THEN END
60  PRINT "there is a "; B$" " at character position "; C
70  GOTO 40
```

In this program, the expression C+1 has been used to denote where the search is to begin. The first time the INSTR command finds an "N", C=3, the next search therefore starts from character position 4. After the third "N" has been found, C=10, and the fourth and last search starts from character position 11. There are no more "N"s to be found so C is set to zero causing the program to END at line 50.

You will always get a zero returned by INSTR if you try to look for a larger string within a smaller string:

```
INSTR( "day", "good day")
```

will return a zero.

Now for three very similar functions:

RIGHT$, LEFT$ and MID$

First of all RIGHT$. Here's an example of its use:

```
10  A$=RIGHT$("The weather is terrible today",5)
210  PRINT A$
```

gives:

```
today
```

The RIGHT most 5 characters have been returned into the String Variable A$.

I expect you can guess what LEFT$ does! Here's an example anyway:

```
10  A$=LEFT$("The weather is terrible today",11)
20  PRINT A$
```

gives:

```
The weather
```

Finally MID$:

```
10  A$=MID$("The weather is terrible today",13,17)
20  PRINT A$
```

gives:

```
is terrible today
```

The first number in the MID$ statement gives the character position of the first character to be returned. The second number tells the computer how many characters are to be returned. So, in the above example, the 17 characters from the 13th position onwards have been returned into the Variable A$.

The number used in these functions must be between 1 and 255. You cannot have a line longer than 255 characters anyway, so this makes sense.

If the first numeric argument in MID$ is greater than the number of characters in the string, then you will get an empty string returned.

Once again, do not forget quotation marks round strings and don't try to equate any of these three commands to a Numeric Variable; you must use String Variables as strings are returned by these functions.

EXERCISES

1) Using INSTR, find the result of:
   a. searching for an empty or null string, i.e. " "
   b. searching for a string within an empty string
   c. searching for an empty string within an empty string.

2) PRINT out the words in "Is there anyone there?" separately. You could use INSTR to find the position of each word and then either only repeat the use of MID$, or use RIGHT$, LEFT$ and MID$.

3) Change the last program so that you can separate the words in any string which you INPUT.

CHAPTER 15

FUNCTIONS

You have already met quite a few functions, for example:

INT	CHAPTER 4
INSTR	CHAPTER 14
LEFT$	CHAPTER 14
MID$	CHAPTER 14

Generally speaking, each of these acts in the same way. You have an object, called the argument, on which the function acts to produce a result. Once a function is defined, it always produces its results in the same way if given a suitable argument:

```
INT(X)
```

X is the argument here. The argument of a function is always enclosed in brackets. The argument must be a number, a numeric variable, or a numeric expression. As long as X is in this range, INT will always return the integer value of X by rounding it down to the nearest whole number.

INT(7.9)	Returns 7
INT(−2.3)	Returns −3

If you get confused with negative numbers, refer to the number line below:

```
..−6 −5 −4 −3 −2 −1  0  1  2  3  4  5...
```

The numbers are arranged in order of increasing size, going from left to right. From this you can see that −3 is in fact the nearest lower whole number to −2.2.

There is another very similar function to INT, called FIX.

FIX also acts on a real number argument to return an integer. In the case of positive numbers, it returns the nearest lower integer like INT,

but for negative numbers it returns the nearest higher integer, unlike INT:

FIX(5.5)	Returns 5
FIX(−2.2)	Returns −2

Now for two more functions which also have numeric arguments, ABS and SGN.

SGN returns the sign of the argument:

If the argument is a negative number, the function returns −1.
If the argument is 0, the function returns 0.
If the argument is a positive number, the function returns +1.

```
10 A=SGN(-402.2)
20 PRINT "SGN(-402.2)=";A
```

gives:

```
SGN(-402.2)=-1
```

ABS, on the other hand, is used to convert negative numbers into positive numbers. Positive numbers are left unchanged by this function:

```
10 A=ABS(-402.2)
20 PRINT "ABS(-402.2)=";A
```

gives:

```
ABS(-402.2)= 402.2
```

The arguments of both these functions can be either expressions or variables:

SGN(4+3*−2)	Returns −1
ABS(percent)	Converts the number in the numeric Variable, percent, to a positive number.

Here is a short program to show you a possible use for ABS:

```
10 REM This program calculates your age
20 INPUT"Today's Date: DAY, MONTH, YEAR" DT,MT,YT
30 INPUT"Your Date of Birth: DAY, MONTH, YEAR" DB,
   MB,YB
40 DA=ABS(DT-DB)
50 MA=ABS(MT-MB)
60 YA=ABS(YT-YB)
70 PRINT "You are";YA;"years,";MA;"months,";DA;
   "days."
```

Now for two functions which deal with both strings and numbers, VAL and STR$.

If a string contains a number, VAL will return the number into a Numeric Variable, i.e. it strips off the quotation marks and removes any blanks within the string. For example:.

```
A=VAL("   273")
PRINT "A=";A
```

gives:

```
A= 273
```

If there is more than one number in a string, VAL only returns the first number it comes across. For this reason VAL does not evaluate expressions within strings:

```
PRINT VAL("3+4=7")
```

gives:

```
3
```

Do not forget that although the argument of VAL is a string, the result returned by VAL is a number and must be put into a Numeric Variable:

```
A$=VAL(7)          IS WRONG
```

VAL cannot cope with numbers embedded between letters in a string:

```
10  A=VAL(" Today is my 20th birthday.")
20  PRINT A
```

gives:

```
0
```

The opposite function of VAL is performed by STR$; it converts a numeric argument into a string:

```
A$="I am   "+STR$(60) +"   today"
PRINT A$
```

gives:

```
I am 60 today
```

Do not forget that although the argument is a number, the result is a string, so you must return it into a String Variable.

In CHAPTER 13 we used a Numeric Array, BANK, to store dates, credits, debits and balances. Unfortunately the date had to be entered as a number, i.e. 240684 rather than 24/6/84. To enter it in the second form, you had to store the dates separately in another string array, so the program ended up using one Numeric Array for the transactions, and two string arrays for the headings and dates. Using STR$, all the data could have been stored in one large String Array. To do this the following kinds of alterations would have to be made:

```
170  INPUT BANK
175  BANK$(R,C)=STR$(BANK)
```

To calculate the new balance after each transaction, you would now first have had to evaluate the strings in the credit and debit columns,

using VAL, and then convert the answer back into a string:

```
210 BANK$(0,3)=STR$(VAL( BANK$(0,1))-VAL(BANK$(0,2)))
```

Perhaps you would like to go back and alter the BANK BALANCE program!

One last function which acts on a string is LEN. LEN returns the length of the string given in its argument:

```
10  A$="CONSTANTINOPLE"
20  A= LEN(A$)
30  PRINT "The word ";A$;" is";A;"characters long."
```

Again, note that although this function acts on a string, it always returns a number, so you must equate this function to a Numeric Variable.

Finally, a rather different function, RND.

RND generates a 14 digit random number between 0 and 1. In fact, the numbers generated are not truly RaNDom as they follow a fixed sequence. However, this sequence is so long, that for all intents and purposes, you can consider the numbers generated to be truly random.

You can have three types of arguments with RND:

If the argument is positive, the number generated by RND within a program is the next one in the sequence. The point at which the sequence is entered is always the same, and is reset by the computer at the end of the program. Type in:

```
10  PRINT RND(1)
20  PRINT RND(1)
30  PRINT RND(1)
```

You will get the following sequence each time the program is RUN:

```
.59521943994623
.10658628050158
.76577651772823
```

Changing the argument to a different positive number will have no effect.

If the argument is zero, the random number generated is the same as the last one. For example:

```
10  X=RND(1)+RND(1)
20  PRINT"The second random number generated
    was";  RND(0)
30  PRINT"The first random number generated
    was";  X-RND(0)
```

If the argument is negative, the part of the sequence from which the random numbers are taken is set by the negative argument:

```
10  A=RND(-3)
20  B=RND(1)
30  PRINT A,B
```

gives the seqence:

```
.84389820420821
.29624868166920
```

each time the program is RUN. Now replace −3 by −4. You will get a different sequence. Changing the positive argument will again have no effect.

A random number between 0 and 1 is rarely of much use. You are much more likely to want a number between 1 and 6, for example. This program simulates the throwing of a dice:

```
10  REM dice program
20  A=INT(6*RND(1)+1)
30  PRINT"You have thrown a ";A
```

However, you get the same number each time the dice is thrown. If you want to get a different random number each time, put the line:

```
RDRST=RND(-TIME)
```

at the beginning of your program.

The TIME function returns the value of the computer's internal clock. On switching on the machine, the clock is set to 0 but is incremented every 1/50th of a second, i.e. TIME will equal 50 when 1 second has elapsed. So using TIME as the argument of a RND function, will result in the sequence of random numbers being read at a different point each time the RND function is encountered in the program.

Do you see how you might use this and the last function in the Hangman game? The game might take this sort of a structure:

1) READ a word at RaNDom from a DATA statement (these statements could be put together after the END of your program). To READ a line at RaNDom, use RESTORE followed by a randomised line number. Think how you might access more than one word in a DATA statement.
2) Once you have READ your word, let the player know how long it is, using LEN.
3) Ask the player to guess a letter.
4) Using INSTR, check to see if the letter is in the word. (See CHAPTER 14 if you have forgotten how to do this.)
5) Count how many guesses the player has made.
6) If the player has not got the word after 10 guesses, END the program. The computer has won.
7) Ask the player if he/she wishes to play again by using INKEY$.

EXERCISE

1) Write the Hangman Program!

CHAPTER 16

DEFINING YOUR OWN FUNCTIONS

So much for the computer's defined functions, let us now define some of our own. This can be done using DEF FN.

Once you have defined a function and given it a name, you can use it anywhere in the program.

Let us take a mathematical example first.

Supposing you want to define a function which will calculate the square of a number, i.e. raise the number to the power of two, which is the same as multiplying the number by itself.

Let us call this function SQUARE and define it by:

```
DEF FNSQUARE (X) = X ^ 2
```

Note that the name of the function comes directly after the command DEF FN.

X is what is called a "dummy parameter". You only use it to show what the function does. When using the function, you must pass a real parameter into it to replace the dummy parameter. This real parameter has to be of the same type as the dummy parameter. In the example given therefore, the real parameter can be either a Numeric Variable or number. When defining a function, the dummy parameters used in the definition must be listed within brackets after the function name.

Let us now use this function to calculate the SQUARE of 13, and put the result into the variable R. To do this, we use the command FN, followed by the name of the function and then the real parameter in brackets:

```
R=FNSQUARE (13)
```

On seeing this, the computer will look for the definition of the function SQUARE. On finding it, all the "X"s in the definition are replaced by 13,

the result calculated, and entered into the variable R.

X is also called a Local Variable. This means that it is only recognised locally, i.e. within the function:

```
10  X=4
20  DEF FNSQUARE(X)= X ^ 2
30  R=FNSQUARE(12)
40  PRINT"R=";R; " X=";X
```

gives:

```
R=144 X=4
```

You can use more than one parameter in a function. All the parameters used must be listed in the definition, the parameters being separated by commas. When calling the function, you must pass the same number of real parameters into the function as dummy parameters used in the definition. You must also pass the correct type of parameter in, i.e. a number or Numeric Variable for a numeric dummy parameter.

The next program will raise any number to the power of another number, both numbers being passed into the function definition:

```
10  DEF FNPOWER(X,Y)= X ^ Y
20  INPUT "Number"; A
30  INPUT "Exponent"; B
40  R=FNPOWER(A,B)
50  PRINT A;" ^ ";B;"=";R
```

The definition of a function must fit into one program line.

Now for an example using strings. This function chops off the first character of a string:

```
10  DEF FNST$(A$)=RIGHT$(A$, LEN(A$)-1)
20  INPUT "string" S$
30  PRINT S$,FNST$(S$)
```

As this function returns a string, the function name must end with the "$" sign. Any valid Variable Name is acceptable as a function name.

EXERCISE

1) Define a function which cuts off the last four characters of a string.

CHAPTER 17

STRUCTURING YOUR PROGRAM

Often in a program you may find yourself wanting to do the same task many times, but at different points in the program. If you do find that it is necessary to repeat a program line many times, it is advisable to put that line into what is called a "subroutine". When you need the line in your program you jump to the subroutine, using the GOSUB statement. When all the lines in the subroutine have been executed, you RETURN to the main program. A subroutine may contain as many program lines as necessary. In the following example, the subroutine lies between program lines 100-130:

```
10 CLS
20 PRINT "DO YOU LIKE ME?"
30 GOSUB 110
40 PRINT "DO YOU THINK I'M BRAINY?"
50 GOSUB 110
60 PRINT "DO YOU THINK I'M GOOD LOOKING?"
70 GOSUB 110
80 PRINT "YOU'RE JUST SAYING THAT TO PLEASE ME!"
90 END
100 REM SUBROUTINE
110 INPUT "YES OR NO "; YN$
120 IF YN$="NO" THEN PRINT " OK. I'LL GO AWAY
    THEN." :END
130 RETURN
```

Notice the RETURN statement at the end of the subroutine. This is very important, as it tells the computer to RETURN to the main program. Execution continues from the first line after the one containing the

GOSUB statement.

It is a good idea to put a REM statement just before a subroutine, to distinguish it from the main program. Do not use the line number of the REM statement when calling the subroutine, however, as you might later want to get rid of all REM statements, if you start to run out of computer memory.

Make sure the computer does not execute the subroutine by accident, by putting an END statement at the END of the main program, just before the subroutines.

Another use of the GOSUB statement, is in the IF . . .THEN. . .ELSE statement. This statement tends to get rather long and complicated; the easiest way to simplify it is by using a subroutine.

The following short program asks for two different, positive numbers. The IF . . . THEN . . . ELSE statement is used to test these numbers. If they are different and positive, their product and sum are PRINTed out. Otherwise you are asked to try again:

```
10  INPUT " TWO POSITIVE, DIFFERENT NUMBERS"; A,B
20  IF A=B OR A<0 OR B<0 THEN PRINT "TRY AGAIN!" ELSE
    PRINT  "A=";A,  "B=";B:PRINT  "A+B=";  A+B:  PRINT
    "A×B="; A*B
30  GOTO 10
```

Now, using a subroutine:

```
10 INPUT "TWO POSITIVE, DIFFERENT NUMBERS"; A,B
20 IF A=B OR A<0 OR B<0 THEN PRINT "TRY AGAIN!"
   ELSE GOSUB 100
30 GOTO 10
90 REM SUBROUTINE
100 PRINT "A=";A,"B=";B
110 PRINT "A+B=";A+B
120 PRINT "AxB=";A*B
130 RETURN
```

the program is slightly longer, but much easier to read.

CHAPTER 18

PROGRAM BRANCHING

At some point you might find that you would like to have a main program, or menu program, which offers you a number of options. For example, the first few lines of the menu program might cause the following to be displayed on the screen:

```
**********MENU**********
1. Display Telephone Directory
2. Add New Entry to Directory
3. Remove Entry from Directory
4. End program
**************************
```

The next part of the program may look something like this:

```
100  A$=INKEY$
120  IF A$="1" THEN GOSUB 1000
130  IF A$="2" THEN GOSUB 2000
140  IF A$="3" THEN GOSUB 3000
150  IF A$="4" THEN GOSUB 4000
160  GOTO 100
```

Line 160 is included in case a key other than 1,2,3, or 4 is pressed.

There is a shorter way of writing this, using ON. . .GOSUB. Using this statement, you can jump to any number of subroutines you wish. In the above example, we want to be able to jump to one of four subroutines; we therefore list the line numbers of the four subroutines after the word GOSUB:

```
ON. . . . GOSUB 1000,2000,3000,4000
```

Now we have to let the computer know which subroutine to jump to. This is done by putting a number or expression after the ON, and before the GOSUB. If the expression evaluates to 1, then the computer jumps to

the first subroutine listed after the GOSUB; if it evaluates to 2, the second subroutine, and so on.

We can now replace lines 100-160 by:

```
100  A$=INKEY$
110  A=VAL(A$)
120  ON A GOSUB 1000,2000,3000,4000
130  GOTO 100
```

Notice that the string, A$, has been changed into the Numeric Variable A, before using it in the ON. . .GOSUB statement; otherwise this would have caused an errror, as only Numeric Variables may follow the word ON.

On finishing the subroutine, the computer RETURNs to line 130.

If the Numeric Variable, A, had contained a real number, e.g. 2.9, the computer would have taken the nearest lower INTeger value of this, and jumped to the second subroutine. The expression must, however, evaluate to a number between 1 and the total number of line numbers listed.

ON. . .GOTO works in exactly the same way as ON. . .GOSUB. In this case, the computer jumps to the line number in the list which corresponds to the number following the ON statement. Execution of the program continues from this line number:

```
10  INPUT "1,2, or 3";A
20  ON A GOTO 370,420,10
```

So, if:

```
A=1   the next line executed is line 370
A=2   the next line executed is line 420
A=3   the next line executed is line 10
```

Notice that the line numbers do not have to be in numeric order in the list.

CHAPTER 19

MATHEMATICAL FUNCTIONS

If you have not done much mathematics and find this chapter rather heavy going, do not hesitate to skip to the next one.

This chapter is going to deal with the mathematical functions:

^, SQR
TAN, SIN, COS and ATN
EXP and LOG

You already know how to raise a number to a power by using the ^ sign:

4 ^ 2=16 (or as it is usually written $4^2 = 16$)

In general, raising a number n, to a power p, is equivalent to multiplying n, by itself, p times. In the above example, n=4, p=2. So 4 ^ 2 is equivalent to 4*4.

The SQuare Root function, **SQR**, finds the square root of a number. This is the inverse of raising a number to the power of two, or squaring it.

If you square 5, i.e. 5 ^ 2, you get 25. To get back to 5 from 25, you take the SQuare Root:

```
10  A=SQR(25)
20  PRINT "The square root of 25 is ";A
```

giving:

```
The square root of 25 is 5
```

The argument of the SQR function must be a positive number or an expression which evaluates to a positive number. (The computer can't handle imaginary numbers!)

Here are a few more points to think about.

What happens if you raise a number to a negative power? It is easy to

find out. By typing in:

```
PRINT 4 ^ -1
```

you get:

```
.25
```

which is 1/4, so in general:

```
n ^ -p = 1/n ^ p
```

Let us see what happens if a number is raised to the power of a decimal number, or fraction. Typing in:

```
PRINT 4 ^ 0.5
```

gives you the answer 2.

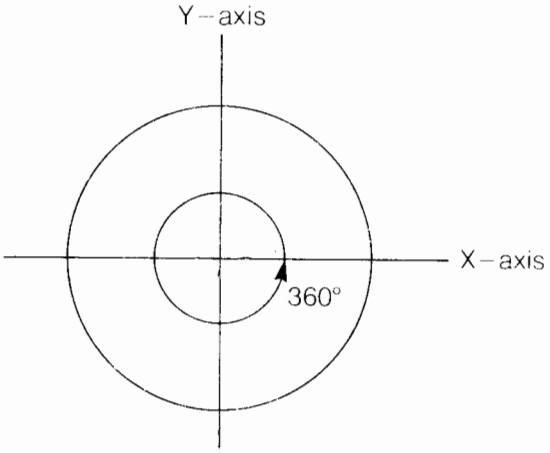
4 ^ 0.5 is equivalent to $4^{1/2}$, so in general:

n ^ 1/p gives you the pth root of n

In the example, n=4, p=2, so you got the second, or as it is better known, the square root of 4, which is 2. So, raising a number to the power of a half, is equivalent to taking the square root of the number.

SIN, COST, TAN and **ATN** are trigonometic functions.

The arguments of these three functions must be angles. You are quite possibly used to thinking of angles in terms of degrees:



A circle subtends 360 degrees at its centre. The angle of the circle is measured from the X-axis, which bisects the circle with a line drawn from the nine o'clock position to the three o'clock position, in the anti-clockwise direction. The Y-axis bisects the circle from the twelve o'clock position to the six o'clock position. The angle between these two

axes is therefore 90 degrees.

However, the computer measures angles rather differently. To understand how, let us consider a circle with a radius of one unit. The type of the unit does not matter here; it can be 1mm, 1m, 1km — anything in fact.

The circumference of the circle is given by the formula:

C=2*PI*r

C=Circumference
PI is a special number used in maths, which starts 3.1415926..
PI is the Greek "p", and pronounced PI.
r is the radius of the circle

Therefore, for our circle, as r=1:

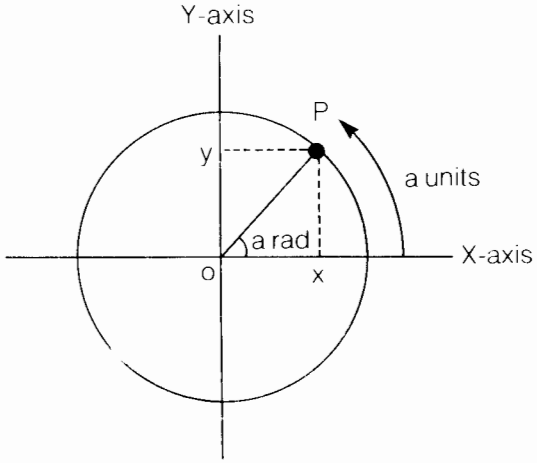
C=2*PI

Looking at this another way, we can say that the circumference of our circle subtends an angle of 2*PI radians at the centre of the circle. So:

2*PI radians is equivalent to 360 degrees.

In radians, therefore, the angle between the X and Y axes is PI/2.

If we now have a point moving round on our circle, we can use the functions COS and SIN to find out where the point is:



At a certain time, the point might be at position P, where it has moved through an angle of a radians from the X axis, or if you prefer, it has moved a distance of a units along the circumference.

If we draw vertical and horizontal dotted lines from P, to cut the X and Y axes, we can find the position of the cuts on the X and Y axes by using

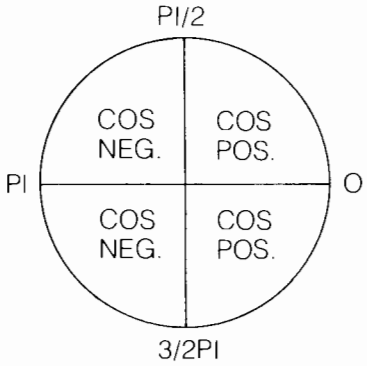
COS and SIN respectively:

x=COS(a)	COS stands for the Cosine of angle a
Y=SIN(a)	SIN stands for the Sine of angle a

When we measure along the X-axis, we consider x to be positive if we are on the RHS of the Y-axis, and x is negative when we are on the LHS of the Y-axis. So we can predict:

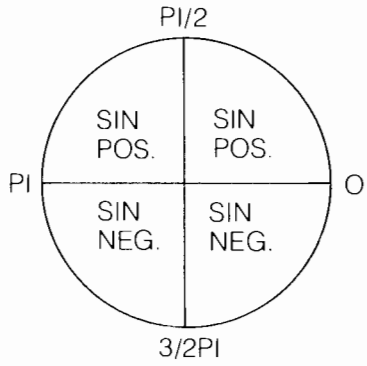
The COS of any angle between 0 and PI/2 is positive.
The COS of any angle between PI/2 and 3*PI/2 is negative.
The COS of any angle between 3*PI/2 and 2*PI is positive.

i.e. For COS:



When we measure along the Y-axis, y is positive when above the X-axis, and negative when below. So we can predict:

The SIN of any angle between 0 and PI is positive.
The SIN of any angle between PI and 2*PI is negative, i.e. for SIN



If we go round the circle more than once, COS and SIN take the same values as the first time round:

```
SIN(a+2*PI) = SIN(a)
COS(a+2*PI) = COS(a)
```

The tangent of an angle is given by:

```
TAN(a)=SIN(a)/COS(a)
```

Given the tangent of an angle, we can find the angle using ATN (which stands for arctan). ATN is the inverse function of TAN:

```
a=ATN(0.6)
```

returns the angle whose TAN is 0.6 into the variable a. The angle is returned in radians.

The computer does not know the inverse functions of COS and SIN, but you can easily define your own functions for these using:

```
ARCSIN(a)=ATN(a/SQR(-a*a+1))
ARCCOS(a)=ATN(a/SQR(-a*a+1))+PI/2
```

Another number often met in mathematics is represented by the letter e.

To have a look at the first 14 digits of this number, type:

```
PRINT EXP(1)
```

You will see:

```
2.7182818284588
```

The function **EXP** is defined by:

```
EXP(x)=e ^ x
```

so:

```
EXP(1)=e
```

The inverse of this function is given by the **LOG** function. This function gives you the log to base e of the argument, i.e. it gives you the natural logarithm of a number which is often written ln.

It is related to the EXP function as shown below:

```
EXP(x)=e ^ x=n
LOG(n)=x
```

You might be more used to taking logs to the base 10. To get log to the base 10, use the following identity:

```
log10(x)=LOG(x)/LOG(10)
```

EXERCISES

1. Define a function called PI, which returns the value of PI. Use the identity:

   PI=ATN(1)*4

2. Define a function which converts degrees to radians. Use the formula:

   a radians = (a*PI)/180 degrees

   You will have to use 3.14... for PI. Alternatively, use the function PI.
3. Define a function which gives the ARCCOS of its argument. Do not forget, $-1 <= COS(a) <= 1$, for any angle of a radians. So your function will only work for arguments between + and −1 inclusive.
4. Define a function which returns the logarithm to base 10 of its argument.

CHAPTER 20

THE ASCII CODES

In CHAPTER 6, you discovered that the letters A-Z and a-z are stored inside the computer as the numbers 65-90 and 97-122 respectively. These numbers are called ASCII Codes.

ASCII stands for the American Standard Code for Information Interchange, and is commonly used by computers to represent characters.

If you want to find the ASCII Code of a character, you may use the Command ASC. For example:

```
10  A1=ASC("A"):A2=ASC("a")
20  PRINT "The ASCII Code of A is ";A1
30  PRINT "The ASCII Code of a is ";A2
```

The argument of the ASC function must be a string; if more than one character is contained in the string, the computer returns the ASCII code of the first character.

The opposite function to ASC is CHR$. When given a suitable numeric argument, CHR$ returns the corresponding ASCII CHaRacter, into a string. For example:

```
10  A1$=CHR$(65):A2$=CHR$(97)
20  PRINT"The Character corresponding to ASCII Code 65
    is ",A1$
30  PRINT"The Character corresponding to ASCII Code 97
    is ";A2$
```

The other characters on the keyboard also have ASCII Codes. Find the ASCII Codes of "%", "?" and "7".

In fact there are 256 ASCII codes altogether, lying between 0 and 255.

You can use CHR$ to find the characters associated with the codes between 32 and 255 inclusive. RUN the following program to find these characters:

```
10  REM ASCII characters: 32-255
20  CLS
30  FOR I=32 to 255
40  PRINT CHR$(I)+" ";
50  NEXT I
```

You will see all the alphanumeric and code characters displayed on the screen, with most of the Graphics Characters. Notice that the last character, which is represented by the code 255, is in fact the cursor; and so changes as you move the cursor over text on the screen!

The codes between 0 and 31 are rather special.

These codes are called Control Codes. Instead of representing a character, they represent an operation, which is to be carried out by the computer when the code is used as an argument in a PRINT CHR$ statement.

You have already met Control Code 13 in CHAPTER 8. Taking PRINT CHR$ of this was equivalent to pressing <RETURN>. This is the only way <RETURN> can be embedded within a program.

Other Control Codes move the cursor around, clear the screen etc.

Try RUNning the next program:

```
10  REM some ASCII Control Codes
20  PRINT CHR$(12)+"Control Code 12 Clears the Screen."
30  PRINT CHR$(7)+"Control Code 7 BEEPs!"
```

The codes 0 to 31 are also used to represent the rest of the Graphics Characters. To access these, however, you must PRINT CHR$(1) first, and then CHR$(A+64) where A is the code number. The next program PRINTs out the first 32 Graphics Characters:

```
10  REM Codes 0—31: more Graphics Characters
20  FOR A=0 TO 31
30  PRINT CHR$(1)+CHR$(A+64)+" ";
40  NEXT A
```

The Keyword, STRING$, is used to PRINT a string of a specified length and made up of a specified character. The string cannot exceed 200 characters, unless you first use a CLEAR statement.

STRING$ must be followed by two arguments, separated by a comma, and enclosed in brackets.

The first argument must be a number, numeric expression or Numeric Variable. This specifies the length of the string.

The second argument may be either a number or a string. If a number is used, this must be the ASCII Code of the character to be PRINTed in the string. For example:

```
10  FOR B=1 TO 10
20  A$=STRING$(B,42)
30  PRINT A$
40  NEXT B
```

gives the pattern:

```
*
**
***
****
*****
******
*******
********
*********
**********
```

If the second argument is a string, then the first character of this string is used, i.e. if line 20 were to be replaced by:

```
20  PRINT STRING$(B,"*")
```

the effect would be the same.

CHAPTER 21

THE SCREEN MODES

So far you have only used the Text Screens, Screens 0 and 1. There are in fact two more screens, screens 2 and 3. We will be using these in the next few Chapters. First of all, a summary of a few of the characteristics of each screen.

The Text Screens

You can alter the width of both these screens by using the KEYWORD WIDTH (see CHAPTER 9):

SCREEN MODE	DEFAULT WIDTH	MAXIMUM WIDTH	CHR SIZE
SCREEN 0	37	40	6x8
SCREEN 1	29	32	8x8

Both screens hold 24 lines of text. Although you can fit more characters across Screen 0 than Screen 1, the space allocated for each character is smaller. Only 6x8 points, or pixels as they are usually called, are used to form each character in Screen 0, whereas 8x8 are used in Screen 1. This means that the alphanumeric characters appear more compressed in Screen 0 than on Screen 1, as the last two pixel columns in the 8x8 pixel squares, used in Screen 1, are left blank, and it is these two columns which are missing from the character positions in Screen 0. The compression of the character positions is more obvious when comparing the graphics characters.

On turning on the computer, one of these text screens is the default screen.

The MSX is capable of producing 16 colours (see CHAPTER 22 for details). In Screen 1 you can set the foreground (text), background and

border colours to any combination of these 16 colours. You cannot have more than one foreground colour at any one time. The default colours are:

FOREGROUND	WHITE
BACKGROUND	DARK BLUE
BORDER	CYAN

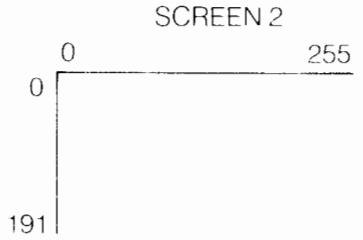
In Screen 0 you can only set the foreground and background colours, the default colours being the same as above.

If at any point you change the foreground colour of a text screen, all the text on the screen changes colour.

The High Resolution Graphics Screen

Screen 2 is the high resolution Graphics Screen.

It is 256 pixels wide, by 192 long. Each pixel can be resolved on this screen. The top left hand pixel is at co-ordinate position (0,0), the bottom right hand pixel at (255,191):



You cannot alter the width of this screen. If you have a WIDTH statement while in Screen 2, the computer will remember it, and on returning to a TEXT Screen will set the WIDTH as specified while in Screen 2.

You may have typed:

```
SCREEN 2
```

while in Command Mode and been surprised that nothing has changed. This is because you cannot access the graphics screens from Command Mode. You must include the SCREEN Command in a program in order to view the Graphics Screens, i.e.

```
10 SCREEN 2
20 GOTO 20
```

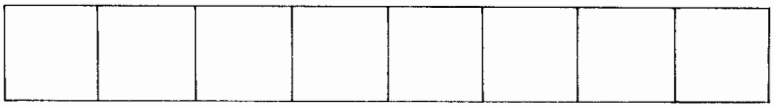
Line 20 freezes the Graphics Screen; this is necessary as the computer immediately returns to the default Text Mode on completing

execution of a program. It will also return to the Text Screen on encountering an INPUT statement in a program, or an Error.

To stop this program and return to the Text Screen, press <CNTL><STOP>.

The pixels in this screen are grouped together in rows 8 pixels long. The first pixel row lies between co-ordinates (0,0) and (8,0):

One 8x1 Pixel Row



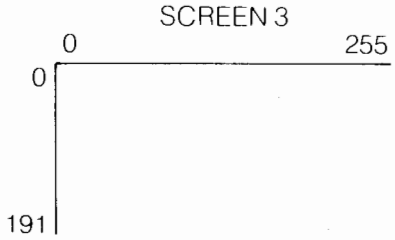
Each row of 8 pixels can only contain two colours, a foreground colour and a background colour. Adjacent rows may contain two different colours. The colour resolution of this screen is therefore 32x192.

To PRINT characters on this screen, see the ADVANCED SECTION, CHAPTER 14. You cannot use the PRINT Command directly.

The Multi-Colour Screen

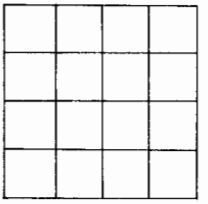
Screen 3 is the Multi-Colour Screen and can be used for low resolution graphics.

Like Screen 2, it is 256 pixels wide by 192 deep, but you can't resolve each pixel on this screen:



The smallest point you can plot on this screen is a block of 4x4 pixels. You can still plot point (0,0), but you will find a block has appeared on the screen whch also covers points (1,0), (2,0), (3,0) to (3,3). So plotting (3,3) would have the same effect.

One 4x4 Pixel Block



Each 4x4 block can only contain one colour, the foreground colour. Therefore the colour resolution of this screen is 64x48.

Like Screen 2, you cannot access this screen from Command Mode; you will have to freeze the screen by using an infinite loop. An INPUT Command or Error will force you back to the default Text Screen.

CHAPTER 22

COLOUR

On switching on the computer, the colours are preset to give you a dark blue background with a cyan border. The foreground, or text, is white.

You can change any or all of these colours by using the COLOR Command. Note the American spelling of the word!

After receiving the command COLOR, the computer expects up to three numbers, separated by commas. The first number sets the foreground colour, the second the background and the last the border. There are 16 colours for you to use:

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

so the default colour is:

COLOR 15,4,7

The Function Key, F6, is predefined to print COLOR 15,4,7 on the screen, followed by <RETURN>. So if you accidently set the foreground colour to the background colour, press this key!

Key F1 is set to print COLOR on the screen.

You do not have to type in all three numbers for the foreground, background and border when using COLOR. Just give a colour code for the one you want to change. For example:

COLOR 1

This sets the foreground to black, leaving the background and border as they were.

COLOR,15,15

The background and border are changed to white, leaving the foreground black. Notice that, although the foreground colour has not been specified, the comma following it must not be left out, as this tells the computer that the next number sets the background.

COLOR,,10

The above command sets the border to dark yellow. You must be in Screen 1,2 or 3 to observe this, as Screen 0 doesn't have a border.

If you change the background colour while in either of the graphics screens, you will have to execute the Command CLS to observe the efect. This clears the screen to the new background colour.

The next program displays all the border and background colour combinations available on the MSX. Type this in on Screen 1:

```
10 REM colours
20 FOR BDR=0 TO 15
30 COLOR,,BDR
40 FOR BACKGD=0 TO 15
50 COLOR,BACKGD
60 FOR DELAY=1 TO 200 :NEXT DELAY
70 NEXT BACKGD
80 NEXT BDR
90 COLOR 15,4,7
```

The DELAY loop in line 60 gives you time to see the different colour combinations; without this line, the colours would flash by too quickly to be seen properly.

Notice that you can use Numeric Variables and Numeric Expressions in the COLOR statement. Numeric expressions, however, won't work here. If you use a real number in a COLOR statement, it will be truncated

to the nearest integer. The number must be in the range of 0 to 15.

Try running this program on the other screens. You will need to insert a CLS Command for Screens 2 and 3.

Have you realised what COLOR 0 does?

It sets the background or foreground to the same colour as the border. If used to set the border colour, the border becomes black.

EXERCISE

1. Set one of the Function Keys to give a different foreground, background and border combination.

CHAPTER 23

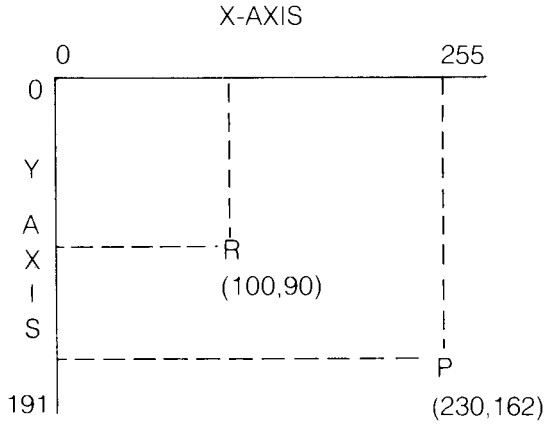
PLOTTING POINTS

The first two commands dealt with in this Chapter, PSET and PRESET, are very similar. As they are Graphics Commands, they can only be used in Screens 2 and 3.

PRESET can be used to set a point to the background colour. The co-ordinates of the point can be either absolute or relative.

After using PRESET, you will find that the current foreground colour is the same as the background.

In both Screens 2 and 3, the x co-ordinate must be in the range of 0 to 25, the y co-ordinate between 0 and 191. Absolute co-ordinates are read directly off the X and Y axes:



The point R is at the absolute x co-ordinate 100, and y co-ordinate 90. P is at absolute co-ordinates (230,162)

A relative co-ordinate gives you the position of a point as

measured from some reference point. For example, the point P is at relative co-ordinates (130,72), measured with respect to R. This means it is 130 points further along the X axis than R, and 72 points lower down the Y axis.

In all the Graphics statements, the reference point is always taken to be the last position of the Graphics Cursor.

The position of the Graphics Cursor can be set by PRESET:

```
10  SCREEN 2
20  PRESET (100,100)
30  GOTO 30
```

If you run this program, nothing appears to happen! However, the Graphics Cursor is now positioned at the point (100,100). Any Graphics command immediately following this, using relative co-ordinates, will now use the point (100,100) as its reference point.

If the next Graphics statement draws a line, for example, you must specify the colour of the line, or reset the foreground colour first, with the COLOR statement. Otherwise your line will be invisible!

If you specify a colour in a PRESET statement, a point of this colour will be plotted, and this colour becomes the current foreground colour. Replace line 20 by:

```
20  PRESET (100,100),1
```

The number following the PRESET statement must be one of the COLOR Codes as given in CHAPTER 22, i.e.a number between 0 and 15. If a real number is used in this range, the decimal part will be ignored.

If your co-ordinates are off the screen, obviously you will not see anything! If the co-ordinates are out of the allowed integer range – 32768 to 32767 you will get an Error Message.

The command **PSET** plots a point of the foreground colour at the specified co-ordinates. These co-ordinates can be either absolute or relative.

If a colour is specified in the statement, then a point of this colour is plotted, so this is identical to using **PRESET** with a specified colour. As with PRESET, this statement also changes the foreground colour to the specified colour.

The following program plots blocks of colour, at random, onto Screen 3, using the command PSET. These blocks are then blotted out by resetting the same random points to the background colour using PRESET:

```
10 REM spots
20 R=TIME
30 RDRST=RND(-R)
40 SCREEN 3
50 REM plot coloured points at random
60 FOR I%=1 TO 300
70 GOSUB 1000
80 PSET(X%,Y%),Z%
90 NEXT I%
100 REM reset points to background colour
110 RDRST=RND(-R)
120 FOR J%=1 TO 300
130 GOSUB 1000
140 PRESET(X%,Y%)
150 NEXT J%
990 REM subroutine randomise
1000 X%=RND(1)*256
1010 Y%=RND(1)*192
1020 Z%=RND(1)*16
1030 RETURN
```

Integer Variables are used throughout this program, to speed execution up, i.e. X% is used instead of X.

Each time the program is RUN, the points will be plotted at different random positions. This is because the point at which the sequence of random numbers is entered is set by the Variable TIME, and this variable will have a different value each time the program is RUN. The same random numbers are generated in the second half of the program, as in the first, so all the plotted points are blotted out. To generate the same numbers, the random number sequence is reset by using the same negative argument in the RND function in line 110 as was used previously in line 30.

Finally, the command **POINT**. This is another Graphics Command. It returns the colour code of a point, in either of the Graphics Screens. The number returned is therefore in the range 0 to 15, representing the colours Transparent to White. −1 is returned if the co-ordinates in the POINT statement are off the screen. Type:

```
10  COLOR ,2
20  SCREEN 2
30  P=POINT(100,100)
```

Now, back in Command Mode, type:

```
PRINT P
```

The number 2 will be displayed, as this is the colour code for mid-green, which is the new background colour.

EXERCISE

1. Try plotting a SINe curve, using:

   Y=A*SIN(X)+A

A sets the amplitude of the curve. Vary X over the range 0 to 2*PI.

CHAPTER 24

DRAWING LINES AND BOXES

This Chapter deals with the command LINE. This is another of the Graphics Commands and so can only be used in Screens 2 and 3.

Drawing Lines

When drawing a line, you may specify the start and end positions, which can be done using either absolute or relative co-ordinates, or even a mixture of the two. Type this program in:

```
10  SCREEN 2
20  LINE (0,0)-(255,191)
30  GOTO 30
```

This program draws a LINE diagonally across the screen. Notice the "-" sign between the co-ordinate pairs.

Absolute co-ordinates have been used. The first pair of co-ordinates, (0,0), gives the starting point of the LINE; (255,191) gives the end position of the LINE.

Change line 10 of this program, so that it Runs in Screen 3, to compare the resolution of the two Graphics Screens.

Try using different co-ordinates in the LINE Command.

If your x or y co-ordinate is greater than 255 or 191 respectively, the computer draws to the edge of the screen, taking x as 255 and y as 191. If, however, your co-ordinates are out of the permitted integer range, -32768 to 32767, you get an Error Message.

You do not have to specify the starting point of the LINE. If this is unspecified, the computer starts drawing from the present position of

the Graphics Cursor. If you have just drawn a LINE, the Cursor will be at the end position of the LINE.

You could have used relative co-ordinates to draw the same LINE. Replace line 20 of the program with:

```
20 LINE (0,0)-STEP(255,191)
```

The statement STEP tells the computer that the following co-ordinates are relative. So, in this case the final x position will be 255 points further along the X axis than the starting point, and the final y position 191 points further down the Y axis putting the last point of the LINE at absolute co-ordinates (255,191).

Coloured Lines

So far you have only drawn lines in the current foreground colour. It is possible to specify the colour of the LINE, however, by following the LINE Command by a comma and then a valid COLOR Code, i.e. a number between 0 and 15. See CHAPTER 22 for the COLOR Codes. Now replace line 20 of the last program with:

```
20  LINE -(255,191),10
```

This draws a diagonal yellow LINE across the screen.

The next program demonstrates the colour resolution on Screen 2.

```
10  SCREEN 2
20  FOR A=0 TO 15
30  LINE(0+A,0)-(200+A,191),A
40  NEXT A
50  GOTO 50
```

You will see 16 diagonal lines drawn across the screen but their colours are rather blurred. If you now replace Line 30 by:

```
30  LINE(0,0+A)-(255,0+A),A
```

You will be able to resolve the colour of each line.

You get these effects because of the way the screen is stored in the computer's memory. Each horizontal line is divided up into groups, 8 pixels long. Each group may only contain two colours at any one time, a foreground and a background colour. On plotting the first diagonal line, the foreground colour is set to 0 and the line is drawn in this colour. The next line sets the foreground colour to 1, but this changes the foreground for the complete group of 8 pixels, so the first pixel of the first line changes colour to black. The 8th line plotted changes the foreground colour of the first group of 8 pixels for the last time; this complete group is now cyan.

The vertical colour resolution is somewhat better than the horizontal

resolution. The colour of each vertical pixel is independent of the pixels above and below, hence the 16 horizontal lines do not interfere with each other.

Try running the two programs on Screen 3.

On this screen, you will see the same four lines each time, as the horizontal and vertical colour resolutions are the same. You can have one colour for every 4x4 pixel block. The colour resolution is therefore the same as the graphics resolution. The first four lines are all plotted on top of each other, so you only see the last one which is light green.

Drawing Boxes

It is possible to draw a box using four LINE Commands, one after the other. However, there is an easier way to draw boxes if you know the co-ordinates of the top left and bottom right hand corners. You still use the LINE Command, but now the first co-ordinate pair gives the co-ordinates of the top, left corner and the second, of the bottom right hand corner. To let the computer know you want a box and not a line, you must follow the COLOR Code by another comma and then the letter "B" for Box. Type in the following program:

```
10  REM Boxes
20  SCREEN 2
30  FOR A=0 TO 90 STEP 10
40  LINE (5+A,5+A)-(255-A,185-A),A/10,B
50  NEXT A
60  GOTO 60
```

This program will draw 10 boxes of different sizes and colours.

Notice that you can use an expression for the COLOR Code, but this expression must evaluate to a number between 0 and 15. If it evaluates to a real number, the decimal part will be truncated.

See the effect of running this program on Screen 3 by changing line 20 to:

```
20  SCREEN 3
```

If you know the lengths of the sides of the box, rather than the co-ordinates of its corners, it is easier to use relative co-ordinates for the second co-ordinate pair, i.e. to draw a box 100 pixels long in the X direction, and 50 pixels deep in the Y, you could use:

```
10  SCREEN 2
20  LINE (10,10)-STEP(100,50),,B
30  GOTO 30
```

The program draws a box of the correct dimensions in the current

foreground colour. The top right hand corner is situated at absolute co-ordinates (10,10).

The second co-ordinates pair are now specifying the lengths of the sides of the box which are parallel to the X and Y axes.

This is the best method to draw squares. For example:

```
10  REM squares
20  SCREEN 2
30  FOR A=10 TO 90
30  LINE (A,A)-STEP(A,A),,B
40  NEXT A
50  GOTO 50
```

The last program draws squares, of increasing size, diagonally across the screen.

Coloured Boxes

To draw a box and fill it with a specified colour, replace the letter "B" by "BF". The colour specified in the LINE statement is now used to paint the box:

```
10  REM black box
20  SCREEN 2
30  LINE (20,20)-(100,100),1,BF
40  GOTO 40
```

EXERCISE

1. Write a program to draw a box of a random size, at random, on the screen. Fill this box with a random colour.

CHAPTER 25

DRAWING CIRCLES AND ELLIPSES

The CIRCLE Command enables you to draw the whole, or part, of either a circle or an ellipse. You must specify the position of the centre of the CIRCLE, in either absolute or relative co-ordinates, and the length of the radius. You have the option of specifying the colour of the CIRCLE and how much of the CIRCLE is to be drawn. CIRCLE is a graphics Command and can only be used on the Graphics Screens.

Drawing Circles

To draw a CIRCLE, you must specify the central co-ordinates of the CIRCLE, and the length of the radius. For example:

```
10  REM drawing circles
20  SCREEN 2
30  CIRCLE (128,100),90
40  GOTO 40
```

This draws a CIRCLE centred at absolute co-ordinates (128,100), with a radius of 90 pixels, in the current foreground colour. Obviously, if you specify a radius which is too large for the screen, you won't see your CIRCLE, or you will just see the part which fits on the screen.

To specify the colour of the circle, follow the radius by a comma, then a valid Colour Code, i.e. a number between 0 and 15. (See CHAPTER 22 for the Colour Codes.) The next program draws the same CIRCLE, but in black:

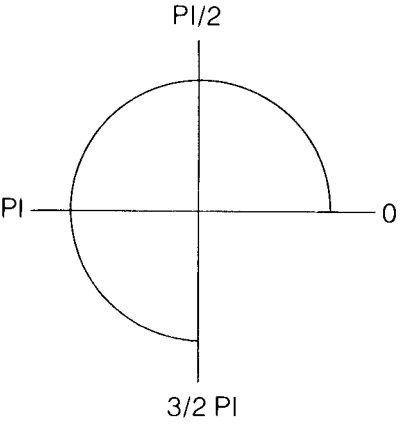
```
10  SCREEN 2
20  CIRCLE (196, 100),90,1
30  GOTO 30
```

Drawing Arcs

To draw just part of a CIRCLE, or an arc, you must specify the start and end angles. The next program draws three quarters of a black CIRCLE:

```
10  REM drawing arcs
20  SCREEN 2
30  PI=4*ATN(1)
40  CIRCLE (128,100),90,1,0,3*PI/2
50  GOTO 50
```

The last two numbers we have added give the start and end angles for the arc. The angles must be specified in radians. (If you cannot remember what a radian is, See CHAPTER 19.) Line 30 only calculates PI, so the arc is drawn between the angles 0 and 3/2 PI radians:



All angles are measured from the X axis, in the anti-clockwise sense. As there are 2*PI radians in a CIRCLE, the specified angles can be anything between 0 and 2*PI.

Drawing Pies

To do this put a "−" sign prefix before the start and end angles. This will result in lines being drawn from the centre of the circle to the start and end positions of the arc. RUN the following program:

```
10  REM drawing pies
20  SCREEN 2
30  PI=4*ATN(1)
40  CIRCLE (128,100),90,,-PI/2,-PI
50  GOTO 50
```

You should see the top left hand sector of the CIRCLE, drawn in the foreground colour.

Drawing Ellipses

To draw an ellipse, you must specify the aspect ratio. This is defined as:

$$\text{ASPECT RATIO} = \frac{\text{VERTICAL RADIUS}}{\text{HORIZONTAL RADIUS}}$$

If the aspect ratio is greater than 1, then the ellipse will be elongated vertically. The specified radius is taken to be the vertical radius.

If the aspect ratio is less than 1, then the ellipse will be elongated horizontally, and the specified radius is taken to be the horizontal radius.

The next program draws a complete, black ellipse, as the start and end angles have not been specified:

```
10  REM drawing ellipses
20  SCREEN 2
30  CIRCLE (128, 100), 90,1,,,1.5
40  GOTO 40
```

As the aspect ratio is 1.5, the ellipse is elongated vertically, and the vertical radius is 90 pixels.

Now replace line 30 with:

```
30  CIRCLE (128, 100), 90,,,0.5
```

You will see a horizontally elongated ellipse, with a horizontal radius of 90 pixels.

Try aspect ratios 100 and 0.01. With these aspect ratios, the ellipses will appear as vertical and horizontal lines respectively, of length 180 pixels.

The following program draws different coloured ellipses, with random aspect ratios, all positioned at the centre of the screen:

```
10  REM elliptical patterns
20  SCREEN 2
30  FOR B=1 TO 15
40  A=RND(- TIME)*2
50  CIRCLE (128,96),90,B,,,A
60  NEXT B
70  GOTO 70
```

If you wish, you can draw part of an ellipse, by specifying the start and end angles, just as you did when drawing circles. The "–" sign prefix can also be used:

```
10  SCREEN 3
20  PI=4*ATN(1)
30  FOR B=1 TO 15
40  A=B/10
50  SA=RND(-TIME)*2*PI
60  EA=RND(-TIME)*2*PI
70  CIRCLE (126,96),90,B,SA,EA,A
80  NEXT B
90  GOTO 90
```

The last program draws fan shapes, centred at (126, 96). The Start and End Angles are calculated using the RND function. Notice that the End Angle can be smaller than the Start Angle. The Aspect ratio is varied over the range 0.1 to 1.5. RUN the program in Screen 2, to compare the resolution.

EXERCISES

1 Write a program to draw 10 small circles or ellipses on the screen at random.
2 Change your program so your circles all have different radii.
3 Draw a circle made up of four sectors, each sector being drawn in a different colour.

CHAPTER 26

THE GRAPHICS MACRO LANGUAGE

The Graphics Macro Language simulates the motion of a pen on paper, and enables you to DRAW complicated pictures easily. Using the command DRAW, you may move the Graphics Cursor to any point on the screen, with the "pen" in either the "up" or "down" position.

The command DRAW must be followed by a string. This string contains the Graphics Macro Commands, which are all represented by single characters.

To DRAW a line in one of the four directions up, down, right or left, the following Graphics Macro Commands are used:

U	DRAWs Up
D	DRAWs Down
L	DRAWs Left
R	DRAWs Right

You must specify the length of the line after the command. This length is measured in pixels.

The next program demonstrates how to use the Macro Language:

```
10  REM square
20  SCREEN 2
30  PSET (123,91)
40  DRAW "R10D10L10U10"
50  GOTO 50
```

On seeing line 40, the computer executes the Macro Commands sequentially, i.e. it DRAWs a line to the right, of length 10 pixels, then down for 10 pixels, left 10, and finally up 10, thus completing the square.

Using the next four commands, diagonal lines may be DRAWn:

E	DRAWs diagonally up and right
F	DRAWs diagonally down and right
G	DRAWs diagonally down and left
H	DRAWs diagonally up and left

Again, each of these commands must be followed by a number. The number in this case, however, specifies the number of pixels moved in the x and y directions.

For example, the relative co-ordinates of the last point of the line DRAWn by E10, would be (10,10), taking the first point of the line as the reference point.

This program draws a hexagon:

```
10 REM hexagon
20 SCREEN 2
30 PSET(128,96)
40 DRAW "E20F20D30G20H20U30"
50 GOTO 50
```

To DRAW a line to a specific co-ordinate position on the screen, the M command is used. This command must be followed by the x and y values of the co-ordinate position of the last point of the line. The given x and y values may be either absolute or relative. For example:

```
M40,50
```

will move the graphics Cursor to the absolute co-ordinate position (40,50), while DRAWing.

To specify relative co-ordinates, the prefix + or − is used:

M+40,50	DRAWs to a position +40 pixels along the X-axis and +50, along the Y-axis
M+40,−50	DRAWs to a position +40 pixels along the X-axis but −50, along the Y-axis

Note that the following Commands are equivalent:

M+0,−10 = U10	M+10,−10 = E10
M+0,10 = D10	M+10,10 = F10
M+10,0 = R10	M−10,10 = G10
M−10,0 = L10	M−10,−10 = H10

In the last example program, the command PSET was used to position the Graphics Cursor, before DRAWing. In fact this can be done by using the B prefix before the M command. B stands for Blank. This prefix will cause the Cursor to Move without DRAWing.

```
30  DRAW "BM128,96"
```

could be used within a program to position the Cursor at the centre of the screen.

The B prefix may also be used with other Macro Commands you have met, which is to be expected as these commands are just special cases of the M command.

If you wish to move back to your starting position after DRAWing a line, use the prefix N.

The next program DRAWs a star on the screen:

```
10  REM star
20  SCREEN 2
30  REM move cursor
40  DRAW "BM128,96"
50  REM draw arms
60  DRAW "NU50ND50NR50NL50"
70  DRAW "NE30NF30NG30NH30"
100 GOTO 100
```

Each time an arm is DRAWn, the Cursor moves back to the centre of the star.

You should now be capable of DRAWing fairly complicated shapes. Once you have defined your shape, it is possible to change its orientation on the screen by using the Angle Command, A.

This command rotates the axes of the screen, in the anticlockwise direction, through 0, 90, 180 or 270 degrees, from the normal, default orientation. This in turn, effects the orientation of the direction Commands U, D, L, R, F, E, G, and H.

The command A must be followed by an integer number between 0 and 3, where:

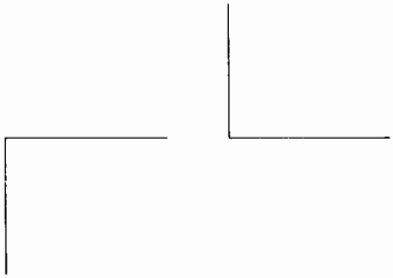
An	rotation
A0	resets axes to the default orientation
A1	rotates default orientation anticlockwise by 90 degrees
A2	rotates default orientation anticlockwise by 180 degrees
A3	rotates default orientation anticlockwise by 270 degrees

Once A is executed, the orientation of all the following direction commands in the program will be rotated by the set angle. To return to the default value, you must use A0 within a DRAW Command.

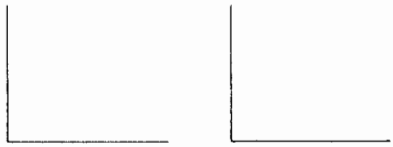
RUN the following program twice:

```
10  REM angle
20  SCREEN 2
30  DRAW "BM50,100NR50ND50"
40  DRAW "A1BM150,100NR50ND50"
50  GOTO 50
```

The first time you will see:



The second time:



This is because the A command is still set to A1. If you wish to repeat the first result, you must reset the Angle of rotation to A0 at the beginning of the string in line 30.

So far all your DRAWings have been in the current foreground colour. To DRAW in a different colour, i.e. to change the foreground colour, you may use the C command. This must be followed by a valid Colour Code. The Colour Codes are the same for this command as for the COLOR Command which you met in CHAPTER 22. So:

C0 Transparent	C8 Medium red
C1 Black	C9 Light red
C2 Medium green	C10 Dark yellow
C3 Light Green	C11 Light yellow
C4 Dark blue	C12 Dark green
C5 Light blue	C13 Magenta
C6 Dark red	C14 Grey
C7 Cyan	C15 White

As with the Angle Command, once C has been set, the current foreground colour will stay this colour until reset by either a C or COLOR Command.

The S Command changes the Scale, or size of your DRAWing. The number, n, following S sets the Scale factor. This number can be any integer between 0 and 255.

The Scale factor (SF) is defined as:

SF = n/4

Therefore, S1 will result in a SF of a 1/4. All the lengths specified by the following U, D, L, R, Commands etc. will be scaled down by 1/4. RUN the next program:

```
10  REM scale
20  SCREEN 2
30  DRAW "BM50,50"
40  DRAW "R150D100L150U100"
50  DRAW "BM50,50"
60  DRAW "S1R150D100L150U100"
70  GOTO 70
```

You will see two rectangles, one a quarter of the size of the other. If you RUN the program again, both rectangles will be DRAWn on top of each other, at the reduced size.

To reset the SF to 1, you may use either S4 or S0. Insert one of these at the beginning of the string in line 40 and RUN the program again.

If you wish to DRAW the same shape more than once in a program, it is advisable to put the string containing the necessary Macro Commands into a String Variable. The next program DRAWs four diamonds on the screen:

```
10 REM diamonds
20 SCREEN 2
30 A$="F40G40H40E40"
40 DRAW"BM50,8"
50 DRAW A$
60 DRAW"BM200,8"
70 DRAW A$
80 DRAW"BM50,104"
90 DRAW A$
100 DRAW"BM200,104"
110 DRAW A$
120 GOTO 120
```

The program can be considerably shortened by putting more than one command in each DRAW statement. To put a String Variable within a DRAW string, the X command must be used. The next program again

DRAWs four diamonds, but this time they are different sizes. The String Variable, A$, is used as a sub-string:

```
10 REM scaled diamonds
20 SCREEN 2
30 A$="F40G40H40E40"
40 DRAW"S4BM50,8XA$;"
50 DRAW"S3BM200,8XA$;"
60 DRAW"S2BM50,104XA$;"
70 DRAW"BM200,104XA$;"
100 GOTO 100
```

Notice that the String Variable Name is followed by a semicolon; this must not be omitted.

If you wish to use a Numeric Variable within a DRAW string, perhaps to set the length of a line, you must precede the Variable Name by the "=" sign. Again, the name must be followed by a semicolon. The following program DRAWs 15 boxes and uses the variable I, to set the scale and colour of each box:

```
10  REM increasing boxes
20  SCREEN 2
30  DRAW"BM31,186"
40  FOR X=1 to 15
50  DRAW"S=X;C=X;U48R48D48L48"
60  NEXT X
70  GOTO 70
```

The left hand vertical sides of the boxes have been carefully positioned to lie at the last pixel position of a horizontal group of 8 pixels. This is to prevent smudging between the last, white, vertical line DRAWn and the 14 horizontal lines which define the tops of the other boxes. The smudge effect results if you try to have more than two colours in a horizontal group of 8 pixels. (See CHAPTER 21 for more details.) To see the smudge effect, replace line 30 with:

```
30  DRAW"BM25,186"
```

EXERCISES

1 Use the S and C Commands to DRAW stars at random on the screen, each star a different colour and size.
2 DRAW a house.

CHAPTER 27

PAINTING

The Command PAINT can be used to fill in any area which is completely surrounded by a border line, with a specified colour. The co-ordinates in the PAINT Command must be the co-ordinates of a point within the shape to be PAINTed. They can be either absolute or relative.

The constraints on the allowed PAINT colours are slightly different in the two Graphics Modes.

Screen 2

Before using PAINT, you must first draw the outline of your shape using one of the Graphics Commands, LINE, CIRCLE or DRAW. In this screen, the PAINT colour must be the same as the colour specified in the Graphics Command. For example:

```
10 REM painted parallelogram
20 SCREEN 2
30 PRESET (100,100)
40 LINE -STEP(-50,50),6
50 LINE -STEP(100,0),6
60 LINE -STEP(50,-50),6
70 LINE -STEP(-100,0),6
80 PAINT(110,110),6
90 GOTO 90
```

The co-ordinates (110,110), are inside the parallelogram. The outline of a red parallelogram is drawn, and then filled in with red PAINT. If you try to PAINT the parallelogram with any other colour, the PAINT will overflow and fill the screen.

If the outline had been drawn in the current foreground colour, then it

would not have been necessary to specify a colour in the PAINT statement, because when no colour is specified in this statement the current foreground colour is used. For example.

```
10  REM painted circle
20  SCREEN 2
40  COLOR 10
50  CIRCLE(100,100),50
60  PAINT STEP(0,0)
70  GOTO 70
```

Line 40 sets the foreground colour to yellow. The outline of the CIRCLE is therefore drawn in yellow, and then PAINTed yellow.

Notice that the co-ordinates in the PAINT statement are those of the centre of the CIRCLE.

In Screen 2, care must be taken to avoid smudgy effects when using the PAINT Command. The following program DRAWs two triangles, backing onto each other, to form a square. One of the triangles is then PAINTed yellow, the other black:

```
10 REM triangles V1
20 SCREEN 2
30 COLOR 10
40 PSET(48,50)
50 DRAW"D96R96H96"
60 PAINT(60,70)
70 COLOR 1
80 PSET(70,60)
90 DRAW"R96D96H96"
100 PAINT(70,60)
110 GOTO 90
```

When you RUN this program you will get a zig-zag effect. This is because in Screen 2, each horizontal group of 8 pixels cannot hold more than 2 colours. There is no problem when PAINTing the yellow triangle, the foreground is yellow and the background is blue (the default colour). When the outline of the black triangle is DRAWn, however, the background colour is still blue but the new foreground colour is black. Most of the yellow triangle is unaffected but where a group of 8 pixels lies in both triangles, the foreground colour of the complete group is changed to black. This results in small sections of the yellow triangle being lost to the black triangle, hence the zig-zag effect.

There is a simple remedy to this problem. RUN the next program:

```
10 REM triangles V2
20 SCREEN 2
30 COLOR 10
40 PSET(48,50)
50 DRAW"D96R96U96L96"
60 PAINT(60,70)
70 COLOR 1
80 PSET(48,50)
90 DRAW"R96D96H96"
100 PAINT(70,60)
110 GOTO 110
```

In this program, a yellow square is drawn and PAINTed. A black triangle is then DRAWn and PAINTed on top of this. As the square only contains one colour, there is no problem when DRAWing the black triangle.

Screen 3

In this screen, you can PAINT a shape any colour and then PAINT a border around it:

```
10  REM painting with a border
20  SCREEN 3
30  LINE(20,20)-(100,100),7,B
40  PAINT(50,50),9,7
50  GOTO 50
```

Two numbers are given after the PAINT statement. The first number gives the colour of the PAINT — this may be any colour. The second number gives the colour of the border. The border PAINT must be the colour of the shape's outline. In the above example, the box was originally draw in cyan, so the border colour must also be cyan. If you try to use any other border colour, the PAINT will overflow and fill the screen.

If the outline of the shape was drawn in the current foreground colour, then you must set the border PAINT colour to this foreground colour, if you are specifying a different colour PAINT for the rest of the shape. RUN the next program:

```
10 REM triangle
20 SCREEN 3
30 COLOR 15
40 LINE (20,20)-STEP(200,100)
50 LINE -STEP(-200,0)
60 LINE -(20,20)
70 PAINT (50,80),10,15
80 GOTO 80
```

You will see a triangle drawn in white, which is then PAINTed yellow with a white border. If the border PAINT had not been set to white in the above program, the PAINT would have covered the screen.

If you want a completely white triangle, however, just replace line 70 by:

```
70  PAINT (50,80)
```

EXERCISES

1 Write a program which draws a black box on the screen, using the "BF" option in the LINE statement. Draw another black box next to it, using the "B" option in the LINE statement and PAINT. RUN this program on both Screens 2 and 3.

2 Write a program in Screen 2 which draws multi-coloured concentric CIRCLEs, each CIRCLE being filled with a different colour. Do not forget, to prevent smudgy effects start with the largest CIRCLE first.

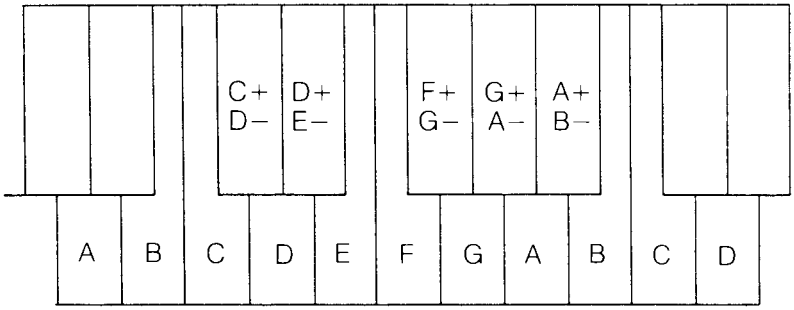
CHAPTER 28

THE MUSIC MACRO LANGUAGE

If you are familiar with standard music notation, the command PLAY will enable you to write, or transcribe music from a manuscript, very easily.

The Music Macro Language is very similar to the Graphics Macro Language, which you have met in CHAPTER 26. The Keyword PLAY is followed by a string containing the various commands allowed in the Music Macro language.

The commands A,B,C,D,E,F and G, play the corresponding musical notes. Any of these commands can be followed by a "+" or "−" sign to indicate whether the corresponding note is sharp or flat. The resultant sharp or flat note, however, must correspond to a black note on the piano; B+, would therefore be invalid.



We have now got access to the notes making up one octave. To select the desired octave, we use the O command. The first note in the octave is C, the last, B. The Octave Command must be followed by a number between 1 and 8. The initial default octave is O4, in which the first note,

C, corresponds to "middle C" on the piano. Using O1 to O8 you have access to every note you can play on a piano.

Once the octave is set, it changes the octave for all the subsequent notes in the program. The following program PLAYs the scale of E major, over two octaves;

```
10  REM E Major: A-G Notation
20  PLAY "EF+G+ABO5C+D+EF+G+ABO6C+D+E"
```

If you RUN this program again, the scale will start off in the 6th octave. To prevent this, add an O4 command at the beginning of the string. Thus line 20 becomes:

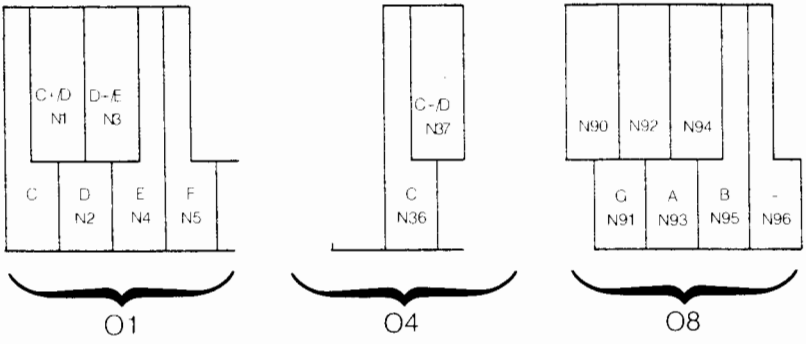
```
20  PLAY "O4EF+G+ABO5C+D+ED+G+ABO6C+D+E"
```

Each note within the 8 available octaves can also be PLAYED using the N command. This command must be followed by a number between 0 and 96 inclusive. As there are 96 notes in 8 octaves, the difference between N52 and N53 for example, is one semitone.

In fact, the 8 octaves available with the N command are shifted up one semitone from those available with the O command. This is because N0, which would correspond to the C in O1, is silent, and can be used as a rest. The lowest note available, using this notation, corresponds to C+ in O1.

The highest note with the O command is B in O8. N96, however, is one semitone higher and plays the next C up from this.

Note that "middle C" is at N36.



The next program plays the same scale of E Major, but this time using the N Command:

```
10  REM E Major: N Notation
20  PLAY"N40N42N44N45N47N49N51"
30  PLAY"N52N54N56N57N59N61N63N64"
```

If you can read music, you will undoubtedly find the A-G notation easier to use.

So far all of our notes have been of the same duration or Length. To change the duration of the following notes, use the Command L. This command must be followed by a number, which can take any value between 1 and 64. The most useful values are listed below;

Ln	Note Duration
L1	plays a whole note or semibreve
L2	plays a half-note or minim
L4	plays a quarter note or crotchet
L8	plays an eighth note or quaver
L16	plays a 16th note or semiquaver
L32	plays a 32th note or demisemiquaver

You can also use L3 etc. L3 would produce a note the third of the Length of a whole note, and can be used for PLAYing triplets.

If you do not set the Length of your note, it is automatically played as a crotchet, i.e. L4 is the default value.

To change the scale of E major, so that it is PLAYed in quavers, you will need to insert L8 at the beginning of the PLAY string. The L command affects all the following notes in the program.

The next program PLAYs E major in quavers, using first the A-G notation, and then the N notation. Notice that the note Length only needs to be set once in the program:

```
10  REM E Major: Quavers
20  PLAY"L8EF+G+ABO5C+D+E"
30  PLAY"N40N42N44N45N47N49N51N52"
```

To set the length of a single note, you may leave out the letter L, and put the number, which specifies the length of the note, after the note to be played:

A+16	is equivalent to L16A+.

Note that you cannot do this with the N notation.

To specify a Rest, the Command R is used. The default value of this command is R4, which is equivalent to a crotchet Rest. To change the

length of the Rest, follow the R command by a number between 1 and 64. This number sets the duration of the rest. So:

Rn	Rest Duration
R1	Rests for a whole note
R3	Rests for a third of a note
R2	Rests for a half-note
R4	Rests for a quarter note
R8	Rests for an eighth of a note
R16	Rests for a 16th of a note
R32	Rests for a 32nd of a note
R64	Rests for a 64th of a note

If you wish to PLAY a "dotted" note or rest, follow the command for that note with a dot or full stop. Each dot after the note results in the duration of the note increasing by half its original length. For example:

```
10  REM dotted notes
20  PLAY"AR64A.R64A.."
```

This PLAYs the note A as a crotchet, followed by a dotted crotchet, and finally a dotted dotted crotchet!

The Rests are included so you can hear the notes individually. Without these Rests, you would just hear the note A PLAYed once.

The Tempo of a piece of music can be set using the T Command. The number which follows the command can take any integer value between 32 and 255, and sets the number of crotchets PLAYed in a minute. You can therefore set the Tempo of a piece of music from a slow 32 crotchets per minute, to an invigorating 255!

The default Tempo is T120.

In the last example of the E Major scale, we changed all the notes to quavers, so the scale was PLAYed twice as quickly as in the previous example. An alternative way to speed the scale up would be to set the Tempo to T240. The notes are now still PLAYed as crotchets, but each crotchet is PLAYed twice as quickly as before:

```
10  REM E Major: Faster Tempo
20  PLAY"T240EF+G+ABO5C+D+E"
30  PLAY"N40N42N44N45N47N49N51N52"
```

The Tempo only needs to be set once in the program.

The Command V, sets the volume. The following number must lie between 0 and 15 inclusive. V0 is very soft or pianissimo; V15 is very loud or fortissimo.

The default value is V8.

Like the other Commands, O, L and T, once this command has been set it will remain at the set value until you either reset it, or turn the computer off.

If a phrase is repeated often in a piece of music, you can assign the string, containing the phrase, to a String Variable. When using the String Variable in a PLAY statement, you must precede the Variable Name by the X Command and follow it with a semicolon.

If you use a Numeric Variable in a PLAY string, this must be preceded by the "=" sign and followed by a semicolon.

Two String Variables, A$ and B$, are used in the next example program. The Tempo is INPUT into the Numeric Variable TEMPO, so you can see the effect of altering this. Try TEMPOs of 120 and 240:

```
10 REM Drink to Me Only: Melody
20 INPUT "Tempo"; TEMPO
30 A$="V8AR64AR64AB-2R64B-05C04B-AGA"
40 B$="B-05C04FB-A2GF2.F2."
50 GOSUB 1000
60 GOSUB 1000
70 PLAY"V905CR64C04A05CF2CR64C04A05CR64C2"
80 PLAY"CD2CR64C04B-AR64A2.G2."
90 GOSUB 1000
100 END
990 REM subroutine
1000 PLAY"T=TEMPO;XA$;"
1010 PLAY"XB$;"
1020 RETURN
```

When typing this program in, do be careful to type the letter O, for the Octave Commands, and not the number 0. Also, take care with the semicolons and the dots.

Notice the length of the notes are all set individually.

The "R64"s are used to separate the sounds when the same note is played twice.

When transcribing this tune, each phrase of music was put into one PLAY statement, to make it easier to check for errors. This can, however, result in rather long program lines; don't forget you can only have up to 255 characters in a line.

The MSX computer can produce up to three sounds simultaneously. We can therefore add two more voices to our tune. The next program shows you how to do this:

```
10 REM Drink to Me Only: 3 Part Harmony
20 INPUT "Tempo";TEMPO
30 A1$="V9AR64AR64AB-2R64B-05C04B-AGA"
40 A2$="V7FEDCDEFEFEF"
50 A3$="V703F2.G2.AGFB-A"
60 B1$="B-05C04FB-A2GF2.F2."
70 B2$="EC2DRC03B-R04CDCDC"
80 B3$="GA02B03C2.F2.C2."
90 GOSUB 1000
100 GOSUB 1000
110 PLAY"V1005CR64C04A05CF2CR64C04A05CR64C2",
    "V8EF2GF2GF2EGF","V8B-F2ED2EF2GA2"
120 PLAY"CD2CR64C04B-AR64A2.G2.",
    "E-DEFR64F2R64FR64F2.E2.",
    "GB-2AG02GAB2.03C2."
130 GOSUB 1000
140 END
990 REM subroutine
1000 PLAY"T=TEMPO;XA1$;","T=TEMPO;XA2$;",
     T=TEMPO;XA3$;"
1010 PLAY "XB1$;",XB2$;","XB3$;"
1020 RETURN
```

The strings associated with the three voices are listed in each of the PLAY statements. The strings must be separated from each other by commas. Notice that the Tempo, Volume etc. are set independently for each voice.

Voice 1 PLAYs at a slightly higher Volume throughout. This is because this voice is PLAYing the melody.

You have probably noticed that by the end of each phrase, the voices are slightly out of synchronisation. This is because the strings for each voice contain different numbers of commands; some strings have a lot more rests and octave changes than others. Each command takes a very small, but finite, execution time. You could try to even up the strings by inserting a few "R64"s and octave changes at suitable places. (Note: An O4 command will do nothing if the string is already in this octave, but it will increase the overall execution time of the string.)

So far all the notes you have PLAYed have sounded very even in tone. It is possible to change this by superimposing an envelope on top of the note; this envelope will control the quality of the note. There are several predefined envelope shapes you can choose from, see SECTION 3: SOUND.

To select an envelope, use the S Command, followed by a number between 0 and 15. The period of the envelope, or modulation, is set by

the M command. This can be set to any value between 1 and 65535; the default value is 255. Type in the following:

```
10  PLAY"S10L1CDE"
```

You will hear the 10th envelope superimposed on the three notes, C, D and E.

Try changing the envelope number. Once you think you have got the idea of this, change the modulation by using the M Command. For example:

```
10  PLAY"M1000S10L1CDE"
```

With the same envelope, try other values of M, i.e. M10, M6000 etc.

Experiment with other values of M and S. The order of the S and M commands is unimportant, but they both must come before the notes to be PLAYed.

Finally the Command BEEP. This is the simplest command to use if you just want the computer to make a noise. Type:

```
BEEP                <RETURN>
```

On executing a BEEP, the computer resets all the Sound Variables. So, after a BEEP, all the Music Macro Commands have been reset to their default values.

SECTION 2

ADVANCED PROGRAMMING GUIDE

CHAPTER 1

ADVANCED PROGRAM EDITING

MSX BASIC has a full screen editor which allows you to edit a program on any part of the screen.

This section goes into the technical details of editing, as well as the advanced editing features of MSX using the control <CTRL> key.

HOW TO EDIT

When writing a program, you often need to write a series of lines, or delete whole sections of your program. MSX BASIC has four very useful commands which can ease the tedium of editing; LIST, AUTO, RENUM and DELETE.

LIST

Once you have entered your program into your computer's memory, you'll want to look at it before you RUN it. To have a look at the program in its entirety, use the LIST command. It will display the program in a clear and logical order; the program can be then edited using the various editing features of MSX.

The LIST command has the following variations which can be used to suit your needs:

LIST	lists the entire program
LIST <line>	lists the line specified
LIST <line1>-<line2>	lists the program from line1 to line2
LIST <line>-	lists from the line specified to the end
LIST -<line>	lists from the begining to the line specified

You will notice that if the program you are listing is fairly long, LIST will

scroll the program until it reaches the last program line, or the last line specified in the LIST command.

There are two methods of getting the relevant section of the program displayed:

A Use LIST <line1>-<line2> so that you only get the section you want.

B Use LIST and when you see the part that you want, press <CTRL><STOP> and break out from the listing.

Note that the <STOP> key only will temporarily halt the listing. If you press the <STOP> key again, listing will continue from where it left off.

AUTO

AUTO command will put you into the automatic line numbering mode, which will offer you a line number every time you press the <RETURN> key. It saves the bother of typing the line number for every line, and makes life a lot easier for the programmer. The line numbers will have a constant increment.

The AUTO command has the following variations which can be used to suite your needs:

AUTO	will give line numbers from 10 in steps of 10.
AUTO <line>	will give line numbers from the specified line number in steps of 10.
AUTO <line>, <increment>	will give line numbers from the specified line number, in steps <increment>.
AUTO ,<increment>	will continue from the line number where you left off, with the specified increment.
AUTO <line>	will give line numbers from the specified line number with the same increment as the previous increment.

Two of the most common ways of using the AUTO command when writing or editing a program are:

1 When starting off a new program which has been already been planned in some form on paper, AUTO is used to get line numbers from 10 in increments of 10. This gives a fairly ordered program.

2 Let's say that you have lines 100, 110, 120, 130, 140, 150, 160 and you want to insert a subroutine between lines 150 and 160. First of all you use RENUM to create more space between 150 and 160. This can be done by RENUM 1000, 150, 100. The result is that you get lines 100, 110, 120, 130, 140, 1000, 1100. Now use AUTO 1010, 5

which will give you a series of line numbers starting from 1010, in increments of 5. The final result will look like this:

100, 110, 120, 130, 140, 1000, <1010, 1015, 1020, 1025, ...>, 1100.

There are some points to remember about AUTO command:

1 There are two ways to exit the AUTO mode:
   a) Press <CTRL><STOP> at the same time.
   b) Press <CTRL><C> at the same time.
   The line from which you exit will not be saved.
2 If there is an existing line number within the program you are editing, the computer will give you a warning signal in the form of a star sign "*" after the line number. If you do not want to ruin that line then press the return key, but if you are replacing that line then ignore the warning "*" and type in what you want.

RENUM

Having typed in your program, invariably you will find that the line numbers are not in fixed increments. It looks messy, and if the program has to be presented to any third party, it will certainly give them a bad impression. The cure is simple: just renumber the whole program using the RENUM command.

RENUM, just by itself, will renumber from 10 in increments of 10. This is the most common form used but you can renumber from a specified line number, in various increments. RENUM will automatically change the line numbers associated with GOTO, GOSUB, RESTORE, THEN, ELSE, ON GOTO, ON GOSUB.

The RENUM command has the following variations which can be used to suit your needs:

RENUM	Renumbers from line number 10 in increments of 10.
RENUM <line1>	Renumbers from the new line number 1 in increments of 10.
RENUM <line1>,<line2>	Renumbers from the old line number 2 starting with the new line number 1 in increments of 10.
RENUM <line1>,<line2> ,<increment>	Renumbers starting with the old number 2 from new number 1, with the given increment.
RENUM ,<line2>, <increment>	Renumbers from the old line number 2, from the new line number 10 in given increment.
RENUM ,<line2>	Renumbers from the old line number 2, from the new line number 10, in increments of 10.

RENUM ,,<increment>	Renumbers from the new line number 10, with the given increment.
RENUM <line1>,, <increment>	Renumbers starting with the new line number 1 from the old line number 10 with the given increment.

RENUM is also used to create enough line number space so that you can insert a section in a program, (such as when used with AUTO, as shown above).

DELETE

Sometimes you'll find a section of a program to be total garbage. If the part you want to delete is a single line, then all you have to do is to type in the line number, then press <RETURN>. However, if you want to delete a whole series of lines, then use the DELETE command. It is simple and effective. It is also permanent, so once you get rid of the lines there is no way of recovery — except to type them in again!

The DELETE command has the following variations which can be used to suit your needs:

DELETE <line>	deletes the line specified.
DELETE <line1>-<line2>	deletes from line1 to line2 inclusive.
DELETE -<line2>	deletes up to line2 inclusive.

CONTROL KEYS AND SPECIAL FUNCTION KEYS

MSX has a number of special function keys and control keys which are used mainly in editing. Here is a table to show you what they do and how to access them. All control keys are accessed by pressing <CTRL> and the relevant key at the same time.

HEX CODE	CONTROL KEY	SPECIAL KEY	FUNCTION
02	* <CTRL><B>		moves cursor left one word.
03	* <CTRL><C>		stops execution when BASIC is waiting for INPUT. Returns to the command level.
05	* <CTRL><E>		deletes everything to the right of the cursor including the character underneath the cursor.
06	* <CTRL><F>		moves cursor to the right one word.
07	* <CTRL><G>		beep.

08	<CTRL><H>	BS	back spaces and deletes a character to the left of the cursor, and then pulls the right hand side of the current line to the left by one.
09	<CTRL><I>	TAB	TAB. Moves to the next TAB position. TAB is set at 8 character intervals. It will place blanks from where it has moved.
0A *	<CTRL><J>		line feed. Physically moves the cursor to the beginning of the next line.
0B *	<CTRL><K>	HOME	moves cursor to the top left of the screen.
0C *	<CTRL><L>	CLS	clears the screen and moves the cursor to the home position.
0D *	<CTRL><M>	RETURN	carriage return. Inputs the current line.
0E *	<CTRL><N>		moves the cursor to the end of the current line.
12 *	<CTRL><R>	INS	goes into the INSERT mode. The cursor becomes half the size and allows you to insert characters to the current cursor position without deleting anything.
15 *	<CTRL><U>		deletes the entire current line.
18 *	<CTRL><X>	SELECT	ignored by current version of MSX.
1B *	<CTRL><[>	ESC	ignored by current version of MSX.
1C *	<CTRL><Y>	→	moves cursor to the right by one.
1D *	<CTRL><]>	←	moves cursor to the left by one.
1E *	<CTRL><^>	↑	moves cursor up by one.
1F *	<CTRL><_>	↓	moves cursor down by one.
7F	<CTRL><DEL>	DEL	deletes the character at the cursor position and pulls the right hand side of the line by one character.

*mark indicates that it will disable INSERT mode if it is ON.

CHAPTER 2

CONSTANTS AND VARIABLES

CONSTANTS

Constants are numbers or strings which do not vary, i.e. 100 and "SPEED" are constants.

There are two types of constants — strings and numerics.

String Constants

A string constant is a sequence of characters up to 255 long. They can be any characters, graphics or control codes which are used in MSX BASIC. They must be enclosed in double quotation marks, e.g.

"LUKE SKYWALKER"
"*---WHAT IS IT?"
"TOM, DICK, AND HARRY"
"$100,000,000"

Numeric Constants

These are positive or negative numbers, and there are six types.

1	Integer constants	Whole numbers between −32768 and 32767 or 1111111111111111 to 0000000000000000 in 16 bit binary. They do not contain decimal points.
2	Fixed point constants	Positive or negative real numbers which sometimes contain a decimal point, e.g. 657.188

3	Floating point constants	Positive or negative real numbers in exponential form. Formats: Single precision <integer>E<integer> −56E10 <fix-point>E<integer> 7.865E5 Double precision <integer>D<integer> −1896243232662D50 <fix-point>D<integer> 7.8827726266265D−5 The accuracy range is between 1E−64 to 1E64.
4	Hexadecimal constants	Hexadecimal numbers denoted by prefix &H. Hex is a number system with base 16, i.e. it uses 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Examples: &HFF (1 byte long) &H8D1A (2 bytes long)
5	Octal constants	Octal numbers denoted by prefix &O. Octal is a number system with base 8, i.e. it uses 0, 1, 2, 3, 4, 5, 6, 7. Example: &O6543
6	Binary constants	Binary numbers denoted by prefix &B. Binary is a number system with base 2, i.e. it uses 0, 1 only. Example: &B01010101

Single Precision and Double Precision

One of the major features of MSX BASIC is that it has 14 digit accuracy double precision BCD arithmetic functions, which most 8 bit computers do not have. Double precision is usually offered in 16 and 31 bit systems but Microsoft has rewritten its 4.5 MS-BASIC to cater for accurate computation.

You can either have single precision or double precision numeric constants but if you do not specify which, the computer will assume double precision.

Accuracy: Single precision — 6 digits
Double precision — 14 digits

Single Precision constants have the following characteristics:

1	Exponential form E	1.65E-2
2	Trailing exclamation mark.	235.77!

Double Precision constants have the following characteristics:

1	Any digit or number without an an exclamation mark or exponential.	1878932.2	156
2	Exponential using D.	562.765333D-06	
3	Trailing # sign	387.001#	

VARIABLES

Variables are names to which you can assign values. These values of a variable can be changed. They can be assigned specifically by the programmer (i.e. PI=3.14) or calculated by the computer (i.e. PI=4*ATN(1)).

Variable Names

Variable names can be any length. However, the computer will only recognise the first two letters of any variable name. If two variables have the same first two letters then they are assumed to be the same, i.e. VAR1 and VAR2 are the same. If you want these to be different, you should use V1 and V2.

The variable names cannot contain any reserved words, i.e. BASIC Keywords, in any part of the name. For example, LENGTH% is wrong because it has LEN in it.

If a variable name starts off with FN, then it is assumed to be a user-defined function.

Type Declaration

If a variable name does not have any type declaration character trailing the variable name, then it is assumed by the computer to be a double precision numeric variable.

A1=<double precision number>

Special trailing characters:

$ A dollar sign indicates that the variable is a string.
Z$="string characters"
ADDRESS$="21B BAKER STREET"

% A percent sign indicates an integer variable.
X%=25
ZXY%=-32768

! An exclamation mark indicates a single precision numeric variable.
F!=5679.7!
SPRES!=4.2888E-02

sign indicates a double precision numeric variable.
H#=3.14161718293
MAXIMUM#=5277.76525D25

DEF<character type>

It is possible to define a character type using various DEF statements. When the type of variable is declared using one of these statements, any variable starting with that declared character will become that character type without declaring it in the variable name. This means that if you DEFSTR A, then any variable starting with A will be a string variable without having the usual dollar sign. However, that doesn't mean that AB% will be a string variable, since the integer declaration character (i.e. %) takes precedence over DEFSTR.

There are four DEF<character type> statements. These are as follows:

DEFSTR statement declares that any variable name beginning with the specified range of characters is to be treated as a string variable:

```
DEFSTR A
AST="PLANET"
```

DEFINT statement declares that any variable name beginning with the specified range of characters is to be treated as an integer.

DEFSNG statement declares that any variable name beginning with the specified range of characters is to be treated as a single precision number.

DEFDBL statement declares that any variable name beginning with the specified range of characters is to be treated as a double precision number.

Note that the type declaration of the character by #, $, %, and ! takes precedence over these statements.

ARRAY (DIM)

An array is a group of variables which have a common name. It is set up by using the DIM statement. DIM statements set aside a section of memory for the use of any array before it is used. Each element of the array has a subscript, or number label. Say, for instance, you define DIM C(3). You then get four variables with the name C( ), i.e. C(0), C(1), C(2), C(3). All array variables start with the zeroth element. This is a one dimensional array, but there is no reason why you can't have a multi-dimensional array such as DIM A(2,2) which gives a table of 3 by 3 variables as shown below:

```
A(0,0) A(1,0) A(2,0)
A(0,1) A(1,1) A(2,1)
A(0,2) A(1,2) A(2,2)
```

This is a two dimensional array but you can have a three dimensional array, or even ten. The maximum dimension for an array is 255 but you are likely to run out of memory if you try to create such a massive array.

If an array has less than 12 elements, i.e. DIM A(10), then there is no need to execute a DIM statement, as the computer automatically sets aside 11 elements of memory space. Programs such as the one shown below are perfectly legal even without the DIM statement.

```
10 FOR I=0 TO 10
20 S(I)=I
30 NEXT I
```

It is possible to have arrays of any type as long as the type is declared. For example, DIM S$(100) will give you one hundred and one elements of a string array.

When the array is initialised, all the values are assumed to be zero for numeric arrays, and null strings for string arrays.

Memory Requirement

Variables are stored in either DIM AREA, VARIABLE AREA or STRING AREA depending on the type (see memory map).

Here is a list of the number of bytes used by these variables.

Variables	Integer	2 bytes
	Single precision	4 bytes
	Double precision	8 bytes
	String	3 plus length of content
Arrays	Integer	2 bytes per element
	Single precision	4 bytes per element
	Double precision	8 bytes per element
	String	3 plus length of content per element

VARPTR

To find out exactly where the data of a variable is stored in memory, you use a special function VARPTR which gives the address of a particular variable, e.g.

```
10 A%=100
20 PRINT VARPTR(A%)
```

You can find out the memory location of any type of variable as long as it exists. It can be a string or even an array.

String Variable Area

All the contents of string variables are stored in the STRING AREA section of the computer's memory (see Memory Map). The size of this area is restricted to 200 bytes at default; therefore the maximum amount of string at any one time is limited to 200 characters. However, you can increase the size of the STRING AREA by using the CLEAR statement. For example, to increase the STRING AREA to 255 bytes, execute:

```
CLEAR 255
```

CHAPTER 3

TYPE CONVERSION

MSX BASIC can convert one type of numeric constant to another. There is a set rule for carrying this out.

1 If a numeric constant of one type is set equal to another type, the number stored will depend entirely upon the type of variable.

Example 1

```
10 C!=1.2345678901
20 PRINT C!
RUN
1.23456
```

The double precision number 1.2345678901 is converted to single precision when assigned to C!, a single precision numeric variable.

Example 2

```
10 I%=1.23E-03
20 PRINT I%
RUN
0
```

The single precision number 1.23E-03 is rounded to the nearest integer which is zero when assigned to an integer variable.

2 During expression evaluation, all variables and constants are converted to the most accurate precision of the variables being used. When the expression is set to equal a variable, the result of the expression is converted to the precision type specified by the variable. If the result variable type is not specified, the result remains at the same precision level as that in which the evaluation took place.

Example 1

```
10 D=1!/3#
20 PRINT D
RUN
0.33333333333333
```

1! is single precision while 3# is a double. The arithmetic was performed in double precision and returned to a double precision variable D.

Example 2

```
10 D=1!/3!
20 PRINT D
RUN
 .333333
```

Both constants are single precision, so the arithmetic was carried out in single precision, but D is a double precision variable so the result is converted to double precision, but lacks the double precision accuracy.

Example 3

```
10 D!=1/3
20 PRINT D!
RUN
.333333
```

The arithmetic was carried out in double precision but the result is converted to single precision, when returned to D!, by truncation.

3 Logical operators convert their constants and variables to 16 bit integer numbers before the operation is carried out. The result is an integer and the numbers involved must be within the integer range or an "Overflow Error" will result.

Example

```
10 D%=156.65 AND 654
20 PRINT D%
RUN
140
```

The numbers are changed to 156 and 654. Result 140 is returned to D% which is an integer variable.

4 When a floating point or a fixed point value is converted to an integer, the fractional part is truncated.

Example 1

```
10 A%=123.456
20 PRINT A%
RUN
123
```

The fixed point value 123.456 is truncated to 123 when assigned to an integer variable A%.

Example 2

```
10 A%=1.0097554D02
20 PRINT A%
RUN
100
```

The floating point number 1.0097554D02 is rounded to 100 when converted to A%, an integer variable.

5 When a single precision number is converted to a double precision number, only 6 digits can be supplied to the double precision result.

Example

```
10 A#=1.33333
20 PRINT A#
RUN
1.33333
```

A# is a double precision variable but that doesn't make the number assigned to it any more accurate — the result is still 1.33333.

TYPE CONVERSION USING MSX BASIC FUNCTIONS

CDBL, CINT, CSNG, VAR, STR$

Although MSX does type conversion automatically when it is evaluating expressions, you may program it to do type conversion, and, when necessary, using one of the type conversion functions. CDBL, CINT, and CSNG follow the same rules as above but VAR and STR$ are string to numeric and numeric to string respectively, and they follow slightly different rules.

CINT

CINT function converts single and double precision numbers, variables, and expressions to integer numbers. The argument must be strictly within the integer range of −32768 to 32767. If not, then an Overflow error will result.

```
CINT(<num-const>)
CINT(<num-var>)
CINT(<num-exp>)
```

Example

```
PRINT CINT(1234.56789)
1234
```

CSNG

CSNG function converts the argument into a single precision number. CSNG rounds off to the nearest 6th decimal place. If an integer is converted, the accuracy will still be the same.

CSNG(<num-const>)
CSNG(<num-var>)
CSNG(<num-exp>)

Example

```
PRINT COS(0.7656)
.72096670541357
PRINT CSNG(COS(0.7656))
.720967
```

CDBL

CDBL function converts integer and single precision numbers, variables and expressions into double precision numbers. When a single precision number is converted, only 6 digits will be supplied to the double precision result.

CDBL(<num-const>)
CDBL(<num-var>)
CDBL(<num-exp>)

Example

```
PRINT CDBL(5.666!)
5.666
```

VAL and STR$

These functions are used for converting strings to numbers and vice versa.

STR$(<numberic>)
STR$ converts the numeric argument into a string.

VAL(<string>)

VAL returns the value of a string whch contains a number. If a string is a string representation of a number then it can be VALed. However, it can't work out an expression within a string. VAL can strip off spaces, tabs, and line feed in front of a number in a string, but can't cope with other characters. VAL can recognise plus and minus signs.

See SECTION 1 CHAPTER 15 for examples.

CHAPTER 4

EXPRESSIONS AND OPERATORS

An expression may be a string, a numeric constant, or a variable, or any kind of combination of these which returns a single value, as long as it is legal. There are four types of mathematical or logical operators.

1 Arithmetic
2 Relational
3 Boolean (Logical)
4 Functional

1 and 2 are covered by this part but see the relevant sections for the others.

ARITHMETIC OPERATORS

The arithmetic operators in order of precedence are:

1	^	exponential	A ^ B
2	–	negation	–A
3	*,/	multiplication and division	A*B, A/B
4	+,–	addition and subtraction	A+B, A–B

To change the order of calculation, use brackets ().
The order of precedence is maintained within the brackets.

DIV and MOD (Integer division)

DIV is denoted by ¥ (yen) sign in MSX BASIC. It performs integer division. Modulus is denoted by MOD.

What are DIV and MOD? Well, here is an example to illustrate their use. In usual division if you divide 10 by 4, you get 2.5.

Similarly,

13/5=2.6

In integer division, however, you don't get the decimal point but instead you get the whole number part of the division, 2, and a remainder, 3.

In MSX BASIC, if you carry out 13 DIV 5 you get

13¥5=2 .....whole number part

If you 13 MOD 5 you get,

13 MOD 5=3 .....remainder

Integer Division and Modulus arithmetic are carried out with 16 bit integers. Before the calculation takes place, any non-integers are changed to integers by truncation, i.e. all fractional parts are discarded.

16.78¥5.89 = 3 (is the same as 16¥5 = 3)

16.78 MOD 5.89 = 1 (is the same as 16 MOD 5 = 1)

MOD follows DIV (¥) in order of precedence.

Example: Showing order of precedence of MOD and DIV:

```
        1580 MOD 789 DIV 10
1                789 DIV 10
                     78
2       1580 MOD     78
             20
```

i.e. 1580 MOD 789 ¥ 10 = 20

RELATIONAL OPERATORS

Relational operators compare two values and return a true (−1) or false (0) answer. They are usually used in IF THEN conditions, but can be incorporated in to an expression.

OPERATOR	RELATION	EXAMPLE
=	Equal to	A=B
<>	Not equal to	A<>B
<	Less than	A<B
>	Greater than	A>B
<=	Less than or equal to	A<=B
>=	Greater than or equal to	A>=B

True returns −1 False returns 0

When there is a mixture of relational operators and arithmetic operators, arithmetic operators take precedence.

Example

A*B=C

A*B is carried out first, then the result of A*B is compared with C.

Relational operators are covered extensively in the IF THEN ELSE section (Section 2 Chapter 7).

CHAPTER 5

SURVEY OF NUMBER SYSTEMS USED IN MSX

Since most human beings on this planet have ten fingers on their hands, they use the decimal number system. Unlike homo sapiens, computers, which are basically a huge array of switches, can have only two 'fingers' — ON or OFF (or in other words, 0 or 1). So computers work in binary, internally, all the time. For the benefit of users, the MSX BASIC will display numbers in decimal but naturally it can also handle binary numbers in users' programs. Apart from binary, MSX can use Octal and Hexadecimal. All hex, binary and octal numbers are integers and must be within the integer range, that is, between −32768 and 32767.

The arithmetic functions are all calculated in Binary Coded Decimal system (BCD) which we will explain later.

BINARY

Definition: Number system with base 2. Uses Ø and 1 to represent all numbers.

MSX BASIC: &B<binary> represents a binary number.
BIN$ decimal to binary

Each digit of a binary number is called a 'bit'. 8 bits of binary are called a 'byte'. One byte is the size of one address of memory in the MSX system. One byte of binary can have any value between 00000000 to 11111111, or 0 to 255 in decimal.

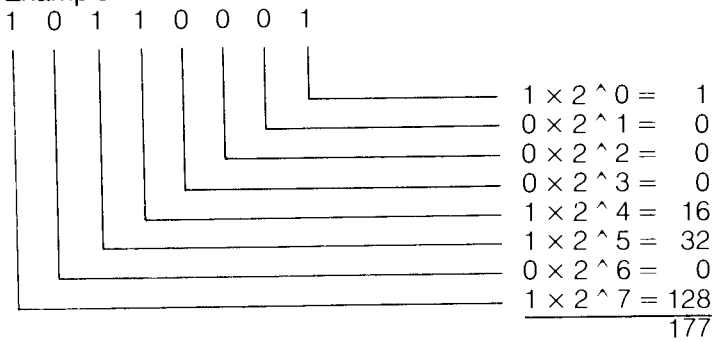
DECIMAL	BINARY	IN 8 BIT
0	0	00000000
1	1	00000001
2	10	00000010
3	11	00000011
4	100	00000100
5	101	00000101
6	110	00000110
7	111	00000111
8	1000	00001000
9	1001	00001001
10	1010	00001010
11	1011	00001011
12	1100	00001100
13	1101	00001101
14	1110	00001110
15	1111	00001111
16	10000	00010000

Binary to decimal and decimal to binary conversion

How to convert 8 bit binary numbers to decimal equivalent.

BIT NUMBER	CORRESPONDING	VALUE
7	2 ^ 7	128
6	2 ^ 6	64
5	2 ^ 5	32
4	2 ^ 4	16
3	2 ^ 3	8
2	2 ^ 2	4
1	2 ^ 1	2
0	2 ^ 0	1

Example: convert 10110001 to decimal.



binary decimal
&B1011001 = 177

How to convert decimal number to binary.
Example: convert 53 to binary

BIT	NUMBER			
7	2 ^ 7	128		0
6	2 ^ 6	64		0
5	2 ^ 5	32	53−32=21	1
4	2 ^ 4	16	21−16=5	1
3	2 ^ 3	8		0
2	2 ^ 2	4	5−4=1	1
1	2 ^ 1	2		0
0	2 ^ 0	1	1−1=0	1

Collect all binary bits and you get 53=&B00110101

In BASIC:

How to convert an 8 bit binary number to its decimal equivalent.

Example

```
10  X%=&B00101011
20  PRINT X%
RUN
43
```

How to convert a decimal number to its binary equivalent.

Example

```
10  X%=171
20  PRINT BIN$(X%)
RUN
10101011
```

OCTAL

Definition: A number system with base 8. This uses numbers between 0 and 7 to represent all numbers.

MSX BASIC: &0<octal> represents an octal number.
OCT**$** decimal to octal

DECIMAL	OCTAL
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17
16	20

Octal to decimal and decimal to octal conversion

How to convert an octal number to decimal equivalent.

<decimal> = <1st digit> *8 ^ 0+<2nd digit> *8 ^ 1+<3rd digit>*8 ^ 2+<4th digit>*8 ^ 3.....

Example Octal &O5436 to decimal

6*8 ^ 0+3*8 ^ 1+4*8 ^ 2+5*8 ^ 3=2846
&O5436=2846

How to convert decimal number to octal.

Example Decimal 6588 to octal

DIGIT	DECIMAL		INTEGER DIV		OCTAL	MODULUS		
4	8 ^ 4	4096	6588 DIV 4096	=	1	6588	MOD 4096	= 2492
3	8 ^ 3	512	2492 DIV 512	=	4	2492	MOD 512	= 444
2	8 ^ 2	64	444 DIV 64	=	6	444	MOD 64	= 60
1	8 ^ 1	8	60 DIV 8	=	7	60	MOD 8	= 4
0	8 ^ 0	1	4 DIV 1	=	4			

So you get &O14674 = 6588 (decimal)

In BASIC

How to convert an octal number to its decimal equivalent.

Example

```
10  X%=&O6517
20  PRINT X%
RUN
3407
```

How to convert a decimal number to its octal equivalent.

Example

```
10  X%=171
20  PRINT OCT$(X%)
RUN
253
```

HEXADECIMAL

Definition: A number system with base 16. It uses numbers between 0 and 9 and additionally A, B, C, D, E, and F to represent all hex numbers.

MSX BASIC: &H<hex> represents a hexadecimal number HEX$ decimal to hexadecimal.

DECIMAL	HEXADECIMAL
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10

Hex to decimal and decimal to hex conversion

How to convert a hex' number to its decimal equivalent.

<decimal> = <1st digit>*16 ^ 0+<2nd digit>*16 ^ 1+<3rd digit>*16 ^ 2+<4th digit>*16 ^ 3.....

Example Hex &HFF to decimal

(&HF)*16 ^ 0+(&HF)*16 ^ 1=(15)*1+(15)*16=255
&HFF=255

How to convert decimal number to its hexadecimal equivalent.

Example: Decimal 7752 to hex

DIGIT	DECIMAL	INTEGER DIV	HEX	MODULUS
3	16 ^ 3 4096	7752 DIV 4096 =	1	7752 MOD 4096 = 3656
2	16 ^ 256	3656 DIV 256 =	14(E)	3656 MOD 256 = 72
1	16 ^ 1 16	72 DIV 16 =	4	72 MOD 16 = 8
0	16 ^ 1	8 DIV 1 =	8	

So you get &H1E48 = 7752 (decimal)

In BASIC

How to convert a hexadecimal number to its decimal equivalent.

Example

```
10  X%=&HEFA3
20  PRINT X%
RUN
4189
```

How to convert a decimal number to its hexadecimal equivalent.

Example

```
10  X%=171
20  PRINT HEX$(X%)
RUN
AB
```

NOTES

1 Binary is often used when designing sprites because is is easy to see the on and off of the bit pattern.
2 Hex is used in describing memory locations and in machine codes.

Summary of Decimal, Binary, Octal and Hexadecimal.

DECIMAL	BINARY	OCTAL	HEXADECIMAL
0	0	0	0
0	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

BINARY CODED DECIMAL (BCD) SYSTEM AND MSX

MSX features up to 14 digit accuracy double precision BCD arithmetic functions. This means arithmetic operations no longer generate strange rounding errors which are invariably associated with many 8 bit computer systems. All arithmetic operations and functions are calculated with this accuracy unless specified otherwise by the user program.

Why use binary coded decimal? What is its advantage and why is it more accurate? To start with, let's have a look at how numbers are worked out in the binary system which is most suitable for computers.

In binary, if you try to express a number smaller than 1 (i.e. one with a decimal point) you sometimes get errors. There are certain numbers which can be expressed in decimal but not in binary.

Say we want to express 0.6525 in binary:

$$0.625 = 1/4+1/8 = 1/(2 \uparrow 2)+1/(2 \uparrow 2)$$
$$0.625 = 2 \uparrow (-2)+2 \uparrow (-3)$$

So you can convert 0.625 to binary without any trouble. However when it comes to a number like 0.1, you can't do this. There is no way of expressing 0.1 in binary exactly. The following shows why:

$$0.1 \sim 1/16+1/32+1/256+1/512+1/4096+1/8192+1/65536 \ldots.$$
$$\sim 0.09999\ldots.$$

or

0.1 ~ 1/(2 ^ 4)+1/(2 ^ 5)+1(2 ^ 8)+1/(2 ^ 9)+1/(2 ^ 13)+1/(2 ^ 16)+ ..... ~ 0.099999....

You can't get a 0.1 in binary; instead you obtain a series.

Instead of turning a decimal into binary completely, MSX uses binary coded decimal.

In the binary coded system, a 4 bit binary represents a digit in decimal. This means that 0001 corresponds to 1, and 0010 corresponds to 2, and so on up to 1001 which is 9. 4 bit binaries between 11 and 15 i.e. 1010, 1011, 1100, 1101, 1110, and 1111, are not used in the binary coded decimal system. Instead 10 (decimal) carries to the next 4 bit — therefore 10 is 0001 0000 in BCD. 1 byte (8 bit) represents two digits in decimal.

0	0000	0000	10	0001	0000	....	90	1001	0000
1	0000	0001	11	0001	0001	....	91	1001	0001
2	0000	0010	12	0001	0010	....	92	1001	0010
3	0000	0011	13	0001	0011	....	93	1001	0011
4	0000	0100	14	0001	0100	....	94	1001	0100
5	0000	0101	15	0001	0101	....	95	1001	0101
6	0000	0110	16	0001	0110	....	96	1001	0110
7	0000	0111	17	0001	0111	....	97	1001	0111
8	0000	1000	18	0001	1000	....	98	1001	1000
9	0000	1001	19	0001	1001	....	99	1001	1001

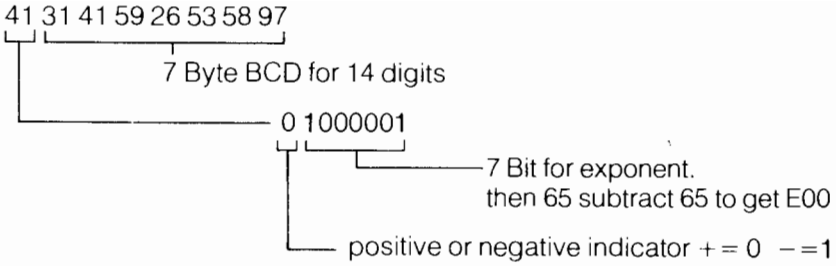
8 byte double precision numbers in BCD

MSX stores numerical data in double precision unless it is programmed to otherwise. It has an accuracy of 14 digits, and is stored in 8 byte binary coded decimal system. The first byte stores the exponent, and the other bytes give the 14 digits, i.e. 2 digits per byte.

To see how the numerical data is stored, first find the memory location of that data using VARPTR function, and then peek the content and display it in hex.

```
10  P=3.1415926534890
20  ADDRESS=VARPTR(PI)
30  FOR I=ADDRESS TO ADDRESS+7
40  PRINT HEX$(PEEK(I)),"  ";
50  NEXT I
60  PRINT
RUN
41 31 41 59 26 53 58 97
OK
```

Let's analyse the above;



The header byte gives the positive or negative sign and the exponent. To get the exponential value, first ignore the 7 bit and calculate the digital value for the rest, i.e. 6th to 0th bits; then subtract 65. This gives a number with size between 1E-64 and 9.99999999999999E62.

Advantages and Disadvantages of the Binary Coded Decimal System

Advantages

1 BCD gives no rounding off errors, and can give numbers which are not possible in binary such as 0.1.

2 It is extremely easy to display a BCD number because there is no need to convert binary to decimal. This speeds up the process of printing to the screen.

Disadvantage

1 Because it is not in ordinary binary, the computer takes more time when it comes to the arithmetic computation. However, the Z 80 CPU has a special intruction DAA, Decimal Adjust Accumulator, to make it easier to process. The DAA converts the accumulator content into packed BCD following add or subtract, with packed BCD operands.

CHAPTER 6

BOOLEAN ALGEBRA
(LOGICAL OPERATORS)

INTRODUCTION

However small your MSX may be, it is still a computer, and a computer is basically made of millions of switches. These can be either ON or OFF, but combinations of these create a working computer. The logic of a computer, or this huge bunch of switches if you like, is governed by a set of rules which come under Boolean Algebra. It is mainly used in multiple condition testings, in IF/THEN statements, with ANDs and ORs, but you can use it in bit manipulation which is important in machine code.

MSX LOGICAL OPERATORS

MSX provides proper Boolean (logical) Operators, unlike some 8 bit micros which seem to miss these out. They are very important in advanced programming. Here is a list of them in order of precedence.

		Computer Jargon
1	NOT	Complement
2	AND	logical AND
3	OR	logical OR
4	XOR	Exclusive OR
5	EQV	Equivalent
6	IMP	Implication

NOT, AND and OR are the important operators since they are the most commonly used logical operators; and besides the others, i.e. XOR, EQV, and IMP, can be derived from combinations of the NOT, AND and OR operators.

Logical operations are always carried out on integer numbers. Any

argument that is not an integer is converted to 16 bit, signed, two's complement integer.

We shall now see how they work, and what they can be used for step by step, starting with NOT.

NOT

NOT is a denial or negation. It gives true for false, and false for true, i.e. 1 for 0 and 0 for 1. In other words, it negates one argument.

Here is the truth table for NOT.

NOT	X	NOT X
	0	1
	1	0

NOT X 100	0000000001100100
−101	1111111110011011

AND

The Boolean algebraic operations of AND can be listed in a truth table as shown below:

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

Therefore, say 100 AND 50 can be calculated as follows:

X	100	0000000001100100
AND Y	50	0000000000110010
	32	0000000000100000

EXAMPLE: USE OF AND

Logical AND can be used for bit testing a particular number. For example, if Y = 2 ^ n, where n is the test bit between 0 and 7, then X AND Y = Y if the test bit n is 1 and X AND Y = 0 if the test bit n is 0.

BIT TEST the 5th bit (n=5) for X=117.

Y=2 ^ 5=32

X	117	0000000001110101
AND Y	32	0000000000010000
	32	0000000000010000

Therefore the 5th bit of number 117 is 1, i.e. true.

OR

When either X, Y or both are true, OR will return true. Logical OR is sometimes called logical sum.

The truth table for OR is:

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

Example 34 OR 67

```
     X  34   0000000000100010
OR   Y  67   0000000001000011
-------------------------------
        99   0000000001100011
```

EXAMPLE: USE OF LOGICAL OR
Logical OR can be used for converting an upper case character to a lower case character, since the difference between the ASCII code of upper case and lower case is constant, i.e. 32 (or 2 ^ 5).

Example: change character A (ASCII code 65) to a (ASCII code 97).

```
     A    65   0000000001000001
OR 2^5    32   0000000000100000
-------------------------------
     a    97   0000000011000001
```

This method is quite flexible, because if you tried to convert a lower case letter to a lower case letter, nothing happens. Try this for yourself.

XOR

XOR, Exclusive OR returns true (1) when X and Y are different.

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

EXAMPLE: 100 XOR 50

```
      X  100   0000000001100100
XOR   Y   50   0000000000110010
-------------------------------
        86   0000000001010110
```

XOR can be calculated using NOT, AND and OR. Here is how you do it.

X XOR Y =(X AND (NOT Y)) OR ((NOT X) AND Y)

So, in plain English, XOR is Either X AND NOT Y OR NOT X AND Y.

X XOR Y =(X AND (NOT Y)) OR ((NOT X) AND Y)

Proof

X	Y	NOT X	NOT Y
0	0	1	1
1	0	0	1
0	1	1	0
1	1	0	0

X	(NOT Y)	X AND (NOT Y)
0	1	0
1	1	1
0	0	0
1	0	0

(NOT X)	Y	(NOT X) AND Y
1	0	0
0	0	0
1	1	1
0	1	0

(X AND (NOT Y))	((NOT X) AND Y)	(X AND (NOT Y)) OR ((NOT X) AND Y)
0	0	0
1	0	1
0	1	1
0	0	0

Therefore X XOR Y=(X AND (NOT Y)) OR ((NOT X)) AND Y)

EXAMPLE: USE OF XOR

XOR has the unique property that if you XOR a value with another value twice, you get the original value back.

X XOR Y XOR Y = X

Let's try 100 XOR 50 XOR 50 as an example:

100 XOR 50

	X	100	0000000001100100
XOR	Y	50	0000000000110010
		86	0000000001010110

(100 XOR 50) XOR 50

	X	86	0000000001010110
XOR	Y	50	0000000000110010
		100	0000000001100100

Therefore 100 XOR 50 XOR 50 = 100

EQV

This is the logical equivalence function. EQV returns true whenever X and Y are equal, and false when they are different.

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

Example 51 EQV 75

	X	51	0000000000110011
EQV	Y	74	0000000001001010
		−122	1111111110000110

The following relation is true.

X EQV Y = (X AND Y) OR ((NOT X) AND (NOT Y))

Proof

X	Y	NOT X	NOT Y
0	0	1	1
1	0	0	1
0	1	1	0
1	1	0	0

X	Y	(X AND Y)
0	0	0
1	0	0
0	1	0
1	1	1

(NOT X)	(NOT Y)	((NOT X) AND (NOT Y))
1	1	1
0	1	0
1	0	0
0	0	0

(X AND Y)	((NOT X) AND (NOT Y))	(X AND Y) OR ((NOT X) AND (NOT Y))
0	1	1
0	0	0
0	0	0
1	0	1

Therefore

X EQV Y = (X AND Y) OR ((NOT X) AND (NOT Y))

IMP

IMP Truth table.

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

100 IMP 50 can be calculated as follows:

	X	100	0000000001100100
IMP	Y	50	0000000000110010
		−69	1111111110111011

The following relation is true.

X IMP Y = (NOT X) OR Y

Proof

X	Y	NOT X
0	0	1
1	0	0
0	1	1
1	1	0

(NOT X)	Y	(NOT X) OR Y
1	0	1
0	0	0
1	1	1
0	1	1

Therefore X IMP Y = (NOT X) OR Y

Well, that is about it as far as MSX BASIC goes but Boolean algebra does not stop there. There are many other useful functions such as NOR and NAND which are not included in MSX but can be simulated using NOT, AND, and OR. Remember that these three are the basics of Boolean algebra, and they can be combined to give the others.

Take a look at the table below. Two binary variables, X and Y, each can have a value of either 1 or 0. They can have four different combinations of 0 and 1, since 2 ^ 2=4. There are 16 combinations of these combinations (or functions) since 2 ^ 2 ^ 2=16. What you get is a list of the all possible combinations or truth tables.

X	Y	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15
0	0	0	0	1	0	1	0	0	0	1	0	1	1	1	0	1	1
0	1	0	0	1	1	0	0	0	1	0	1	1	0	1	1	0	1
1	0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	1
1	1	0	1	0	1	0	1	0	0	0	1	1	1	0	0	1	1

You can recognise the following:

f2 is NOT
f5 is AND
f9 is OR
f13 is XOR
f11 is IMP
f14 is EQV

These are all in MSX BASIC but what about the others, e.g. f1 and f3 etc.?

In this section we will look into non-MSX logical operators. These can't be used in MSX as they are, so an equivalent NOT AND OR expression will be given.

f0 Always false function whatever the combination of X and Y is. Not used in computing.

f1 Always X whatever the value of Y.

f2 NOT X (see NOT).

f3 Always Y whatever the value of X.

f4 NOT Y (see NOT).

f5 X AND Y (see AND).

f6 X AND (NOT Y).

f7 (NOT X) AND Y.

f8 NOR function. Its MSX equivalents are

(NOT X) AND (NOT Y) or alternatively,
NOT (X OR Y) ..... see De Morgan's Law.

f9 X OR Y (see Logical OR).

f10 ((NOT X) OR Y) Similar to IMP.

f11 X IMP Y (see IMP). Similar to f10. Alternatively X OR (NOT Y).

f12 NAND function. It is negation of AND (NAND is short for NOT AND).

NOT (X AND Y)
(NOT X) OR (NOT Y)
X NAND Y = (NOT X) OR (NOT Y) = NOT (X AND Y).
See De Morgan's law later on.

f13 X XOR Y. Exclusive OR (see XOR). Alternatively (X AND (NOT Y)) OR ((NOT X)) AND Y)

f14 X EQV Y. Logical equivalence.
X EQV Y=(X AND Y) OR ((NOT X) AND (NOT Y))

f15 Always true whatever the value of X and Y. Not used.

NAND and NOR, both not included in MSX BASIC, are useful functions to remember.

Some Useful Logical Relations

There are a variety of logical relationships which can be used in Boolean algebra. They can be used in shortening long winded logical expressions by way of minimisation.

USING LOGICAL OR

1 X OR 0=X
2 X OR X=X
3 X OR X OR X OR X OR ... =X
4 X OR (NOT X)=1
5 X OR 1=1
6 X OR (X AND Y)=X
7 X OR ((NOT X) AND Y)=X OR Y

USING LOGICAL AND

1 X AND 0=0
2 X AND X=X
3 X AND X AND X AND X . . . . . =X
4 X AND (NOT X)=0
5 X AND 1=X
6 X AND (X AND Y)=X AND Y
7 X AND ((NOT X) AND Y)=0

Examples

Try out some of the following examples on your computer and see if the above relations are really true.

	ANSWERS
PRINT 145 OR 0	145
PRINT 167 OR 167	167
PRINT 167 OR 167 OR 167	167
PRINT 133 OR (133 AND 222)	133
PRINT 111 AND 0	0
PRINT 111 AND 111	111
PRINT 111 AND 111 AND 111 AND 111	111
PRINT 234 AND (NOT 234)	0
PRINT 234 AND (234 OR 54)	234

De Morgan's Rules

Apart from the above relationships, Boolaen algebra has two special tricks to reduce the length of logical expressions.

1 Negation of Logical OR

The negation (NOT) of a logical OR is equivalent to the logical AND of negations of the variables making up the logical OR. This may be expressed as:

NOT (X OR Y)=(NOT X) AND (NOT Y)

2 Negation of Logical AND

The negation of a logical AND is equivalent to the logical OR of the negations of the variables comprising the logical AND. This may be expressed as:

NOT (X AND Y)=(NOT X) OR (NOT Y)

Laws of Rearrangement

In mathematics, there are a number of laws which allow you to rearange and simplify complex expressions. They are known as the commutative, associative, and distributive laws. Boolean algebra has similar laws as follows:

1 Commutative Laws

a. X OR Y = Y OR X
b. X AND Y = Y AND X

2 Associative Laws

a. X AND (Y AND Z) = (X AND Y) AND Z = X AND Y AND Z
b. X OR (Y OR Z) = (X OR Y) OR Z = X OR Y OR Z

3 Distributive Laws

a. X AND (Y OR Z) = (X AND Y) OR (X AND Z)
b. X OR (Y AND Z) = (X OR Y) AND (X OR Z)

Summary of MSX Logical Operator's Truth Table

X	Y	AND	OR	XOR	EQV	IMP
0	0	0	0	0	1	1
1	0	0	1	1	0	0
0	1	0	1	1	0	1
1	1	1	1	0	1	1

CHAPTER 7
BOOLEAN II:
THE IF/THEN/ELSE

When you are writing a computer program, it is very likely that the computer will have to make decisions by itself while running your program. To make decisions on certain conditions, use the IF statement to first test the conditions, and then carry out certain tasks depending on the outcome. IF is always used in conjunction with THEN, and sometimes ELSE key words. It has basically two types of format:

IF <conditions> THEN <statements>

and

IF <conditions> THEN <statements> ELSE <statements>

IF is followed by the <conditions> to be tested. If the <conditions> turn out to be true, then the computer executes the <statements> after the THEN word. If the computer finds that the <conditions> are not true then, if there is no ELSE word in that line, it will carry on from the next line without executing the <statements> after the THEN.

If, however, there is an ELSE word within that line, then if the <conditions> is found to be false, the <statements> after ELSE are executed. In this case the <statements> between THEN and ELSE are completely ignored.

To begin with, let's look at the <conditions> part of the IF THEN ELSE structure.

<CONDITIONS>

This bit follows all the rules and regulations described in the EXPRESSIONS AND OPERATORS chapter, and BOOLEAN ALGEBRA chapter. This section shows the practical side of what we said in the last two chapters.

Numerical Comparison

The simplest condition compares two values, e.g. they can test whether one number is larger than another. They use the relationed operators, such as <,>,<>,=,<=,>=.

Examples

```
1   15<8      returns   0 (false).
2   a%=67     returns   -1 (true) if a%=67
                        0 false if a% is not 67.
```

In an IF THEN format:

```
1   IF D%=100 THEN PRINT "D% IS 100"
2   IF V<=G THEN PRINT "V<=G"
```

These are the simplest forms but you may also use arithmetic expressions on both sides of the relational operator.

```
1   IF F%*H%=9876 THEN PRINT "TRUE"
2   IF 2 ^ 8=N%-19*Y% THEN PRINT "TRUE"
```

Also, there is no reason why you can't use BASIC functions such as COS and INSTR, etc. In fact, you can even incorporate a user defined function FN, too.

```
1   IF INSTR(A$,B$)=5 THEN PRINT B$
2   IF COS(0.6242)>= TAN(D%) THEN PRINT "TRUE"
3   IF FNA(D%*I%-100)=&HFF THEN PRINT "FF"
```

Numerical Condition without Comparison

It is not strictly necessary to have a relation as a <condition>. In fact, it can be a single numerical variable or just a simple expression Take a look at this example:

```
IF X% THEN PRINT "X%=TRUE"
```

In this case the computer takes the condition to be true when the variable X% is non-zero, and false when X%=0.

In general:

```
IF <expression> THEN <statement>
```

<expression>=TRUE when <expression> returns non-zero.
<expression>=FALSE when <expression> returns 0.

You can use this in a variety of situations. Here are some examples to illustrate this point.

```
1   IF A<>0 THEN PRINT "A NOT 0"
    . . . can be
    IF A THEN PRINT "A NOT 0"
```

2 To check if there is a certain character within a string.
IF INSTR(S$,"a") THEN PRINT "S$ CONTAINS CHARACTER 'a' "

String Comparison

You may compare two strings using relational operators in much the same way. The comparison is made by taking one character at a time from each string and comparing the ASCII codes. If the ASCII codes differ, the lower case code number precedes the higher. For which character corresponds to which ASCII code, see the ASCII code table. If during string comparison the end of one string is reached, the shorter string is said to be smaller. String comparison can be used to alphabetize strings.

In the following example, all the conditions are true.

1 "abc"="abc"
2 "ABC"<"abc"
3 "ABC"<"ABD"
4 "ab"<"abc"
5 "1234"<"12345"

Multiple Condition Testing using Boolean Operators

It is possible to test multiple conditions in one go, using Boolean logical operators. There are 6 logical operators in MSX BASIC but only 3 of them, NOT, AND, and OR, are relevant. They follow all the rules governing Boolean algebra, such as De Morgan's Laws.

1. Simple use of AND

   IF <condition-1> AND <condition-2> THEN <statements>

   In this case, only if both conditions are true, will the computer execute the <statement> after THEN, e.g.

   IF X=0 AND Y$="YES" THEN PRINT "FINE"

2. Simple use of OR

   IF <condition-1> OR <condition-2> THEN <statements>

   In this case, if either one or both conditions are true, then the computer will execute the <statements> after THEN, e.g.

   IF X=0 OR Y=0 THEN PRINT "ONE OF THEM IS ZERO"

3. Simple use of NOT

   If NOT <condition> THEN <statements>

   If the condition is true then NOT <condition> returns the opposite. i.e. false, and vice versa, e.g.

   IF NOT(V=U*8) THEN PRINT "V NOT U*8"

Now, you may have noticed that it is pointless using NOT in the above relation when you can rewrite it as

```
IF V<>U*8 THEN PRINT "V NOT U*8"
```

Here is a list of such relations which have the same meanings.

```
NOT(X=Y)     X<>Y
NOT(X<>Y)    X=Y
NOT(X<=Y)    X>Y
NOT(X>=Y)    X<Y
NOT(X>Y)     X<=Y
NOT(X<Y)     X>=Y
```

So, don't waste your computer's memory.

What NOT can do is reverse the result of an expression such as:

```
IF NOT INSTR(A$,"y") THEN PRINT "y is NOT IN STRing A$"
```

The above example is self-explanatory.

Minimisation — De Morgan's Law

NEGATION OF LOGICAL OR
The following three have exactly the same effect.

```
IF NOT(X=9 OR Y=10)
IF NOT(X=9) AND NOT(Y=10)
IF X<>9 AND Y<>10
```

NEGATION OF LOGICAL AND
The following three have exactly the same effect.

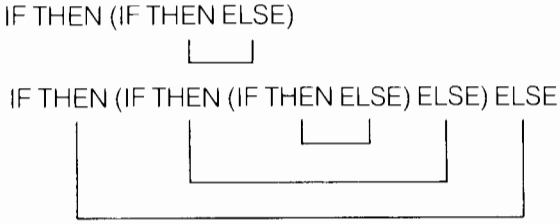
```
IF NOT(X=9 AND Y=10)
IF NOT(X=9) OR NOT(Y=10)
IF X<>9 OR Y<>10
```

If in doubt, try them out and see for yourself.

IF/THEN/ELSE STRUCTURES

ELSE is a part of the IF/THEN/ELSE structure. ELSE tells the computer that if the <condition> in the IF is not satisfied (i.e. false), then skip the statements after THEN, and execute the statements after ELSE.

Multiple IF/THEN/ELSE statements can be set up as well as nested IF/THEN/ELSE structures. They are only limited in the length of one line, which is 255 characters. If there are less ELSE clauses than THENs, then each ELSE is matched to the nearest THEN.



Multiple IF/THEN/ELSE

The second IF statement is executed after the first IF statement is found to be false.

Syntax

IF . . . THEN GOTO <line> format can be abbreviated into two forms:

1. IF . . . THEN <line>
2. IF . . . GOTO <line>

Also ELSE GOTO <line> can be abbreviated:

1. IF . . . THEN . . . . ELSE <line>
2. IF . . . THEN . . . . GOTO <line>

Points to Remember

IF/THEN/ELSE statements tend to get too long to fit on a single line, because you can have multiple statements after THEN and ELSE. If this is the case, then it is a good idea to use subroutines using the GOSUB statement.

It is advisable that you avoid nested IF/THEN/ELSE statements, as they can get into a bit of muddle. You can create a spaghetti program if you don't watch out.

CHAPTER 8

PRINT USING

INTRODUCTION

MSX has a comprehensive print format control, using the PRINT USING statement. The PRINT USING statement allows you to print strings or numbers in various formats.

Syntax

PRINT USING <string-exp>;<list of items>

There are two parts to the argument of PRINT USING — the format specifier string, and the list of items to be printed. They are separated by a semi-colon. The <list of items> can be numbers or strings. <string-exp> contains the special formatting string which determines how the items are to be printed. It can be either a string or a string variable.

CHARACTER EXPLANATION

! The exclamation mark specifies that the first character of the string is to be printed.

Here is an example program, which prints out only the initial of each of the first names and surnames given in the data statements.

```
10 FOR I = 1 TO 3
20 READ FIRST$,SUR$
30 PRINT USING "!";FIRST$,"-",SUR$
40 NEXT I
50 DATA JOSEPH,MINTER
60 DATA MAX,RAYNE
70 DATA DEREK, MCNALLY

RUN
J-M
M-R
D-M
```

@ Inserts a specified string into the position given by @.

```
10 A$="QWERTY"
20 PRINT USING "THIS IS A @ KEYBOARD";A$
THIS IS A QWERTY KEYBOARD
```

The "#" sign indicates a digit to be printed. For example: PRINT USING "#.###";1.3 prints

```
1.300
```

i.e. it formats the number to be displayed.
As you can see in the above example, you may include a decimal point in the formatting string, between the "#" signs. If the number has fewer digits than specified by the formatting characters, it will be right-justified with preceding spaces.

```
10 PRINT "####.########"
20 PRINT USING "####.########";ATN(1)*4

####.########
   3.1459265
```

If the formatting characters specify more decimal places than the number to be displayed, the empty spaces will be filled with zeros.

```
10 PRINT "##.######"
20 PRINT USING "##.######";99.999

##.######
99.999000
```

Numbers will be rounded if they do not fit in the specified field.

```
10 PRINT "##.###"
20 PRINT USING "##.###";10.2367

##.###
10.237
```

You may print two or more numbers with the same format in one PRINT USING statement.

```
10 PRINT USING "##.##",9.866,18.7

 9.97    18.70
```

If the number to be printed is negative, then the negative sign will be displayed but it will occupy one digit space in the specifier.

```
10 PRINT "###.####"
20 PRINT USING "###.####";-2.63

###.####
 -2.6300
```

If the number to be printed does not fit into the field specified by the format characters, a % percent sign is printed in front of the

number. This also happens when the rounded number exceeds the size of the field.

```
PRINT USING "#.###";9.9999

%10.000
```

+ A plus "+" sign at the beginning or the end of the formatting string will result in the sign of the number being printed, i.e. a "+" or "−" at the position specified.

```
PRINT USING "+###.#####";-0.123422,10.986
-0.12342    +10.98600
PRINT USING "####.#+";10.71,100,-200.5
 10.7+   100.0+   200.5-
```

****** The double asterisk sign causes the leading spaces in the numeric field to be filled with "*"s. Also, they represent two more digit spaces in the field.

```
10 PRINT "**##.########"
20 PRINT USING "**##.########";ATN(1)*4
30 PRINT USING "**##.########";-ATN(1)*4

**##.########
***3.14159265
**-3.14159265
```

$$ The double dollar sign puts a dollar sign in front of a number. $$ specifies two more digit spaces in the field, one of which will be the $ dollar sign.

```
10 PRINT "$$#.########"
20 PRINT USING "$$#.########";ATN(1)*4
30 PRINT USING "$$#.########";-ATN(1)*4

$$#.########
 $3.14159265
-$3.14159265
```

Note that you can't use the exponential format in conjunction with $$ sign.

****$** This sign is a combination of ** and the $$ sign. All blank spaces are filled by * and the number is preceded with a $ sign. **$ specifies three more digit positions, one of which is the $ sign.

```
10 PRINT "**$#.########"
20 PRINT USING "**$#.########";ATN(1)*4
30 PRINT USING "**$#.########";-ATN(1)*4

**$#.########
**$3.14159265
*-$3.14159265
```

, A comma is placed to the left of the decimal point in the formatting string causes a comma to be printed every third digit to the left of the decimal point.

```
10 PRINT "##########,.##"
20 PRINT USING "##########,.##";1090382.88#
30 PRINT "$$##########,.##"
40 PRINT USING "$$##########,.##";1090382.88#

##########,.##
  1,090,382.88
$$##########,.##
   $1,090,382.88
```

A comma at the end of a format string prints a comma at the end of the number.

```
PRINT USING "##.##,";12.567

12.57,
```

^^^^ This is the exponent formatting specifier. This allows space for E+xx. You can also specify the position of decimal points. Significant digits are left justified, and the exponent is adjusted.

```
PRINT USING "##.##^^^^";200.00
 2.00E+02
PRINT USING "+#.##^^^^";200.00
+2.00E+02
PRINT USING "#.##^^^^+";-200.00
2.00E+02-
```

Notes

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

LPRINT USING

LPRINT USING is exactly the same as PRINT USING but this writes to the printer and not to the screen.

PRINT# USING

PRINT# USING is exactly the same as PRINT USING but this writes to the device specified by the file number after the # mark. You must execute an OPEN statement to open a channel to the device you want to write to, before using this statement.

Syntax

PRINT#<file number>,USING <string-exp>;<list of expression>

CHAPTER 9

EVENT HANDLING AND INTERRUPT BY BASIC

INTRODUCTION

The MSX computer is equipped with various event handling statements. Event handling is interrupt driven. That is, to say, that while the computer is executing a program it is also looking out for a particular event to happen. If that event occurs, it interrupts the proceedings by immediately jumping to a predetermined user subroutine.

The MSX can handle interrupts for the following events:

1	Set Time Interval	ON INTERVAL GOSUB
2	Function key press	ON KEY GOSUB
3	<CTRL><STOP> press	ON STOP GOSUB
4	Trigger press	ON STRIG GOSUB
5	Sprite collision	ON SPRITE GOSUB
6	Errors	ON ERROR GOSUB

1) TIME INTERVAL INTERRUPT

Statements used:

ON INTERVAL=<time> GOSUB <line>
INTERVAL ON/OFF/STOP

The MSX has its own internal clock, which starts when you switch on the computer. The time can be found using the TIME function. TIME is incremented every 1/50th of a second, every time the Video Display Processor makes an interrupt.

Using the ON INTERVAL GOSUB statement, you can specify a time interval for the interrupt to occur. Since the TIME is incremented every 1/50th of a second, to set the time interval for the interrupt to 1 second

you have to equate the INTERVAL to 50, i.e. ON INTERVAL=50 GOSUB <line>.

ON INTERVAL=<time> GOSUB also sets the subroutine which the BASIC should jump to once the interrupt occurs. The computer will jump to this subroutine as soon as the interrupt occurs, regardless of which line in the main program the computer is in.

The interval interrupt is enabled with INTERVAL ON. This tells the computer to start counting the clock. Unless this statement is executed, the ON INTERVAL GOSUB statement will not operate.

Once the time interrupt occurs, an automatic INTERVAL STOP is executed. This prevents the time interrupt occurring during the current time interrupt subroutine. However, it remembers if an interrupt has happened during the current subroutine and the computer will immediately execute the subroutine again, if this has happened, unless the current subroutine disables the time interrupt altogether by executing INTERVAL OFF.

After it leaves the interrupt subroutine, the computer will automatically execute INTERVAL ON to enable the interrupt, unless INTERVAL OFF is executed within the subroutine.

You may disable the time interrupt any time you like using INTERVAL OFF.

Time interrupt is disabled as soon as the BASIC gets out of a program, i.e. no time interrupt in direct mode is possible. Also, it is disabled in any error trapping subroutines.

EXAMPLES

A short program to simulate a digital watch with beeps:

```
10 ON INTERVAL=50 GOSUB 60
20 INTERVAL ON
30 CLS
40 S=0:M=0
50 GOTO 50
55 REM INTERVAL SUBROUTINE
60 IF S=60 THEN S=0:M=M+1:LOCATE10
,11:PRINT "MIN";M:BEEP
70 S=S+1
80 LOCATE 10,10:PRINT "SEC";S
90 BEEP
100 RETURN
```

LINE 10 SETS INTERVAL TO 1 SEC SUBROUTINE AT 60
LINE 20 ENABLES INTERVAL TRAPPING
LINE 30 CLEARS THE SCREEN
LINE 40 INITIALISES S AND M
LINE 50 INFINITE LOOP TO WAIT FOR AN INTERRUPT
LINE 60 IF 60 SECONDS THEN 1 MINUTE
LINE 70 S+1 SECOND
LINE 80 PRINTS SECONDS
LINE 90 BEEPS
LINE 100 RETURNS TO WHEREVER IT LEFT MAIN ROUTINE: IN THIS CASE ALWAYS LINE 50

```
SEC 54
MIN 3
```

GIVES A BEEP EVERY SECOND AND A DOUBLE BEEP EVERY MINUTE

2) FUNCTION KEY INTERRUPT

Statements used:

ON KEY GOSUB <line>,<line>,.....
KEY (<key number>) ON/OFF/STOP

It is possible to set up interrupts for the 10 function keys using ON KEY GOSUB and KEY () ON/OFF/STOP STATEMENTS.

ON KEY GOSUB is followed by a list of line numbers, specifying the interrupt subroutine for each function key. If there is no corresponding subroutine for a particular function key, then the line number can be omitted.

KEY (<key number>) ON statements enable function key interrupts for each function key. Unless this statement is executed, the computer will not look out for an interrupt for the function key. When there is an interrupt, the computer jumps to the subroutine specified by ON KEY GOSUB.

Once the function key interrupt occurs, an automatic KEY (<key number>) STOP is executed. This prevents the interrupt by the same function key occurring during the current interrupt subroutine. However, it remembers if that function key has been pressed during the current subroutine, and the computer will immediately go to the same subroutine again, if this has happened, when it has finshed the current subroutine, unless the current subroutine disables the KEY interrupt altogether, by executing KEY (<key number>) OFF.

After it leaves the interrupt subroutine, the computer will automatically execute KEY ON to enable the interrupt, unless KEY (<key number>) OFF is executed in the subroutine.

KEY interrupt is disabled when the program is not running, and also during error trapping routines.

EXAMPLE

```
10 ON KEY GOSUB 100,120
,140,160,180,200
20 KEY (1) ON
30 KEY (2) ON
40 KEY (3) ON
50 KEY (4) ON
60 KEY (5) ON
70 KEY (6) ON
80 GOTO 80
90 REM SUBROUTINES HERE
100 PRINT "FUNCTION 1"
110 RETURN
120 PRINT "FUNCTION 2"
130 RETURN
140 PRINT "FUNCTION 3"
150 RETURN
160 PRINT "FUNCTION 4"
170 RETURN
180 PRINT "FUNCTION 5"
190 RETURN
200 PRINT "FUNCTION 6"
210 RETURN
```

LINE 10 SETS SUBROUTINES FOR F1, F2, F3, F4, F5 AND F6
LINE 20 ENABLES FUNCTION KEY F1
LINE 30 ENABLES FUNCTION KEY F2
LINE 40 ENABLES FUNCTION KEY F3
LINE 50 ENABLES FUNCTION KEY F4
LINE 60 ENABLES FUNCTION KEY F5
LINE 70 ENABLES FUNCTION KEY F6
LINE 80 INFINITE LOOP TO WAIT FOR A FUNCTION KEY
LINE 100 ROUTINE FOR F1
LINE 110 RETURNS TO WHEREVER IT CAME FROM
LINE 120 ROUTINE FOR F2
LINE 130 RETURNS TO WHEREVER IT CAME FROM

```
LINE 140  ROUTINE FOR F3
LINE 150  RETURNS TO WHEREVER IT CAME FROM
LINE 160  ROUTINE FOR F4
LINE 170  RETURNS TO WHEREVER IT CAME FROM
LINE 180  ROUTINE FOR F5
LINE 190  RETURNS TO WHEREVER IT CAME FROM
LINE 200  ROUTINE FOR F6
LINE 210  RETURNS TO WHEREVER IT CAME FROM
```

3) <CTRL><STOP> INTERRUPT AND BREAK PROOFING

Statements used:

ON STOP GOSUB <line>
STOP ON/OFF/STOP

A BASIC program is very easy to break into. All you need to do is to press <CTRL><STOP>. This halts the program immediately and puts you back into command mode. However, you will sometimes find that you need to 'break-proof' a program to prevent users of the program seeing the contents. This is done by using trapping <CTRL><STOP> with ON STOP GOSUB.

The ON STOP GOSUB statement sets the subroutine for the <CTRL><STOP> interrupt. Within this subroutine, you can either put a message saying "Don't break out" or just a RETURN statement so the program can immediately carry on even if there was a <CTRL><STOP> interrupt.

STOP ON/OFF/STOP enables/disables <CTRL><STOP> trapping.

When STOP ON is executed, the BASIC starts to check if <CTRL><STOP> has been pressed every time it executes a new statement. If a <CTRL><STOP> has been detected, then the BASIC is diverted to the subroutine specified by the ON STOP GOSUB statement executed earlier on in the program.

Once the interrupt occurs, an automatic STOP STOP is executed. This stops the interrupt occurring during the current subroutine. However, it remembers if <CTRL><STOP> is pressed again during the current subroutine, and the computer will immediately go to the subroutine again once it has left the current subroutine, if this has happened, unless the current subroutine disables the <CTRL><STOP> interrupt altogether by executing STOP OFF. After it leaves the interrupt subroutine, the computer will automatically execute STOP ON to enable the interrupt, unless STOP OFF is executed within the subroutine.

The only way to break out of a break-proofed program is to RESET the computer. Therefore you should remember to save your break-proofed program before you RUN it.

EXAMPLES
Here is a ready-made 'break-proofing' routine to add on to your program.

```
10 ON STOP GOSUB 10000
20 STOP ON
..
..
   ADD YOUR PROGRAM HERE
..
..

9999 REM CTRL-STOP SUBROUTINE
10000 RETURN
```

LINE 10 STOP SUBROUTINE SET TO LINE 10000
LINE 20 SWITCHES ON THE STOP DETECTOR
LINE 10000 RETURNS TO WHEREVER THE INTERRUPT WAS DETECTED

Note that ON STOP trapping does not prevent the STOP key halting the program. It just prevents you from breaking into a program by <CTRL><STOP>.

If you press the STOP key only, you will find that it will halt your program and display the cursor. If you press the STOP key for the second time, the program will continue.

The <CTRL><STOP> interrupt is disabled when the program is not running and also during error trapping routines.

4) JOY STICK TRIGGER INTERRUPT

Statements used:

ON STRIG GOSUB <a list of line numbers>
STRIG(<n>) ON/OFF/STOP

The trigger number, <n>, for each trigger is as follows:

0=Space bar
1=trigger 1 of joystick 1
2=trigger 1 of joystick 2
3=trigger 2 of joystick 1
4=trigger 2 of joystick 2

Every MSX compatible joystick has two trigger buttons. The <space> bar is also considered as a trigger.

ON STRIG GOSUB sets trigger interrupt subroutines for these

triggers. You can define subroutines for all the triggers at once by listing the line numbers for each trigger.

If there is no subroutine for a particular trigger, then you may omit it and just place a comma.

For example:

```
ON STRIG GOSUB 1000,200,,,
```

will interrupt to a subroutine when
a) trigger 0 space bar
or
b) trigger 1 on joystick 1
is pressed but does nothing if any of the other triggers are pressed.

STRIG(<n>) ON actives the trigger trapping for the specified trigger. You can only enable one trigger at a time.

Once the interrupt occurs, an automatic STRIG(<n>) STOP is executed. This stops the interrupt occurring during the current trigger subroutine. However it remembers if a trigger is pressed during this subroutine and the computer will immediately goto the corresponding subroutine for that trigger once it has left the current subroutine, unless the current subroutine disables trigger interrupt for that trigger altogether by executing STRIG(<n>) OFF. After it leaves the trigger subroutine, the computer will automatically execute STRIG(<n>) ON to enable the interrupt.

The STRIG interrupt is disabled when the program is not running and also during error trapping routines.

EXAMPLES

Here is a very short program which demonstrates how the trigger trapping works by testing each of the joystick triggers. The program is written so that both joysticks can be tested as well as the space bar as a trigger. If you press the <s> key, the program will end.

```
10 ON STRIG GOSUB 100,120,140,
   160,180
20 STRIG(0) ON
30 STRIG(1) ON
40 STRIG(2) ON
50 STRIG(3) ON
60 STRIG(4) ON
70 IF INKEY$="s" THEN END
80 GOTO 70
90 REM TRIGGER ROUTINES
100 PRINT "SPACE BAR PRESSED"
110 RETURN
120 PRINT "TRIGGER 1 JOYSTICK 1"
130 RETURN
140 PRINT "TRIGGER 1 JOYSTICK 2"
150 RETURN
160 PRINT "TRIGGER 2 JOYSTICK 1"
170 RETURN
180 PRINT "TRIGGER 2 JOYSTICK 2"
190 RETURN
```

LINE 10 SETS ALL THE TRIGGER SUBROUTINES
LINE 20 SWITCHES THE TRIGGER 0 ON
LINE 30 SWITCHES THE TRIGGER 1 ON
LINE 40 SWITCHES THE TRIGGER 2 ON
LINE 50 SWITCHES THE TRIGGER 3 ON
LINE 60 SWITCHES THE TRIGGER 4 ON
LINE 70 IF <s> IS PRESSED THEN END
LINE 80 LOOPS BACK TO LINE 70
LINE 90 TRIGGER ROUTINES

5) SPRITE COLLISION INTERRUPT

See the Advanced Graphics Sprite section for detailed explanation.

6) ERROR DETECTION

See the Error Handling section for more detail.

CHAPTER 10

ERROR HANDLING

ERROR MESSAGES

When MSX BASIC encounters an error while executing a program or a command, it displays an error message and halts the execution. Here is a summary of error messages.

CODE	MESSAGE	DESCRIPTION
1	NEXT without FOR	A variable in a NEXT statement does not match with a variable in a previous FOR statement; or a NEXT statement is encountered without a previous FOR statement.
2	Syntax error	Spelling mistakes in the keyword or wrong punctuation.
3	Return without GOSUB	A RETURN statement is encountered without a matching GOSUB statement.
4	Out of DATA	A READ statement is executed when all the data in DATA statements is used up; or there are no DATA statements to READ from.
5	Illegal function call	An Illegal paramater is passed to a mathematical or string function.
6	Overflow	The number is too large for the variable type, function or statement to handle.

7 Out of memory

Ran out of memory due to a program that is too large, too many FOR, too many GOSUB, too many functions or variables.

8 Undefined line number

Non-existent line number referred to in a GOTO, GOSUB, IF/THEN/ELSE statement or a DELETE command.

9 Subscript out of range

A subscript of an array element is out of the range specified in the DIM statement; or the number of the subscript is wrong.

10 Redimensioned array

Dimensionalising an already existent array using DIM.

11 Division by zero

A division by zero in an expression has been encountered.

12 Illegal direct

A statement that is not allowed in direct mode.

13 Type mismatch

A string variable is assigned to a number or vice versa.

14 Out of string space

Ran out of string work area in systems RAM.

15 String too long

An attempted creation of a string of more than 255 characters long.

16 String formula too complex

A string expression is too complex for the computer to handle.

17 Can't continue

Attempted CONTinue on a program that has ended with an error, or has been edited or ENDed.

18 Undefined user function

Calling an FN function without first defining it with DEF FN statement.

19 Device I/0 error

An input/output error on a cassette, printer etc. It is a fatal error with no way of recovery.

20 Verify error

CLOAD? cassette tape verification has failed.

21 No RESUME
An error trapping routine has no RESUME statement when one is expected by the computer.

22 RESUME without error
An unexpected RESUME statement is encountered outside an error trapping routine.

23 Unprintable error
The error is encountered but there is no message.

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

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

26-49 Unprintable error
Reserved for future expansion.

50 Field overflow
A FIELD statement is attempting to allocate more bytes than specified for the record length of a random file in the OPEN statement; or the end of the FIELD buffer is encountered while doing sequential I/0 (PRINT#, INPUT#) to a random file.

51 Internal error
Machine malfunction.

52 Bad file number
A statement or command referring to a non-existent file, or out of MAXFILES range.

53 File not found
A LOAD or OPEN statement referring to a non-existent file.

54 File already open
Attempt is made to OPEN an already OPENed file.

55 Input past end
An INPUT statement is executed after all the data in the file has been exhausted. Use EOF to prevent this happening.

56 Bad file name
An illegal file name given in LOAD, SAVE, or other I/0 statements.

57 Direct statement in file
A direct statement is encountered while LOADing. LOAD is terminated.

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

59 File not OPEN
The file referred to is not yet OPENed.

60-255 Unprintable error
No error codes. These may be defined by the user.

ERROR HANDLING

MSX BASIC contains many functions and statements to help you debug your program. It is usual practice to incorporate an error handling or trapping routine within your own program while it is still under development. This will help you to spot any mistakes you may have made while doing a trial RUN of your program.

Before we move on to how to write such error trapping routines, let's have a look at each of the BASIC keywords used in error handling.

ERL means error line
ERL is a reserved variable which contains the line number of the line containing the error found in the program just RUN.

ERR means error code number
ERR contains the error code number after the computer has detected an error in a program. It has a value between 1 to 255. The meanings of errors associated with the ERR number are listed as in the previous page.

ERROR

There are basically two ways of using the ERROR statement.

1. To simulate the occurrence of an error. An ERROR statement with an integer argument will fool the computer into thinking that there is an error and will terminate the program and print out the error message with the line number.
2. To create user defined errors. An ERROR statement together with the use of an ON ERROR GOTO error trapping function will let you create your own customised errors (more on this later).

ON ERROR GOTO <line>

ON ERROR GOTO enables error trapping, and specifies which line to goto once an error has been detected. Once the computer is told to trap errors it will jump to the specified line, to execute the error handling subroutine, if an error occurs during a program or in direct mode, i.e. outside the program.

RESUME

RESUME means resume BASIC operation after an error handling procedure has taken place using the error trapping statement ON ERROR GOTO. After the error has been dealt with, RESUME tells the computer to continue the execution according to the syntax below:

RESUME or RESUME 0
Resumes execution from the statement which caused the error.

RESUME NEXT
Resumes execution from the statement following the one which caused the error.

RESUME <line>
Resumes execution from the line number specified.

Error Handling Routines

To incorporate an error handling routine into your program you must have the following:

1. ON ERROR GOSUB statement right at the beginning of the program to enable the error trapping from then onwards. After this statement has been executed the computer will look out for an error, and divert to the error handling routine whether you are in direct mode or indirect mode.
2. An error handling subroutine placed right at the end of the program.

A simple program with an error handling routine looks like this.

```
10 ON ERROR GOTO 100
20 PRINT 10
30 PRONT 20
40 PRINT 30
50 END
99 REM ERROR HANDLING ROUTINE
100 ON ERROR GOSUB 0
```

LINE 10 ENABLES ERROR TRAPPING AND SETS THE ERROR SUBROUTINE TO LINE 100

LINE 30 MISTAKE HERE

LINE 50 END OF PROGRAM

LINE 99 ON ERROR GOSUB 0 CAUSES THE BASIC TO STOP AND PRINTS OUT THE ERROR WHICH HAS CAUSED THE TRAP

When the above is executed the following will happen:

```
RUN
 10
Syntax error in 30
```

The error in line 30 was detected and the program diverted to line 100. ON GOSUB 0 has an effect of displaying the error which caused the trap and halted the program. It also disables the error trapping.

So far, all this can be done without having the error trapping at all. If there was no ON ERROR GOSUB statement the computer will still detect the error in line 30, and give the "Syntax error in 30" message. The error trapping routine, however, does have other features.

For instance, the error handling subroutine could contain an error recovery procedure, to put you back into the program. To do this, you need to know what kind of errors you want the recovery to cope with, and how the computer should RESUME operation.

In this example there are too few data items in the DATA statement in line 60. This causes an out of DATA error (code 4) when the computer is on its fourth loop. The error handling routine is equipped with a recovery procedure for such an error in the form of line 100. This line finds out if the error was really the out of DATA error by checking the ERR error code, and then executes a RESTORE statement so that the data can be used again. It RESUMEs operation from the line where the error was detected.

```
10 ON ERROR GOTO 100
20 FOR I= 1 TO 5
30 READ A$
40 PRINT A$
50 NEXT I
60 DATA ONE,TWO,THREE
70 END
90 REM ERROR ROUTINE
100 IF ERR=4 THEN RESTORE:
:PRINT "out of DATA":RESUME
110 ON ERROR GOTO 0
```

LINE 10 ENABLES ERROR TRAPPING AND SETS THE ERROR SUBROUTINE TO LINE 100
LINE 20 LOOP FIVE TIMES
LINE 30 READS DATA
LINE 40 PRINTS A$
LINE 50 NEXT LOOP
LINE 60 ONLY THREE DATA ITEMS HERE

LINE 70 END
LINE 100 IF ERR WAS OUT OF DATA (CODE 4) THEN RESTORE AND RESUME OPERATION FROM WHERE IT HAS LEFT OFF
LINE 110 PRINTS OUT ERROR MESSAGE IF ANY OTHER ERROR IS ENCOUNTERED

```
RUN
ONE
TWO
THREE
out of DATA
ONE
TWO
```

With some errors you wouldn't want the computer to RESUME operation from the same line. Errors such as Syntax error can't really be remedied within a program, so they have to be edited. However, if you want the computer to skip the line with the error while giving you a message, and then follow the program through to the end, use the RESUME NEXT statement which resumes operation from the next line after the one in which the error was detected.

```
10 ON ERROR GOTO 100
20 PRINT 10
30 PRoNT 20
40 PRINT 30
50 END
99 REM ERROR HANDLING ROUTINE
100 PRINT "Error code ";ERR;
"at line";ERL
110 RESUME NEXT
```

LINE 10 ENABLES ERROR TRAPPING AND SETS THE ERROR SUBROUTINE TO LINE 100
LINE 30 MISTAKE HERE
LINE 50 END OF PROGRAM
LINE 100 DISPLAYS ERROR CODE AND THE ERROR LINE
LINE 110 THEN RESUMES OPERATION FROM NEXT LINE

```
RUN
10
Error code 2 at line 30
30
```

Note that you can specify which line to goto in the RESUME statement, so in the above example, RESUME 40 has the same effect.

It is possible to display the line which contains the mistake using an error handling routine. It will be much easier to correct the mistake if the line with the error is displayed straight after the error has been detected. To do this you must end the error recovery routine with the LIST. (list dot) command. LIST. has the effect of listing the last line referred to by the computer.

```
10 ON ERROR GOTO 100
20 PRINT 10
30 PRoNT 20
40 PRINT 30
50 END
99 REM ERROR HANDLING ROUTINE
100 PRINT "Error code ";ERR;
"at line";ERL
110 LIST.
```

LINE 10 ENABLES ERROR TRAPPING AND SETS THE ERROR SUBROUTINE TO LINE 100
LINE 30 MISTAKE HERE
LINE 50 END OF PROGRAM
LINE 100 DISPLAYS ERROR CODE AND THE ERROR LINE
LINE 110 THEN LISTS THE LINE WITH THE ERROR

```
RUN
10
Error code 2 at line 30
30 PRoNT 20
```

Now you can correct the line 30 to:

30 PRINT 20

HOW TO CREATE YOUR OWN CUSTOMISED ERRORS

An ERROR statement together with the use of an ON ERROR GOTO error trapping function will let you create your own customised errors.

There are about 36 errors between error code numbers 1 and 60 in the present version of MSX BASIC. Code numbers 61 to 255 can be used by the programmer.

To program your own error, first of all you must put the computer into the error trapping mode by executing the ON ERROR GOTO <line> statement at the beginning of the program. Then, if in the middle of the program, a condition arises such that you want to call the error trapping routine, you can do so by:

IF <condition> THEN ERROR <error code>.

ON ERROR GOTO will be activated and the computer will immediately start executing the error trapping subroutine. Within the subroutine you must have a line saying:

IF ERR=<error code> THEN PRINT "Error message"

In the following example all the commands inputted with KILL will be treated as a new error (No. 255), and are dealt with in the error trapping routine using the ERR function.

```
10 ON ERROR GOTO 100
20 INPUT "MY LORD, WHAT IS THY COMMAND";A$
30 IF INSTR(A$,"KILL") THEN ERROR 255
40 PRINT "OK."
50 END
..
100 IF ERR=255 THEN PRINT"KILLING
IS ILLEGAL IN THIS ADVENTURE.":RESUME 20
110 END
```

LINE 10 ACTIVATES ERROR TRAPPING
LINE 20 INPUT
LINE 30 CHECKS FOR INVALID WORD
LINE 100 ERROR MESSAGE RESUME FROM 20
LINE 110 END IF ERROR IS NOT 255

```
RUN
MY LORD, WHAT IS THY COMMAND? KILL HIM
KILLING IS ILLEGAL IN THIS ADVENTURE
MY LORD, WHAT IS THY COMMAND? GOTO EAST
OK
```

It is usual practice in MSX BASIC, when you are defining your own error codes, to work your way down from 255, so that you can avoid clashes with any future enhancements to the error messages made by the manufacturers of MSX computers.

Notes on Error Handling

The ERROR statement can also be used to simulate the occurrence of an error. For example:

```
ERROR 2
```

will give:

```
Syntax error
```

If the error code has no previously defined error message, then, when the computer encounters this error in the program, it will print "Unprintable Error".

No error trapping is carried out during an error trapping routine. If there is an error within the error trapping routine, then it will give the error message and halt the program.

Not all errors can be dealt with by error trapping routines. It is therefore recommended to end the error trapping routine with ON ERROR GOTO 0 which will stop the BASIC program execution and display the error message.

ON ERROR disables all event handling traps such as ON INTERVAL and ON STRIG.

CHAPTER 11

CASSETTE SAVING AND LOADING

INTRODUCTION

There are several ways of saving and loading programs and data to a cassette tape recorder.This section deals with more sophisticated use of the cassette tape recorder.

Statements and Functions Associated with Cassette Saving and Loading

Here is a list of statements and functions which are associated with the display screen.

Saving and loading BASIC programs (see the Introduction to MSX BASIC for details).

CLOAD	Loads a BASIC program from cassette
CSAVE	Saves a BASIC program to cassette
CLOAD?	Verifies a BASIC program on tape to the one in the computer's memory
MOTOR	Switches on/off cassette motor.

Saving and Loading and Merging using the ASCII format

SAVE	Saves a BASIC program to the specified device, in an ASCII file.
LOAD	Loads a BASIC program saved in an ASCII file from the specified device.
MERGE	Merges a BASIC program saved in an ASCII file to a BASIC program in the computer's memory.

Saving and Loading a section of the computer's memory as machine code program or data.

BSAVE	Saves a section of memory.
BLOAD	Loads a section of memory.

Setting the BAUD rate of cassette saving

SCREEN
CSAVE

THE CASSETTE SAVE BAUD RATE

The baud rate is the rate of data transmitted to the cassette tape recorder. The MSX uses the FSK format which gives you the option of two baud rates: 1,200 baud or 2,400 baud. This can be set by either the SCREEN statement or the CSAVE command. The default Baud rate is 1200. Their syntaxes are as follows:

SCREEN ,,,1	1200 baud
SCREEN ,,,2	2400 baud
CSAVE "<name>"	1200 baud
CSAVE "<name>",1	1200 baud
CSAVE "<name>",2	2400 baud

The recorded baud rate can be either 1200 baud or 2400 baud but the CLOAD, LOAD, and BLOAD will automatically sense which rate the recording is in and load accordingly.

SAVING AND LOADING IN THE ASCII FORMAT

When an ordinary BASIC program is recorded on cassette tape, it is in a tokenised format so that the program can be saved in an efficient format. However, if you need to record your program in ASCII codes, you must use the SAVE statement. To load in a program saved in ASCII code you must use the LOAD statement. In both SAVE and LOAD, you must specify the device you are saving from and loading to, in the form of <device-descripter>. However, only the cassette is supported in the current MSX BASIC, so the <device-descriptor> should always be "CAS:". Here is a list of syntax:

SAVE "<device-descriptor> <program-name>"	Saves a Basic program in an ASCII file.
LOAD "<device-name>"	Loads the first BASIC file found, saved in ASCII format.
LOAD "<device-name> <file-name>"	loads a BASIC file with the specified name.
LOAD "<device-name> <file-name>",R	loads a BASIC file with the specified name, then runs it.

Note that the R option in LOAD will auto-run the BASIC program just loaded.

The main use of saving in ASCII code is that the program can be merged to another, which is what we will look into next.

MERGING TWO BASIC PROGRAMS

Let us say that you have two Basic programs, program 1 and program 2 and you want to merge program 2 with program 1, such that program 2 comes after program 1 when it is eventually merged. Let's say that both are initially saved on a cassette tape.

PROG1

```
10 REM PROGRAM 1
20 PRINT "WELCOME"
30 PRINT"TO"
40 PRINT"MSX"
```

PROG2

```
10 REM PROGRAM 2
20 FOR I = 32 TO 58
30 PRINT CHR$(I)
40 NEXT I
```

First, load program 2 and renumber it so that all line numbers are greater than those in program 1.

```
CLOAD "PROG2"
RENUM 50
LIST
```

```
50 REM PROGRAM 2
60 FOR I = 32 TO 58
70 PRINT CHR$(I)
80 NEXT I
```

Then save it in an ASCII file using SAVE:

```
SAVE "CAS:PROG2"
```

Load program 1 from the cassette:

```
CLOAD "PROG1"
```

After you have loaded program 1 from cassette, rewind the program to where PROG2 is recorded. Type in:

MERGE"CAS:PROG2"

and play the cassette. The computer will add program 2 to the end of program 1.

```
10 REM PROGRAM 1
20 PRINT"WELCOME"
30 PRINT"TO"
40 PRINT"MSX"
50 REM PROGRAM 2
60 FOR I=32 TO 58
70 PRINT CHR$(I)
80 NEXT I
```

Please note the following when you are merging two programs:

1) If a line number occurs in both the initial program (1), and the second program (2), the resulting MERGEd program will contain the line from the second program (2).

2) MERGE needs the device-name in its syntax.
   MERGE "<device-name><file-name>"
   <device-name>=CAS: for cassette.

3) If the file name is omitted in the MERGE statement, then the next ASCII file encountered on the tape will be merged.

SAVING AND LOADING A SECTION OF THE COMPUTER'S MEMORY

To save a section of memory, at a specified memory address, on to cassette, use the BSAVE command. You must specify the start address and the end address of the memory you want to save. This command is used for saving machine code programs and data in the form of bytes.

If you are saving a machine code program, you have an option to specify the execution address. When you are loading the machine code in with BLOAD, if the BLOAD specifies to auto-execute the machine code program, it will execute the program from the execution address specified in BSAVE.

Anything that is saved with BSAVE must be loaded using BLOAD.

If the execution address is omitted when the code is BSAVEd, then the computer assumes the execution address is the start address when the program is BLOADed with the R option to auto-execute the program.

Here is a list of syntax:

BSAVE "<device-name>:<name>",<start address>,<end address>
BSAVE "<device-name>:<name>",<start address>,<end address>,<execution address>
BLOAD "<device-name>:<name>",R

<name> and R are optional.

R option automatically executes the machine code program just loaded. R stands for RUN.

BLOAD "<device-name>:<name>",<num-const>

When the off-set <num-const> is specified the file will be entered in memory at the position specified by <num-const>.

<device-name>=CAS: for cassette

The addresses can be given in hexadecimal numbers, e.g.

BSAVE "CAS:PROG",&HF300,&HF380,&HF30A

CHAPTER 12

ADVANCED GRAPHICS I

CHARACTERISTICS OF EACH SCREEN MODE

MSX has one of the most versatile graphics chips, the TMS 9929A, developed by Texas Instrument Inc. in the United States. It is one of the most comprehensive graphics chips available, and provides you with a variety of facilities, such as 16 colour high resolution graphics and sprite animation. It can even manage its own 16K RAM, which means that the central processor is free from providing memory for the graphics screen.

MSX BASIC has been written especially with these strong graphics capabilities, and easy programming for beginners, in mind. It is easy to use once you have experimented with it, and it can produce some spectacular pictures if you try hard enough.

Let us then start from the beginning by first explaining the display modes available on the MSX.

SCREEN MODE

MODE	TEXT/GRAPHICS	RESOLUTION	COLOUR	INPUT	GRAPHIC	SPRITE
0	40×24 textmode	40 x 24 char	2 out of 16	yes	no	no
1	32×24 textmode	32 x 24 char	2 out of 16	yes	no	yes
2	Hi-res graphic	256 x 192 char	16	no	yes	yes
3	Multi colour	64 x 48 block	16	no	yes	yes

NOTES: INPUT — use or not of the INPUT statement in the given screen mode. You can't INPUT while in a graphics mode.

GRAPHICS — use of graphics statements, for example DRAW, in the given screen mode. Generally you can't use graphics statements in text modes (0 and 1).

Statements and Functions Associated with Screen Modes

Here is a list of statements and functions which are associated with the display screen. More details are given in BASIC reference section.

Statements associated with ALL screen modes

SCREEN	Sets the screen mode.
CLS	Clears the screen.
COLOR	Sets foreground, background and border colour.

Statements and functions associated with TEXT MODES ONLY

WIDTH	Sets the width of text screen.
LOCATE	Sets the position of the cursor on text screen.
TAB	Sets the horizontal position of the cursor on current line.
CSRLIN	Returns the current vertical position of the cursor.
POS(0)	Returns the current horizontal position of the cursor.

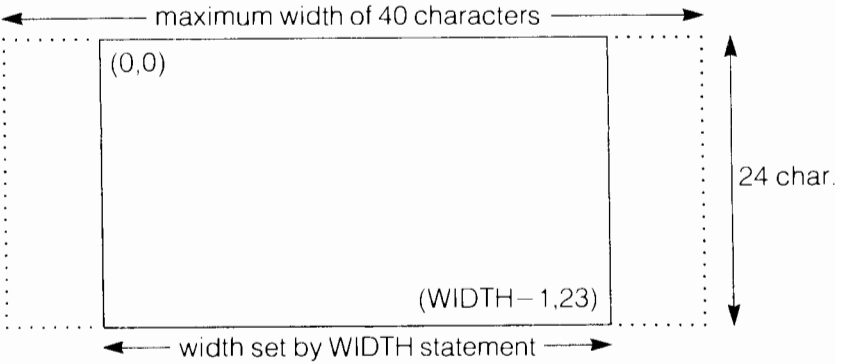
Statements and functions associated with GRAPHICS MODES ONLY

CIRCLE	Draws a circle.
DRAW	Draws according to graphics macro language.
LINE	Draws lines, squares and boxes.
PAINT	Paints with current foreground colour.
PSET	Plots a point.
PRESET	Unplots a point.
POINT	Returns the colour of the pixel referred to.

Statements associated with the VIDEO DISPLAY PROCESSOR

VPOKE	Poke to Video RAM.
VPEEK	Peek from Video RAM.
VDP	Returns values of VDP register.
BASE	Returns the base address of Video RAM tables.

MODE 0: 40 × 24 TEXT MODE SCREEN 0



MODE 0 gives 40 characters per line, which is the maximum number of characters you can have in one line for MSX computers. In this mode you may write and edit your program. You will find that the display of a program in this mode is easier to read.

MODE 0, however, has a number of disadvantages. For instance, the characters are displayed in a compressed format, i.e. 6 by 8 instead of 8 by 8 and hence some graphics characters will appear cut off. The 2 rightmost pixels of each character are not displayed at all! That does not affect alphanumeric characters, however.

The default width of the screen in MODE 0 is 37 characters. You may increase the display width to 40 characters using the WIDTH statement. The coordinates of the top lefthand corner are (0,0) while the bottom righthand corner is (WIDTH−1,23), or when initialised it is (38,23).

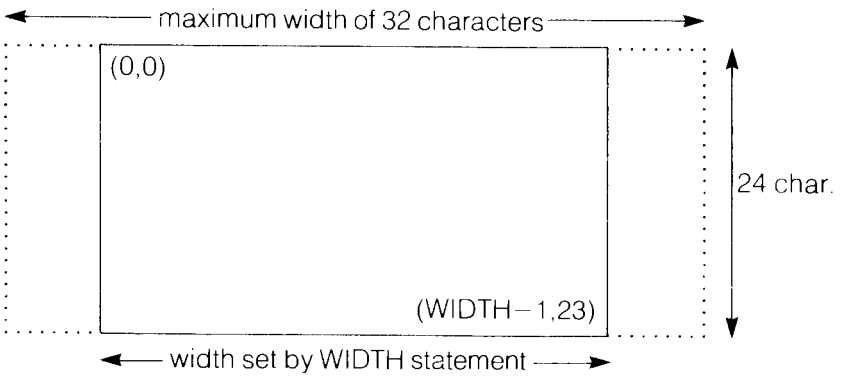
You may position the text cursor using TAB and LOCATE. To find out where the cursor is located, the function POS(0) and CRSLIN are used.

In MODE 0, you cannot use SPRITES, and also you can only use 2 out of 16 colours, although the combination of two colours is entirely your choice. The default colour is the same as MODE 1, white foreground colour and blue background. Note that MODE 0 does not have any border: it simply disappears from the display, so setting the border colour in MODE 0 is meaningless. Note that the colour of the display will change immediately upon execution of the COLOR statement.

Unlike the graphics mode, you are free to use INPUT statements and the function key list is displayed at the 23rd line unless it is disabled by KEY OFF.

All graphics statements and functions in this mode will be treated as an "Illegal function call", so watch out for that.

MODE 1: 32 × 24 TEXT MODE — SCREEN 1



Mode 1 has a resolution of 32 by 24 characters but no graphics. In this mode you may write programs and edit them. It can display 8 by 8 characters without any 'cut-offs' (as in the case in MODE 0.)

The default width of the screen is 29 characters. You may increase the display width to 32 characters using the WIDTH statement. The reason for the smaller default width is because some TVs and monitors cannot display the entire screen.

The coordinates of the top lefthand corner are (0,0), while the bottom righthand corner is (WIDTH−1,23), or when initialised, it is (28,23).

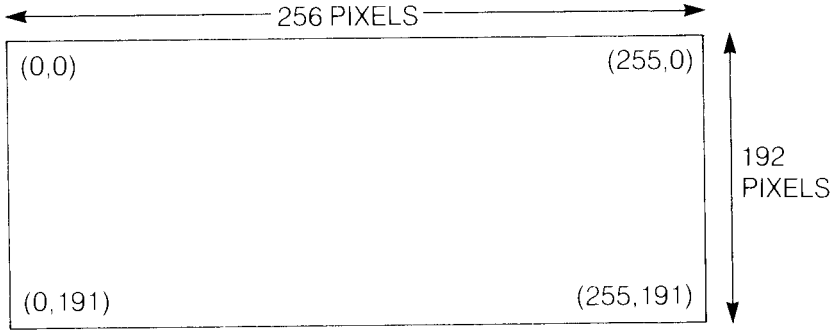
You may position the text cursor using TAB and LOCATE. To find out where the cursor is located, use the POS(0) and CRSLIN functions.

You can only use 2 out of 16 colours, although the combination of two colours is entirely your choice. Note that the colour of the display will change immediately upon execution of the COLOR statement.

Unlike the graphics mode you are free to use INPUT statements, and the function key list is displayed at the 23rd line unless it is disabled by KEY OFF.

All graphics statements and functions in this mode, except those involved in sprites, will be treated as illegal function calls.

MODE 2: 256 × 192 HIGH RESOLUTION GRAPHICS MODE : SCREEN 2



MODE 2 is the most widely used graphics mode which provides the user with a high resolution graphics screen, with 16 colours. In this mode you may use sprites and graphics macro language through DRAW statements. This is the mode which is used in most games programs.

Its horizontal resolution is 256 pixels, while the vertical resolution is 192 pixels. The colour resolution, however, is somewhat different; its colour resolution is 32 by 192. This means you can only select two colours per 8 by 1 horizontal block of pixels. You can select the background and foreground colours for each 8 pixel blocks, but if you plot a

point with a third colour, the pixels plotted in the previous colour within that block will change to the new foreground colour automatically. Hence you can obtain smudgy pictures if you are not careful with where you are plotting and with what colour. However, if you draw carefully with 8 pixel colour resolution in mind, you can produce good pictures. More on advanced drawing techniques later.

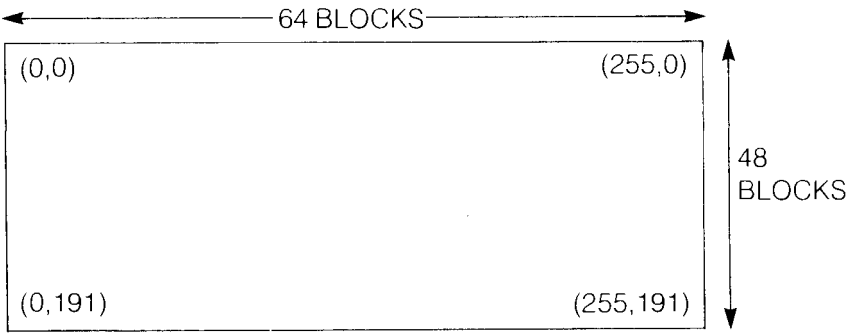
You may use sprites in this mode. The Sprites are on a different screen plane from that of the main screen plane. This means that whatever sprites are displayed, they do not affect what is plotted on the main screen in any way. You will see that when a sprite leaves a particular part of the screen, the main screen is totally undisturbed.

In graphics MODE 2, the function key list cannot be displayed. You will also find that ordinary PRINT statements will not work in this mode. To PRINT onto the graphics screen, you must first OPEN a file to the screen and use PRINT#. This will be explained in more detail in the Graphics mode printing section.

The COLOR statement in this mode does not change the display colour immediately. To change the colour of the entire scren you must execute a CLS statement.

You cannot use the INPUT statement in this mode, because it would force the screen back to the previous text mode used.

MODE 3: 64 × 48 MULTI-COLOUR LOW RESOLUTION GRAPHICS MODE: SCREEN 3



Multi-colour MODE 3 gives a low resolution of 64 by 48 blocks but the individual blocks can have their own colour (unlike MODE 2). You won't get the smudged effects as in MODE 2 but it is in low resolution, so there are few applications of this mode.

Its coordinates are the same as in MODE 2, but 4 by 4 pixel co-ordinates represent one block. This allows you to draw and plot in the same coordinates system as in MODE 2, which is advantageous.

You may use sprites in this mode in much the same way as in MODE 2.

In multi-colour MODE 3, the function key list cannot be displayed. You will also find that the ordinary PRINT statement will not work in this mode. To print onto the graphics screen you must first OPEN a file to the screen and use PRINT#. The size of the printed characters will be 4 times as large as in MODEs 1 and 2 . This will be explained in more detail in the Graphics mode printing section.

The COLOUR statement in this mode does not change the display colour immediately. To change the colour of entire screen, you must execute a CLS statement.

You cannot use the INPUT statement in this mode because it would force the screen back to the previous text mode used.

CHAPTER 13

ADVANCED GRAPHICS II

COLOUR IN HIGH RESOLUTION MODE 2

This section deals exclusively with how colours are used in MODE 2. The COLOR statement sets the colour of the current background, foreground and border. However, the background colour of the whole of the screen will not change unless you clear the screen with a CLS statement. After the COLOR statement is executed, any point, line, or shape drawn will be in the new foreground colour, unless another colour is specified within the drawing statements.

The colour resolution of MODE 2 is 32 by 192. This means that you can only select two colours per 8 by 1 horizontal block of pixels. You can specify the background and foreground colours for each 8 pixel block but if you try to plot a point with a third colour to a block which already has two colours, the pixels plotted in previous foreground colour within that block will change to the new foreground colour automatically. This can cause smudgy pictures if you are not careful. The short program below will demonstrate this point.

```
10 SCREEN 2
20 COLOR 4,15,15
30 CLS
40 LINE (0,0)-(4,191),4,BF
50 IF NOT STRIG(0) THEN 50
60 LINE (6,0)-(6,191),10

70 GOTO 70
```

LINE 10 SELECTS MODE 2
LINE 20 SETS BACKGROUND TO WHITE
LINE 30 CLEARS THE SCREEN TO WHITE
LINE 40 DRAWS A RECTANGLE IN BLUE
LINE 50 WAITS UNTIL SPACE BAR IS PRESSED
LINE 60 DRAWS A LINE IN DARK YELLOW BESIDE THE BLUE RECTANGLE
LINE 70 LOCKS GRAPHICS SCREEN

Do run this program. First, you will see a blue rectangle drawn on white background, at the edge of the screen. Then, if you press the <space> bar you will see a yellow line being drawn and the blue rectangle will change to yellow as well. The program did not plot any yellow rectangle where the blue one was. Why did the colour of the rectangle change just by plotting a yellow line near it?

To understand what has happened, let us see how the colour attributes are stored in the video RAM.

If you take the top left hand side section of the screen and magnify it, it may be represented as the table given below. (B means blue and W means white.)

Y X =	0	1	2	3	4	5	6	7	8	9	.	.	.	.
0	B	B	B	B	W	W	W	W	W	W	W	W		
1	B	B	B	B	W	W	W	W	W	W	W	W		
2	B	B	B	B	W	W	W	W	W	W	W	W		
3	B	B	B	B	W	W	W	W	W	W	W	W		
4														

Within the MSX's Video RAM, the picture above is held in binary code as shown below. Each bit corresponds to a point on the screen, 1 = foreground colour, and 0 = background colour. In this case, 1 represents blue, 0 white.

	0	1	2	3	4	5	6	7	8	9	.	.	.	.
0	1	1	1	1	0	0	0	0	0	0	0	0		
1	1	1	1	1	0	0	0	0	0	0	0	0		
2	1	1	1	1	0	0	0	0	0	0	0	0		
3	1	1	1	1	0	0	0	0	0	0	0	0		
4														

This table of memory, however, does not reveal what the colour of each point is. This is because there is a separate colour attribute table in the VRAM which contains the foreground and background colour for each of 8 by 1 pixel blocks. Let us take the top left hand corner block and see how its pattern and colour is stored in the memory.

PATTERN TABLE

0	1	2	3	4	5	6	7
1	1	1	1	0	0	0	0

COLOUR ATTRIBUTE TABLE

1	0
4	15

As you can see from the above, while the pattern table gives which pixel is in foreground and which one is in background colour, the colour attribute table gives which colour is the foreground colour and which is the background. This means that it is physically impossible to plot more than 3 colours in one block. So, when you try to draw a yellow line in column 6, first the foreground colour in the attribute table is changed to the colour code of yellow i.e. 10, and then pixel 6 is set to this new foreground colour.

PATTERN TABLE

0	1	2	3	4	5	6	7
1	1	1	1	0	0	1	0

COLOUR ATTRIBUTE TABLE

1	0
10	15

Pixels 0, and 1, 2, and 3 stay as foreground colour but now the foreground colour itself has changed from blue to yellow, and hence with it the colour of 0, 1, 2, and 3 pixels have changed.

The explanation is simple when you see it in terms of the memory table, isn't it?

So, what do we do about this problem? There is only one answer: try not to plot more than 3 colours in one block.

For example, the program above can be changed so that the yellow line is not plotted in the same block as the blue rectangle, by moving the whole picture left by 2 pixels. The improved program, which will plot a blue rectangle and yellow line side by side, will look like the following:

```
10 SCREEN 2
20 COLOR 4,15,15
30 CLS
40 LINE (2,0)-(6,191),4,BF
50 IF NOT STRIG(0) THEN 50
60 LINE (8,0)-(8,191),10

70 GOTO 70
```

LINE 10 SELECTS MODE 2
LINE 20 SETS BACKGROUND TO WHITE
LINE 30 CLEARS THE SCREEN TO WHITE
LINE 40 DRAWS A RECTANGLE IN BLUE
LINE 50 WAITS UNTIL THE SPACE BAR PRESSED
LINE 60 DRAWS A LINE IN DARK YELLOW BESIDES THE BLUE RECTANGLE
LINE 70 LOCKS THE GRAPHICS SCREEN

Note the changes which have been made in the coordinates specifiers in lines 40 to 60.

NOTE: if you want to know more about how the screen is stored inside the video RAM, turn to the Video RAM section.

CHAPTER 14

ADVANCED GRAPHICS III

HOW TO PRINT TO THE GRAPHICS SCREEN USING FILES

PRINTing to the graphics screen is not as straight forward as PRINTing in a text mode. You must open a file to the graphics screen, and then use the PRINT# statement. In fact, you can open a file to a text screen if you like, but this is not necessary.

Here is a demonstration program, which will print to all four screen modes, using files. Line 30 OPENs the file according to which screen mode (M) is selected. It executes OPEN "CRT:" AS #1 when M is less than 2. This means OPEN a file to the text screen "CRT:" AS, file number #1. When M is more than or equal to 2, then the statement executes OPEN "GRP:" AS #1 which means OPEN a file to the graphics screen "GRP:" AS, file number #1. A file OPENed should be CLOSEd once its function has finished. This is done in line 60: CLOSE#1.

```
10 FOR M = 0 TO 3
20 SCREEN M
30 IF M<2 THEN OPEN "CRT:" AS #1
ELSE OPEN "GRP:" AS #1
40 PRINT#1,"THIS IS MODE ";M
50 FOR G=1 TO 1000
60 CLOSE#1
70 NEXT M
```

LINE 10 LOOPS 0 TO 3
LINE 20 SELECTS SCREEN MODE

LINE 30 IF TEXT MODE THEN OPEN "CRT:" ELSE OPEN "GRP:" GRAPHICS SCREEN
LINE 40 PRINTS A MESSAGE
LINE 50 DELAY
LINE 60 CLOSE FILE
LINE 70 NEXT MODE

In the program above you will see the screen change to each different mode while printing the message "THIS IS MODE . . .". Note that in the high resolution screen the PRINTing is basically in the same format as in text MODE 1, i.e. 32 by 24 characters. In the multi-colour MODE 3, the text is four times the size of the normal text mode. This is because the multi-colour mode is in low resolution, and in order to PRINT characters it has to be done in block graphics.

How to Print in High Resolution Graphics Mode 2

When printing in graphics MODE 2 you cannot use the LOCATE statement to position the characters to be printed. This is because the computer does not use the text cursor in the graphics mode. Instead, it uses the graphics cursor, and therefore the computer PRINTs to the last graphics cursor position. To position the characters to be displayed, use the DRAW command to move the graphics cursor with graphics macro language's BM (blank move).

Example program: PRINT# and PRINT# USING in graphics mode 2

```
10 SCREEN 2
20 OPEN "GRP:" AS #1
30 DRAW "BM100,100"
40 PRINT#1,"HELP"
50 DRAW "BM102,102"
60 PRINT#1,"HELP"
70 DRAW "BM100,120"
80 PRINT#1, USING"###.#######";ATN(1)*4
90 GOTO 90
```

LINE 10 SELECTS SCREEN MODE 2
LINE 20 OPEN FILE TO GRP
LINE 30 POSITIONS THE GRAPHICS CURSOR AND PRINT# THERE
LINE 50 REPOSITION THE CURSOR
LINE 60 PRINTS HELP AGAIN
LINE 70 REPOSITIONS CURSOR AGAIN
LINE 80 PRINT# USING
LINE 90 LOCKS GRAPHICS MODE

In the above example you will see two "HELP" messages overlapping each other. This arises because in the graphics mode the computer does not erase before printing. This can create some interesting effects, but it is often a nuisance. The only way to erase what is printed on a part of the screen is to use the LINE statement and overwrite it with a rectangle filled with the background colour.

If you include the line:

```
45 LINE (100,100)-STEP(32,8),4,BF
```

this will erase the "HELP" message written in line 40, thus avoiding the overlap.

As you can see in the program above, you can also use PRINT USING, in the form of PRINT# USING. All the various syntax of PRINT USING are the same for the graphics modes, as for the text mode.

Note that ATN(1)*4 = Pi = 3.14159 . . .

How to Print Using Colour in Graphics MODE 2

One of the advantages of printing in a graphics mode is that you can use all 16 colours for printing characters as well as for the graphics. To print in a specified colour, all you have to do is to specify the current foreground colour before PRINT#.

Here is an example to illustrate 16 colour printing.

```
10 COLOR 1,15,15
20 SCREEN 2
30 OPEN "GRP:" AS #1
40 FOR I = 0 TO 15
50 COLOR I
60 DRAW "BM+8,0"
70 PRINT#1,"COLOUR CODE";I
80 NEXT I
90 GOTO 90
```

LINE 10 BACKGROUND COLOUR IS WHITE
LINE 20 SCREEN MODE 2
LINE 30 OPENS FILE TO GRP AS #1
LINE 40 LOOP
LINE 50 COLOUR OF FIRST LOOP
LINE 60 MOVES GRAPHICS CURSOR
LINE 70 GIVES MESSAGE
LINE 80 NEXT LOOP
LINE 90 LOCKS THE SCREEN

The program above will result in the following display.

COLOUR CODE 0	TRANSPARENT
COLOUR CODE 1	BLACK
COLOUR CODE 2	MEDIUM GREEN
COLOUR CODE 3	LIGHT GREEN
COLOUR CODE 4	DARK BLUE
COLOUR CODE 5	LIGHT BLUE
COLOUR CODE 6	DARK RED
COLOUR CODE 7	CYAN
COLOUR CODE 8	MEDIUM RED
COLOUR CODE 9	LIGHT RED
COLOUR CODE 10	DARK YELLOW
COLOUR CODE 11	LIGHT YELLOW
COLOUR CODE 12	DARK GREEN
COLOUR CODE 13	MAGENTA
COLOUR CODE 14	GREY
COLOUR CODE 15	WHITE

How to Print in Multi-Colour Graphics MODE 3

In the multi-colour MODE 3, the text is four times the size of normal mode because the graphic display is in low resolution. Hence in order to PRINT characters, block graphics must be used, and the resulting characters are huge. Here is a demonstration program.

```
10 SCREEN 3
20 OPEN "GRP:" AS #1
30 DRAW "BM0,0"
40 PRINT#1,"PI"
50 DRAW "BM0,65"
60 PRINT#1, USING"#.######";ATN(1)*4
70 GOTO 70
```

LINE 10 SELECTS MULTI-COLOUR MODE
LINE 20 OPENS A FILE TO THE SCREEN
LINE 30 POSITIONS THE CURSOR
LINE 40 PRINTS "PI"
LINE 50 REPOSITIONS THE CURSOR
LINE 60 PRINTS THE VALUE OF Pi
LINE 70 LOCKS THE GRAPHICS SCREEN

CHAPTER 15

ADVANCED GRAPHICS IV

SPRITE GRAPHICS

The MSX's video display processor has a facility to produce sprites. Sprites are user-definable characters or shapes, which can be displayed on the screen without affecting the backdrop graphics screen. This is because the sprites are displayed in different sprite screen planes. They can be made to hide behind one another and move about without causing flicker. It gives you power to create arcade games without too much trouble with the graphics.

Sprite Size

There are four sprite sizes you can use, but only one sprite size may be used at once. The size is determined using the SCREEN statement.

	SIZE	
SCREEN ,0	8 by 8	pixel unmagnified
SCREEN ,1	8 by 8	pixel magnified to 16 × 16
SCREEN ,2	16 by 16	pixel unmagnified
SCREEN ,3	16 by 16	pixel magnified to 32 × 32

Note that magnification gives you 2 × 2 pixel resolution.

Defining Sprites: SPRITE$

You can define your sprite using SPRITE$. You can have up to 256 sprite patterns when you are using sprite size 0 or 1 (8 by 8 pixel), or 64 of them in sprite size 2 or 3 (16 by 16 pixel).

Syntax

Sprite$ (<interger>)=<string>

<integer> = sprite pattern number
<integer> = 0 to 255 when sprite size is 0 or 1
<integer> = 0 to 63 when sprite size is 2 or 3

The SPRITE$ is a string. The length of the sprite string is fixed to 32 bytes (or characters), but for small 8 by 8 sprites you only need to define the first 8 bytes.

Each byte in the sprite string represents an 8 pixel row and is assigned to the sprite string by adding a character with the ASCII code equal to the byte. This can be done simply by using the CHR$() function. For example, to assign an 8 by 8 sprite:

SPRITE$(<integer>)=CHR$(<1st row>)+CHR$(<2nd row>)
+CHR$(<3rd row>)+CHR$(<4th row>)+CHR$(<5th row>)
+CHR$(<6th row>)+CHR$(<7th row>)+CHR$(<8th row>)

The CHR$() function can be replaced by an actual character if you know what is.

To define a 16 by 16 sprite, you must define all 32 bytes in the sprite pattern.

Here is a standard way of defining an 8 by 8 sprite.

First plot a drawing of your sprite on a piece of paper.

128	64	32	16	8	4	2	1		BINARY		DECIMAL
.	.	.	1	.	.	.	.	=	&B00010000	=	16
.	.	1	1	.	.	.	.	=	&B00110000	=	48
.	1	1	1	.	.	.	.	=	&B01110000	=	112
1	1	1	1	1	1	1	1	=	&B11111111	=	255
1	1	1	1	1	1	1	1	=	&B11111111	=	255
.	1	1	1	.	.	.	.	=	&B01110000	=	112
.	.	1	1	.	.	.	.	=	&B00110000	=	48
.	.	.	1	.	.	.	.	=	&B00010000	=	16

Then you can form a string expression as follows:

SPRITE$(0)=CHR$(16)+CHR$(48)+CHR$(112)+CHR$(255)+
CHR$(255)+CHR$(112)+CHR$(48)+CHR$(16)

EXAMPLE
Using the binary numbers is the easiest way of defining a sprite because it is easy to see what the sprite is going to look like in the program.

```
10 SCREEN 2,0
20 FOR I=1 TO 8
30 READ A$
40 S$=S$+CHR$(VAL("&B"+A$))
50 NEXT I
60 SPRITE$(0)=S$
70 PUT SPRITE 0,(100,100),15,0
80 GOTO 80
90 REM BINARY DATA
100 DATA 00010000
110 DATA 00110000
120 DATA 01110000
130 DATA 11111111
140 DATA 11111111
150 DATA 01110000
160 DATA 00110000
170 DATA 00010000
```

LINE 10 HIRES SCREEN, NORMAL SIZE SPRITE
LINE 30 READS THE BINARY NUMBER AS A STRING
LINE 40 S$ IS A TEMPORARY STRING
LINE 60 DEFINES SPRITE 0
LINE 70 PUTS SPRITE 0 AT x=100,y=100 COLOUR WHITE IN SPRITE PLANE 0

Result: you see an arrow in the middle of the screen.

We deal with the PUT SPRITE statement in detail, later.

If you could be bothered to calculate the decimal values of each byte, then the program can be written in a much shorter version. The above program can be reduced to the following few lines.

```
10 SCREEN 2,0
20 SPRITE$(0)=CHR$(16)+CHR$(48)
+CHR$(112)+CHR$(255)+CHR$(255)
+CHR$(112)+CHR$(48)+CHR$(16)
70 PUT SPRITE 0,(100,100),15,0
80 GOTO 80
```

```
LINE 10  HIRES SCREEN, NORMAL SIZE SPRITE
LINE 20  DEFINE SPRITE 0
LINE 70  PUTS SPRITE 0 AT x=100,y=100 COLOUR WHITE IN SPRITE PLANE 0
```

How to define a 16 by 16 sprite.

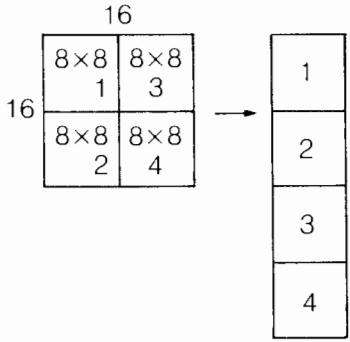
128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
.	.	.	1	1	1	1	1	1	1	1	1	1	.	.	.
.	.	1	1	1	1	1	1	1	1	1	1	1	1	.	.
.	1	1	.	.	.	.	.	.	.	.	.	.	1	1	.
1	1	.	.	.	1	1	1	1	1	1	.	.	.	1	1
1	1	.	.	.	1	1	1	1	1	1	.	.	.	1	1
1	1	.	.	1	.	.	.	.	.	.	1	.	.	1	1
.	1	1	.	1	.	.	.	.	.	.	1	.	1	1	.
.	1	1	.	.	1	.	.	.	.	1	.	.	1	1	.
.	.	1	1	.	.	1	1	1	1	.	.	1	1	.	.
.	.	1	1	.	1	.	.	.	.	1	.	1	1	.	.
.	.	.	1	1	.	1	1	1	1	.	1	1	.	.	.
.	.	.	1	1	.	.	.	.	.	.	1	1	.	.	.
.	.	.	.	1	1	.	.	.	.	1	1	.	.	.	.
.	.	.	.	1	1	.	.	.	.	1	1	.	.	.	.
.	.	.	.	.	1	1	.	.	1	1	.	.	.	.	.
.	.	.	.	.	.	1	1	1	1	.	.	.	.	.	.
.	.	.	.	.	.	.	1	1	.	.	.	.	.	.	.

The diagram above in binary and decimal.

00011111	= 31	11111000	= 248
00111111	= 63	11111100	= 252
01100000	= 96	00000110	= 6
11000111	= 199	11100011	= 227
11001000	= 200	00010011	= 19
01101000	= 104	00010110	= 22
01100100	= 100	00100110	= 38
00110011	= 51	11001100	= 204

00110100	= 52	00101100	= 44
00011011	= 27	11011000	= 216
00011000	= 24	00011000	= 24
00001100	= 12	00110000	= 48
00001100	= 12	00110000	= 48
00000110	= 6	01100000	= 96
00000011	= 3	11000000	= 192
00000001	= 1	10000000	= 128

16 by 16 SPRITE patterns are stored in Video RAM in the following way:



A 16 × 16 SPRITE$ = "Sprite quadrant 1" + "Sprite quadrant 2" + "Sprite quadrant 3" + "Sprite quadrant 4"

Here is a fairly efficient program which defines and displays the 16 by 16 bit sprite we have designed above. The data is stored in DATA statements and are added one by one to the SPRITE$(0) in the FOR/TO/NEXT loop.

```
10 SCREEN 2,2
20 FOR I = 1 TO 32
30 READ B%
40 S$=S$+CHR$(B%)
50 NEXT I
60 SPRITE$(0)=S$
70 PUT SPRITE 0,(100,100),15,0
80 GOTO 80
90 DATA 31,63,96,199,200,104,100,51
100 DATA 52,27,24,12,12,6,3,1
105 DATA 248,252,6,227,19,22,38,204
110 DATA44,216,24,48,48,96,192,128
```

LINE 10 HIRES SCREEN MODE 2 WITH 16 BY 16 SPRITES SIZE
LINE 20 LOOP TO READ IN 32 BYTES
LINE 30 READ
LINE 40 ADDS DATA TO S$
LINE 60 DEFINE SPRITE STRING 0
LINE 70 PUTS SPRITE 0 ON PLANE 0 IN WHITE

```
LINE 90   DATA FOR QUADRANT 1
LINE 100  DATA FOR QUADRANT 2
LINE 105  DATA FOR QUADRANT 3
LINE 110  DATA FOR QUADRANT 4
```

How to put a Sprite onto the Screen: PUT SPRITE

There are 32 sprite planes in front of the text/graphics plane, with the sprite plane No 0 right at the very front. This means that the smaller the plane number, the higher the priority. If there are two sprites overlapping, the one with the smallest sprite plane number will be displayed in front, with the other hidden behind.

You may define up to 256 sprite patterns, but only one sprite can be displayed on any particular plane. This limits the maximum number of sprites displayed at any one time to 32.

Also, you can only place a maximum of four sprites per horizontal line.

To place a sprite on to the screen, use the PUT SPRITE statement. It has the following syntax:

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

The <sprite plane number> has a range of 0 to 31.
The <coordinates specifier> tells the computer where to place the top left corner of the sprite.

There are two <coordinates specifier> formats:

1) (<x-coordinate>,<y-coordinate>)
   These specify an absolute position on the screen.

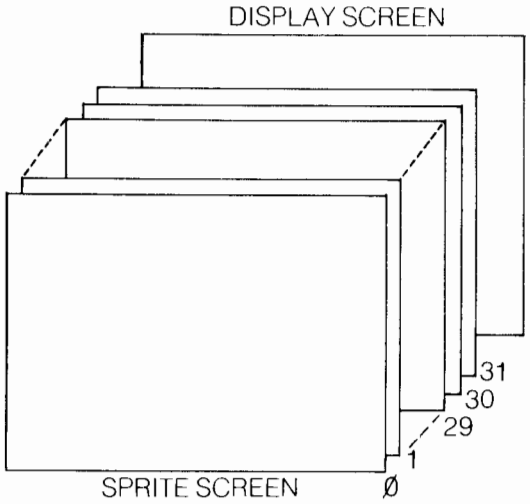
2) STEP (<x-offset>,<y-offset>)
   These coordinates are of a point relative to the last point referred to.

<x-offset>, <y-offset>, <x-coordinates>, <y-coordinates> can be variables, or expressions, as well as just simple numerical constants. This means that the sprites can be controlled by variables allowing you to move them smoothly.

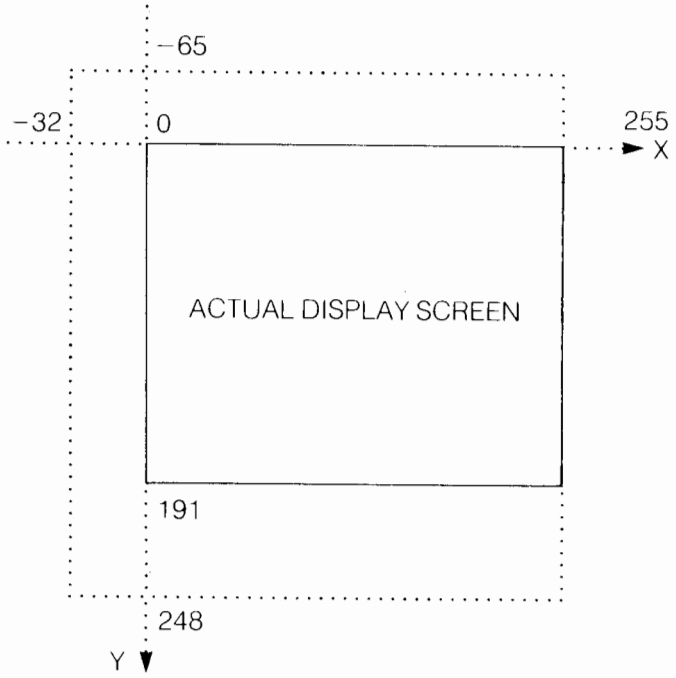
If <coordinates specifier> is omitted, then the computer will PUT SPRITE at the last point referred to.

Examples

```
PUT SPRITE 0,(50,50),1,1
PUT SPRITE 1,STEP (10,10),12,2
```



The Sprite Screen



The sprite screen is slightly larger than the actual display screen. It has a range between −32 and 255 in the X-coordinate and between −65 and 248 in the Y-coordinate. The actual screen coordinates ranges are smaller, so if the sprite is placed outside the actual screen display, then that sprite will be either partially or completely hidden.

The sprites are provided with a complete screen wraparound facility. This means that if a sprite goes beyond the sprite screen, it will appear again on the opposite side of the screen.

How to Move Sprites

By altering the values of the coordinates specifier in the PUT SPRITE statement, you can move the sprites to anywhere on screen. When the PUT SPRITE statement is executed, the previous sprite in the old location is erased automatically.

This example program displays a square moving slowing from right to left, going beyond the screen to the left, and emerging from the right.

```
10 SCREEN 2,0
20 SPRITE$(0)=STRING$(8,CHR$(255))
30 FOR X=200 TO -200 STEP -1
40 PUT SPRITE 0,(X,100),1,0
50 FOR D=1 TO 50:NEXT
60 NEXT
70 END
```

LINE 10 HIRES SCREEN, NORMAL SIZE
LINE 20 DEFINES A SQUARE SPRITE
LINE 40 PUTS THE SPRITE AT X,100
LINE 50 DELAY

Sprite Colour

You can use any of the 16 colours. However, you can only use one colour per sprite, so it is not possible to have a multi-coloured sprite. To simulate a multi-color sprite, place two or more sprites with different colours on top of one another in different sprite planes and move them together.

Note that the background colour of a sprite is always transparent so that you can see through the gap in the sprite.

This example defines two sprites, 0 and 1. The sprite number 0 is displayed in white, and the sprite 1 is in dark yellow. On the display you will see each of the sprites separately, as well as the combination of the two which simulates a multi-colour sprite.

```
10 SCREEN 2,0
20 SPRITE$(0)=CHR$(16)+CHR$(48)
+CHR$(112)+CHR$(255)+CHR$(255)+
CHR$(112)+CHR$(48)+CHR$(16)
30 SPRITE$(1)=CHR$(224)+CHR$(192)
+CHR$(128)+CHR$(0)+CHR$(0)+CHR$(
128)+CHR$(192)+CHR$(224)
40 PUT SPRITE 0,(20,20),15,0
50 PUT SPRITE 1,(40,40),10,1
40 PUT SPRITE 2,(60,60),15,0
40 PUT SPRITE 3,(60,60),10,1
80 GOTO 80
```

LINE 10 HIRES SCREEN, NORMAL SIZE
LINE 20 DEFINES SPRITE 0
LINE 30 DEFINES SPRITE 1
LINE 40 PUTS SPRITE 0 IN WHITE
LINE 50 PUTS SPRITE 1 IN DARK YELLOW
LINE 40 PUTS SPRITE 2 AND 3 ON THE SAME COORDINATES USING 2 COLOURS IN DIFFERENT SPRITE PLANES

How to Hide Sprites

If you give a special number to the y-coordinate, you can disable sprites:

Y=208

If 208 (&HD0) is given to the Y coordinate, all sprite planes with higher sprite plane numbers are disabled until a value other than 208 is given to that plane.

```
10 SCREEN 2,0
20 SPRITE$(0)=STRING$(8,CHR$(255))
30 PUT SPRITE 0,(20,20),15,0
40 PUT SPRITE 1,(40,40),15,0
50 PUT SPRITE 2,(60,208),15,0
60 PUT SPRITE 3,(80,80),15,0
70 PUT SPRITE 4,(100,100),15,0
80 GOTO 80
```

LINE 10 HIRES SCREEN, NORMAL SIZE
LINE 20 DEFINES A SQUARE SPRITE

LINE 50 Y=208 DISABLES SPRITE PLANE 2 ONWARDS. SPRITES AT LINE 60 AND 70 ARE NOT DISPLAYED

If 209 (&HD1) is specified to Y, then that sprite disappears from the screen.

```
10 SCREEN 2,0
20 SPRITE$(0)=STRING$(8,CHR$(255))
30 PUT SPRITE 0,(20,20),15,0
40 PUT SPRITE 1,(40,40),15,0
50 PUT SPRITE 2,(60,209),15,0
60 PUT SPRITE 3,(80,80),15,0
70 PUT SPRITE 4,(100,100),15,0
80 GOTO 80
```

LINE 10 HIRES SCREEN, NORMAL SIZE
LINE 20 DEFINES A SQUARE SPRITE
LINE 30 SQUARE SPRITE ON PLANE 1
LINE 40 SQUARE SPRITE ON PLANE 2
LINE 50 Y=209 DISABLES THIS SPRITE
LINE 60 SQUARE SPRITE ON PLANE 3
LINE 70 SQUARE SPRITE ON PLANE 4

The 'Fifth' Sprite Rule

There is a maximum limit of four sprites which can be displayed on one horizontal line. If this rule is violated, then only the four sprites with the latest sprite plane numbers will be displayed normally. The rest will not be displayed on that line. The sprite number of the violating fifth sprite can be found from the VDP status register using the VDP function (see VDP function section).

Example

```
10 SCREEN 2,0
20 SPRITE$(0)=STRING$(8,CHR$(255))
30 PUT SPRITE 0,(20,100),15,0
40 PUT SPRITE 1,(40,100),15,0
50 PUT SPRITE 2,(60,100),15,0
60 PUT SPRITE 3,(80,100),15,0
70 PUT SPRITE 4,(100,104),15,0
80 GOTO 80
```

LINE 10 HIRES SCREEN, NORMAL SIZE
LINE 20 DEFINES A SQUARE SPRITE
LINE 30 SQUARE SPRITE ON LINE 100
LINE 40 SQUARE SPRITE ON LINE 100
LINE 50 SQUARE SPRITE ON LINE 100
LINE 60 SQUARE SPRITE ON LINE 100
LINE 70 SQUARE SPRITE ON LINE 104

In the above example, you will see four square sprites, on the same horizontal line, 100. The fifth sprite is on line 104 so it only shows half the square, the other half being hidden due to the fifth sprite rule. If it was on line 108, the entire fifth sprite would be visible.

How to Animate your Sprites

Simple sprite animation can be achieved with a minimum of fuss with the MSX. All you have to do is to swap two sprite patterns rapidly so the characters appear to be changing.

Here is a simple program which shows a space invader with moving legs. The SPRITE$(0) contains the invader with legs closed, and SPRITE$(1) contains the one with legs open.

```
10 SCREEN 2,1
20 SPRITE$(0)=CHR$(60)+CHR$(126)
+CHR$(129)+CHR$(219)+CHR$(126)
+CHR$(36)+CHR$(36)+CHR$(36)
30 SPRITE$(0)=CHR$(60)+CHR$(126)
+CHR$(129)+CHR$(219)+CHR$(126)
+CHR$(36)+CHR$(60)+CHR$(129)
40 PUT SPRITE 0,(100,100),11,0
50 FOR I=1 TO 500:NEXT
60 PUT SPRITE 0,(100,100),11,1
70 FOR I=1 to 500:NEXT
80 GOTO 40
```

LINE 20 INVADER WITH LEGS CLOSED
LINE 30 INVADER WITH LEGS OPENED
LINE 40 SPRITE 0 (LEGS CLOSED) YELLOW
LINE 50 DELAY
LINE 60 SPRITE 1 (LEGS OPENED) YELLOW
LINE 70 DELAY

Demonstration Program

In this example you will see two planets orbiting a Sun. The whole system drifts from the top left-hand-side of the screen.

```
10 SCREEN 2,0
20 COLOR 15,1,1
30 CLS
40 SPRITE$(0)=CHR$(126)+STRING$(6,
CHR$(255))+CHR$(126)
50 SPRITE$(1)=STRING$(3,CHR$(0))+
CHR$(24)+CHR$(24)+STRING$(3,CHR$
(0))
60 FOR I=0 TO 6.28 STEP 0.2
70 X=X+1.5
80 Y=Y+1
90 X1=30*COS(I)
100 Y1=30*SIN(I)
110 X2=15*COS(I)
120 Y2=15*SIN(I)
130 PUT SPRITE 0,(X,Y),11,0
140 PUT SPRITE 1,(X1+X,Y1+Y),9,1
150 PUT SPRITE 2,(X2+X,Y2+Y),15,1
160 NEXT
170 GOTO 60
```

LINE 10 HIRES SCREEN, NORMAL SIZE BLACK BORDER AND BACKGROUND, WHITE FOREGROUND
LINE 30 CLEARS SCREEN
LINE 40 SPRITE 0 IS THE SUN
LINE 50 SPRITE 1 IS A PLANET
LINE 60 LOOP
LINE 70 MOVE SUN IN X BY 1.5
LINE 80 MOVE SUN IN Y BY 1
LINE 90 X1 = X-COORDINATE FOR PLANET 1
LINE 100 Y1 = Y - COORDINATE FOR PLANE 1
LINE 110 X2 = X-COORDINATE FOR PLANET 2
LINE 120 Y2 = Y-COORDINATE FOR PLANET 2
LINE 130 PUT SUN AT NEW POSITION
LINE 140 PUT PLANET 1 AT NEW POSITION
LINE 150 PUT PLANET 2 AT NEW POSITION
LINE 160 NEXT LOOP
LINE 170 LOOP AGAIN

How to Detect Sprite Collisions

The TMS 9918/9929 VDP can detect sprite collisions. This is a useful facility for those arcade games which involve shooting or avoiding the enemy. There are two BASIC statements which deal with sprite collision detection — ON SPRITE GOSUB and SPRITE ON/OFF/STOP.

The ON SPRITE GOSUB statement sets the sprite collision interrupt subroutine. It tells the computer that the sprite collision subroutine is at the specified line number.

The SPRITE ON statement switches on the actual collision detection, which diverts the program to the subroutine specified by the ON SPRITE GOSUB statement when two sprites collide. You must execute the SPRITE ON statement to start the detection.

If any one pixel of two sprites overlaps, that is considered as a collision. This happens even if the overlapping pixels are transparent.

Although the computer can detect a sprite collision, it cannot tell you which two sprites have collided or even where the collision took place. You must program the computer to work this out.

Once the interrupt occurs, i.e. two sprites collide, an automatic SPRITE STOP is executed. This stops the computer calling the subroutine again, if another collision occurs while the computer is executing the current sprite collision subroutine. However, it remembers if there is another sprite collision during the current subroutine, in which case the computer will immediately go to the subroutine again after it has left the current subroutine, unless the current subroutine disables the sprite collision detector altogether by executing SPRITE OFF.

After it leaves the sprite subroutine the computer will automatically execute SPRITE ON to enable the interrupt, unless SPRITE OFF is executed in the subroutine.

The SPRITE OFF statement completely stops the computer from detecting any sprite collision.

Example

In this example, you will see two square sprites, one yellow and another white, approaching each other from opposite sides of the screen. When they collide, ON SPRITE GOSUB will come into effect and make the BASIC jump to the sprite collision subroutine. The SPRITE OFF in the sprite interrupt routine prevents any further detection of sprite collisions. This is included because the two sprites will still be overlapping after the computer executes the subroutine. If SPRITE OFF was not there, the computer will eternally loop back to the sprite subroutine. (Try this routine without SPRITE OFF, and see what happens.)

```
10 ON SPRITE GOSUB 110
20 SCREEN 2,0
30 SPRITE$(0)=STRING$(8,CHR$(255))
40 SPRITE$(1)=STRING$(8,CHR$(255))
50 SPRITE ON
60 FOR I=10 TO 240
70 PUT SPRITE 0,(I,100),11,0
80 PUT SPRITE 0,(250-I,100),15,0
90 NEXT I
100 GOTO 50
105 REM SPRITE COLLISION ROUTINE
110 SPRITE OFF
120 BEEP
130 RETURN
```

LINE 10 SETS TRAP SUBROUTINE TO 110
LINE 20 SETS GRAPHICS SCREEN
LINE 30 SPRITE 0 IS A SQUARE
LINE 40 SO IS SPRITE 1
LINE 50 SPRITE COLLISION DETECTOR ON
LINE 60 LOOP
LINE 70 SPRITE (YELLOW) MOVES FROM LEFT
LINE 80 SPRITE (WHITE) MOVES FROM RIGHT
LINE 90 NEXT LOOP
LINE 100 RESTART THE LOOP
LINE 110 KILLS TRAP (ONCE IS ENOUGH)
LINE 120 BEEP INDICATOR
LINE 130 GO BACK TO THE LOOP

Notes on Sprite Collisions

The computer looks for a sprite collision every time the computer executes a statement. Therefore one cannot say that the computer looks for a sprite collision in any particular time interval.

The VDP function can give you the status of the Sprite collision flag in register 8 of the Video Display Processor. (See the VDP section.)

CHAPTER 16

ADVANCED GRAPHICS V

HOW TO ACCESS THE VIDEO DISPLAY PROCESSOR

VDP Function

The VDP TMS 9929A has 8 write-only registers, and 1 read-only register. You can access these VDP registers from BASIC, using the VDP function.

For the write-only registers, VDP(0) to VDP(7), you can only assign values to them, i.e. VDP(1)=2. There is no way of directly reading these registers from the VDP.

However, the computer keeps a copy of these registers in its systems work area. They are stored in memory location between ROGSAV and RG7SAV. (See the section on System RAM in the BIOS reference section for details.)

VDP(8) is the read-only status register. You may read this register as below:

```
PRINT VDP(8)
```

You should remember that if you assign a wrong number to these registers, the screen can sometimes break up. If this does happen, you should RESET/SWITCH OFF the computer to initialise the VDP. Always make sure you know what you are doing before you use the VDP function.

VDP Registers

WRITE ONLY REGISTERS

Register 0 . . . . . VDP(0)

The register 0 contains two VDP option control bits, BIT 1 (M3) and BIT 0 (EV). BIT 2 to BIT 7 must be zero.

BIT 0 EV: external VDP. It enables/disables input from an external VDP. EV=0 for most MSX.
0 = disable
1 = enable

BIT 1 M3: mode bit 3. M3 is used to select the screen mode and is used in conjunction with register 1. (see register 1)

RO

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	0	M3	EV=0

Register 1 . . . . . VDP(1)

Register 1 contains 7 VDP option control bits. BIT 2 is reserved and should always be zero.

BIT 0 MAG: magnification option for sprites.
0 = unmagnified 1 × 1 pixel resolution
1 = magnified 2 × 2 pixel resolution

BIT 1 SIZE: size selector for sprites
0 = 8 × 8 pixel sprites
1 = 16 × 16 pixel sprites

BIT 2 Reserved = 0

BIT 3 M2: mode bit 2. M2 is used to select the screen mode. (See Bit 4)

BIT 4 MI: mode bit 1. M1 is used to select the screen mode. Combination of M1, M2, and M3 gives the screen mode.

M1	M2	M3	
0	0	0	Text mode 1
0	0	1	High resolution graphics mode 2
0	1	0	Multi-colour graphics mode 3
1	0	0	Text mode 0

BIT 5 IE: Interrupt enable
0 = disables VDP interrupt
1 = enables VDP interrupt

BIT 6 BL: Blank enable/disable. This allows you the switch off the display.
0 = causes the screen display to blank and show the border colour only
1 = enables the screen display

BIT 7 VRAM: selects what kind of video RAM is used. (VRAM=1 for most MSX)
0 = 4K RAM
1 = 16K RAM

R1

B7	B6	B5	B4	B3	B2	B1	B0
VRAM	BL	IE	M1	M2	0	SIZE	MAG

Register 2 . . . . . VDP(2)

The register 2 defines the base address of the Name Table. Its range is between 0 and 15. This register represents the upper 4 bits of the 14 bit Name Table address. Therefore, you should write in the address divided by &H400.

R2

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	NAME	TABLE	ADDRESS	

NOTE: R2 * &H400 = NAME TABLE ADDRESS

Register 3 . . . . . VDP(3)

The register 3 defines the base address of the Colour Table. Its range is between 0 and 255. This register represents the upper 8 bits of the 14 bit Colour Table address. Therefore, you should write in the address divided by &H40.

R3

B7	B6	B5	B4	B3	B2	B1	B0
COLOUR	TABLE	ADDRESS					

NOTE: R3 * &H40 = COLOUR TABLE ADDRESS

Register 4 . . . . . VDP(4)

The register 4 defines the base address of the Pattern, Text or Multi-colour Generator Table, depending on the screen mode. Its range is between 0 and 7. This register represents the upper 3 bits of the 14 bit Pattern/Text/Multi-colour Generator Table address. Therefore, you should write in the address divided by &H800.

R4

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	PAT	GEN	ADD

NOTE: R4 * &H800 = PATTERN/TEXT/MULTI-COLOUR GENERATOR TABLE ADDRESS

Register 5 . . . . . VDP(5)

The register 5 defines the base address of the Sprite Attribute Table. Its range is between 0 and 127. This register represents the upper 7 bits of the 14 bit Sprite Attribute table address. Therefore, you should write in the address divided by &H80.

R5

B7	B6	B5	B4	B3	B2	B1	B0
0	SPRITE ATTRIBUTE TABLE BASE ADD						

NOTE: R4 *&H80 = SPRITE ATTRIBUTE TABLE ADDRESS

Register 6 . . . . . VDP(6)

The register 6 defines the base address of the Sprite Pattern Generator Table. Its range is between 0 and 127. This register represents the upper 3 bits of the 14 bit Sprite Pattern Generator Table address. Therefore, you should write in the address divided by &H800.

R6

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	SPR PAT GE.ADD		

NOTE: R6 *&H800=SPRITE PATTERN GENERATOR TABLE ADDRESS

Register 7 . . . . . VDP(7)

The register 7 is divided into two halves. The upper 4 bits contain the foreground colour of text mode, while the lower 4 bits contain the background colour for text and graphics modes.

Range for the foreground colour is between 0 and 15. Similary, for the background colour.

R7 = <Foreground colour> * &H10 + <background colour>

R7

B7	B6	B5	B4	B3	B2	B1	B0
FOREGROUND COLOUR				BACKGROUND COLOUR			

READ ONLY REGISTER

The Status Register . . . . . VDP(8)

The Status flag contains the interrupt pending flag, the sprite collision flag and the fifth sprite flag.

Interrupt flag F : BIT 7

The Interrupt flag is set to 1 at the end of raster scan of the screen display. It is reset to a 0 after the Status register is read, or when the VDP is externally reset.

Sprite Collision Flag C : BIT 5

The VDP checks if two or more sprites overlap each other by more than one pixel. This is done every 1/50th of a second.
1 = sprite collision
0 = No overlap

Fifth Sprite Flag 5S : BIT 6

5S is set to 1 when there are more than 4 sprites on a horizontal line (lines 0 to 192). Note that the VDP cannot cope with more than 4 sprites on a single line. If there are 5 or more sprites, the fifth sprite will not be displayed.

Fifth Sprite Number

If there are more than 4 sprites on a horizontal line and 5S is set to 1, then the sprite number of the violating fifth sprite is given in the Fifth Sprite Number.

R2

B7	B6	B5	B4	B3	B2	B1	B0
F	5S	C	FIFTH SPRITE NUMBER				

CHAPTER 17

ADVANCED GRAPHICS VI

THE VIDEO RAM

The MSX has 16K of Video RAM which is separate from the main programming memory. It is managed and refreshed by the Video Display Processor and therefore, it is independent of the Z-80 central processor. (Dynamic RAMs used by MSX must be refreshed so that they don't loose their stored data.)

This gives the following advantages:

1) The Z-80 processor does not have to refresh the RAM for the screen display thus saving time.
2) The graphics and display does not use up precious memory from the main RAM which is managed by the Z-80.

The Video RAM is divided up into various sections, each with its unique function.

It is quite possible to access the Video RAM from BASIC using the following statements and functions:

BASE	Gives base address of various tables in Video RAM
VPEEK	Reads content of Video RAM
VPOKE	Writes to the Video RAM

The BASE function gives the current base address position of the various screen tables in the Video RAM. (See BASE for more explicit details.)

TABLE NAME AND LOCATION IN HEX	TEXT MODE 0	TEXT MODE 1	GRAPHICS MODE 2	GRAPHICS MODE 2
Pattern name table	&H0000	&H1800	&H1800	&H0800
Colour table		&H2000	&H2000	
Pattern generator table	&H0800	&H0000	&H0000	&H0000
Sprite attribute table		&H1B00	&H1B00	&H1B00
Sprite generator table		&H3800	&H3800	&H3800

CHAPTER 18

ADVANCED SOUND EFFECTS USING PSG

MSX's sound generator processor, AY-3-8912, has been around for quite a long time. It was developed by General Instruments in the late '70s, and it is still the most popular sound chip around. It is used widely in small 8 bit computer systems and arcade games machines. The reasons for its success are its flexibility, and its ability to generate up to 3 channels of periodic sound and one white noise channel simultaneously. Also, because it is a microprocessor on its own, when the sound is generated the processor does not halt the central processor's operation. Although it is now slightly out of date, there is a lot you can do with this sound generator.

This section deals with the workings of the AY-3-8910 sound generator and the use of the SOUND statement.

As an example, try out the following short program which generates a weird flying-saucer noise.

```
10  SOUND 0,87
20  SOUND 7,62
30  SOUND 8,16
40  SOUND 11,179
50  SOUND 12,45
60  SOUND 13,14
```

Wave Forms generated by PSG

Basically, the PSG can only generate two types of wave form: a periodic square wave and white noise.

periodic wave example

```
10 SOUND 7,62
20 SOUND 8,15
30 SOUND 1,1
40 SOUND 0,28
```

white noise example

```
10 SOUND 7,55
20 SOUND 8,15
```

However, these sound waves can be shaped and modified into complex noises so that they do sound life-like. To do that you have to control the frequency, loudness, tone and envelope of each sound channel using the SOUND statement which accesses the PSG.

PSG Sound Registers

AY-3-8910 PSG has 14 registers which you can write to, using the SOUND statement. The SOUND statement has the following syntax:

SOUND <register-number>,<value to be written>

By writing suitable values to these registers, you can select channels, sound waves, tone frequency, noise, and loudness.

There are 14 registers, and they have the following functions:

Register	Description
0	Fine tune frequency on channel A
1	Coarse tune channel A
2	Fine tune frequency on channel B
3	Coarse tune channel B
4	Fine tune frequency on channel C
5	Coarse tune channel C
6	Frequency of the noise generator
7	Mixer switches on/off channels A, B and C.
8	Volume control for channel A
9	Volume control for channel B
10	Volume control for channel C
11	Envelope period (low)
12	Envelope period (high)
13	Envelope pattern

CHOOSING SOUND FOR EACH CHANNEL: REGISTER 7, THE MIXER

This is the register which switches the sound on/off for all three channels. It can select the source of the sound wave, from the noise generator, or from the individual tone generator for each channel.

Bits 5, 4, and 3 of register 7 switch on the noise generator for C, B, and A channels respectively. 1 means off and 0 means on. Bits 2, 1, and 0 of register 7 switch on the tone generator for channel C, B, and A respectively. So, for example, if you want channel C to emit a tone, bit 2 must be set to 0.

BIT	B7	B6	B5	B4	B3	B2	B1	B0
	/	/	NOISE			TONE		
	/	/	C	B	A	C	B	A
	0	0	1	1	1	0	1	1
			OFF	OFF	OFF	ON	OFF	OFF

The SOUND statement format for the above is:

SOUND 7,&B00111011 , or in decimal, SOUND 7,59

The tone of channel A depends on registers 0 and 1, but more on this later.

If you want channel B to emit white noises, and C and A as above, then the register 7 changes to:

BIT	B7	B6	B5	B4	B3	B2	B1	B0
	/	/	NOISE			TONE		
	/	/	C	B	A	C	B	A
	0	0	1	0	1	0	1	1
			OFF	ON	OFF	ON	OFF	OFF

The SOUND statement format for the above is:

SOUND 7,&B00101011 , or in decimal SOUND 7,43

You can make a channel emit both white noise and tone if you switch both on. Say you want all channels to emit both tone and noise, then you get:

BIT	B7	B6	B5	B4	B3	B2	B1	B0
	/	/	NOISE			TONE		
	/	/	C	B	A	C	B	A
	0	0	0	0	0	0	0	0
			ON	ON	ON	ON	ON	ON

The SOUND statement format for the above is:

SOUND 7,&B00000000 , or in decimal SOUND 7,0

NOTE: Bit B7 and B6 are not used, and therefore should be set to 0.

TONE GENERATOR: REGISTERS 0,1,2,3,4,5

To set the frequency of the tone generator for each channel, you write to registers 0,1,2,3,4,5. Each channel has two registers for fixing the frequency of the tone to be generated.

Register 0 and 1 for channel A.
Register 2 and 3 for channel B.
Register 4 and 5 for channel C.

Register	Range	Description
0	0 - 255	Fine tune frequency on channel A
1	0 - 15	Coarse tune channel A
2	0 - 255	Fine tune frequency on channel B
3	0 - 15	Coarse tune channel B
4	0 - 255	Fine tune frequency on channel C
5	0 - 15	Coarse tune channel C
6	0 - 31	Frequency of the noise generator

Frequency step (f) is calculated as follows:

f = (Register 0) + (Register 1) × 256

This formula does not give you the frequency in Hz., but the higher the frequency number the lower the tone generated.

Therefore, the highest frequency is generated by SOUND 0,0: SOUND 1,0 which gives f=0; and the lowest frequency is given by SOUND 0,255: SOUND 1,15.

To calculate the frequency generated in Hz. use the following formula:

$$f = \frac{178900}{(16 \times Hz)}$$

178900 is the clock frequency of PSG (which may be different for the U.K MSX model)

If you want to be exact about what frequency you want the PSG to generate, you must do the above calculation. No! not with a calculator; use the computer instead.

```
10  INPUT "FREQUENCY IN HERTZ";HZ
20  F=INT (1789800#/(16*HZ))
30  PRINT "FINE TUNE REGISTER 0, OR 2, OR 4 ";F MOD 256
40  PRINT "COURSE TUNE REGISTER 1, OR 3, OR 5 ";INT(F/
    256)
```

The above program should do the trick.

Example:

Generate 2000Hz sound wave from channel A.

```
f=1789800/(16*2000)=55.9
10  SOUND 0,55
20  SOUND 1,0
30  SOUND 7,62      mixer
40  SOUND 8,10      loudness
```

NOISE GENERATOR FREQUENCY: REGISTER 6

To change the frequency of the white noise generator, you write to the PSG register 6.

The range is between 0 and 31: the lower the value, the higher the frequency.

Example:

```
10  SOUND 7,55              channel A noise on
20  SOUND 8,15              loudness on channel A set to 15
30  FOR F=0 TO 31
40  SOUND 6,F
50  FOR I=1 to 200:NEXT     Delay
60  NEXT F
```

You should hear a gradual decrease in the frequency of the white noise generated.

LOUDNESS: REGISTERS 8, 9, 10

The loudness of the sound emited from channels A, B, and C are controlled by registers 8, 9, and 10 respectively.

```
no sound   maximum volume
0 <----------------> 15
REGISTER  8 = CHANNEL A
REGISTER  9 = CHANNEL B
REGISTER 10 = CHANNEL C
```

When 16 is written to these registers, the envelope is enabled. This means that the loudness changes according to the envelope shape and frequency. The envelope is determined by registers 11, 12, and 13.

You should remember that the volume output from a television speaker depends on the volume control on the TV. If the volume control is turned off, you won't hear anything. The volume of sound from the MSX's audio output also depends on the external amplifier.

HOW TO USE ENVELOPES: REGISTERS 11, 12, AND 13

When envelope is not used, the volume of sound generated stays constant at the level fixed by the registers 8, 9, and 10. This is rather boring, as all you hear is a constant pitch of noise. However, Ay-3-8910 PSG has an enveloping facility which can change the volume periodically according to one of the envelope shapes available.

Envelope is enabled by setting volume control registers 8, 9 or 10 to 16. When the envelope is switched on, the volume of sound changes according to registers 11, 12, and 13.

	ENVELOPE OFF	ENVELOPE ON
CHANNEL A REGISTER 8	0 - 15	16
CHANNEL B REGISTER 9	0 - 15	16
CHANNEL C REGISTER 10	0 - 15	16

The shape of the envelope pattern can be selected by setting register 13 to a number between 0 and 15. However, the PSG duplicates some of the patterns, so you only get 8 different envelopes.

The rate of change of volume or envelope period, can be changed using registers 11 and 12; register 12 gives a coarse period change, and register 11, a fine period change.

	range
register 11	0 - 255
register 12	0 - 255

Period = (register 12) + 256 * (register 11)

Note that you can have only one envelope at a time, but all three channels may use the same envelope.

See SOUND in the BASIC reference section for the envelope shapes.

CHAPTER 19

HOW TO USE FILES

In MSX BASIC it is possible to save data, such as the contents of a string array, to cassette, and conversely to read in data. This set of data is called 'a file'. There are a set of BASIC statements and functions which enable you to write to and read from the device enabled.

Maxfiles

Before you actually use a file, it is a good idea to increase MAXFILES, which is the function used to set the maximum number of files allowed at any one time. The default value of MAXFILES is 1 but this is usually not enough since you can only use the cassette as a filing device when MAXFILES is 1. The maximum number for MAXFILES is 15.

Technically speaking, MAXFILES has the effect of increasing the size of the File Control Block in the memory. The File Control Block is used as a work area by the MSX BASIC ROM when you are using files. (See Memory Map. CHAPTER 20.)

OPEN

To tell the computer that you are starting to read or write files, you must use the OPEN statement to declare a specified device opened. It allocates a buffer in the File Control Block and sets the mode of I/O operation for the buffer.

You need to execute OPEN before the following statements:

PRINT#	PRINT# USING
INPUT#	LINE INPUT#
INPUT$(#)	EOF

OPEN "<device name>[<file name>]" [FOR <mode>] AS [#] <file number>

There are four devices supported in the current version of MSX. They are:

CAS:	cassette
CRT:	text screen
GRP:	Graphic screen
LPT:	line printer

Of the above, only CAS: and GRP: have any real use. CAS: is specified when you are using a cassette tape recorder to store or retrieve data, and GRP: is specified when you are trying to write to a Graphics screen. It is not possible to PRINT in screen MODE 2 and 3, so you have to send information to the Graphics screen as a file. A detailed description of how to use files to print to a graphics screen is given in CHAPTER 14.

There are three modes of I/O:

OUTPUT	Specifies sequential output mode, i.e. writing to a device using PRINT# and PRINT# USING.
INPUT	Specifies sequential input mode, i.e. reading from a device using INPUT#, etc.
APPEND	Specifies sequential append mode.

<file number> is an integer whose value is between one and MAXFILES, the maximum number of files. File numbers are given so that when quoted in PRINT# statements, for example, the computer knows which device to read or write to. The <file number> is associated with the file as long as it is OPENed. It must not be one which is already being used in the program.

Example

Let us say we want to OPEN a file to a cassette to write information to it. The OPEN statement will be as follows:

```
OPEN "CAS:INFO" FOR OUTPUT AS #1
```

CLOSE

The opposite function of OPEN is performed by the CLOSE statement. It is a good idea to close a file as soon as you have finished with it.

PRINT#

To write to various devices, we use a special print statement, PRINT#. The "#" sign is followed by the file number which is specified in the OPEN statement.

When used with the cassette as the device, PRINT# has an effect of recording the information given.

PRINT# USING

This is a file version of PRINT USING and has the same effect. Its main use is in PRINTing to the Graphics screen in a specified format.

INPUT#

To read in data from devices other than the keyboard, such as a cassette, we use the INPUT# statement. It has the same effect as the INPUT statement but instead of typing in the information required from the keyboard, it reads in from the device specified by the OPEN statement; in most cases the device is the cassette.

During the input for numeric variables, leading spaces, carriage returns and line feeds are ignored. The same goes for string variables. Quotation marks are also ignored.

INPUT$(#)
LINE INPUT#

These are the file versions of INPUT$ and LINE INPUT statements, respectively (see INPUT#).

EOF

EOF returns whether the end of file has been reached. If the end of file is detected EOF returns −1, otherwise it is always 0.

Example Program

Here is a demonstration program which should give you a rough idea of how to use files.

The program requires you to input five names, and it stores these into a cassette file. Then it asks you to load it back again.

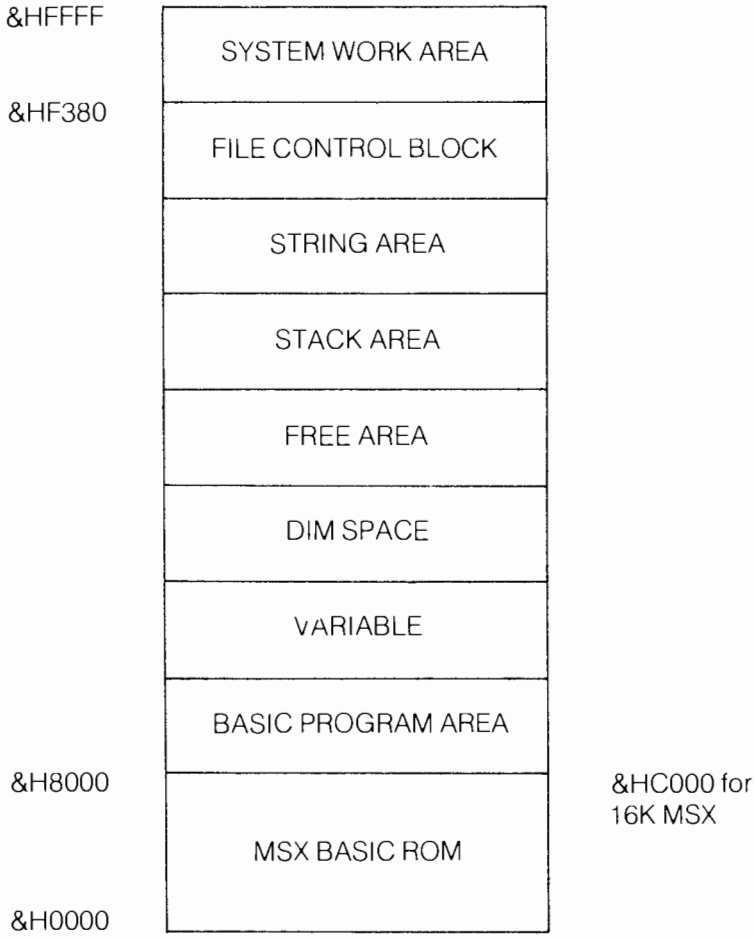
```
10 DIM N$(5)
20 MAXFILES = 4
30 FOR I = 1 TO 5
40 INPUT "PLEASE INPUT A NAME";N$
50 NEXT I
60 PRINT "IS CASSETTE ON? (y=OK)"
70 IF INKEY$<>"y" THEN 70
80 OPEN "CAS:" FOR OUTPUT AS #2
90 FOR I = 1 TO 5
100 PRINT#2,N$(I)
110 NEXT I
120 CLOSE#2
130 PRINT "READY TO LOAD? (y=OK)"
140 IF INKEY$<>"y" THEN 140
150 OPEN "CAS:" FOR INPUT AS #3
160 FOR I = 1 TO 5
170 INPUT#3,N$(I)
180 NEXT I
190 CLOSE#3
200 FOR I=1 TO 5
210 PRINT N$(I)
220 NEXT I
```

LINE 10 ARRAY FOR NAMES
LINE 20 INCREASE THE MAXIMUM FILE NUMBER
LINE 30 INPUT FIVE NAMES SO WE CAN DO SOMETHING WITH THEM
LINE 60 CASSETTE ON (RECORD)
LINE 70 WAITS FOR CASSETTE TO BE ON
LINE 80 OPENS FILE TO CASSETTE
LINE 90 OUTPUT TO CASSETTE USING PRINT#
LINE 120 THEN CLOSE FILE WHEN FINISHED
LINE 130 REWIND CASSETTE SO WE CAN LOAD THE DATA SAVED (PLAY)
LINE 150 OPEN FILE FOR INPUT
LINE 160 READS THE CASSETTE FILE USING INPUT#
LINE 190 CLOSE FILE WHEN FINISHED
LINE 200 PRINTS OUT THE RESULTS

```
RUN
PLEASE INPUT A NAME? TOM
PLEASE INPUT A NAME? PAUL
PLEASE INPUT A NAME? BELLA
PLEASE INPUT A NAME? JANE
PLEASE INPUT A NAME? CHRIS
IS CASSETTE ON (y=ON)                PRESS RECORD
READY TO LOAD (y=OK)      REWIND AND THEN PRESS PLAY
TOM
PAUL
BELLA
JANE
CHRIS
Ok
```

CHAPTER 20

MEMORY MAP



Work Area

The WORK AREA is situated between memory location &HFFFF and &HF380 and is used by the MSX BASIC ROM for its internal operations. This area holds the various system variables, a list of which is given in the BIOS reference section.

File Control Block

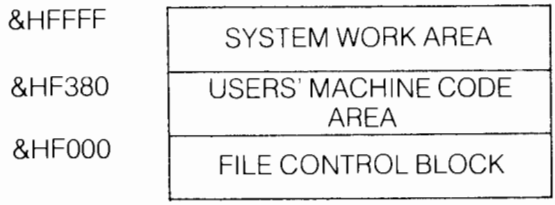
This area is reserved for the input/output operations using files. Statements such as PRINT# and INPUT# use this block.

The size of the File Control Block is altered using the MAXFILES function.

The upper limit of the File Control Block is &HF380. However, you can lower this limit to create a memory area to store machine code programs and data. This is done by executing a CLEAR statement which lowers this limit.

Example: Create a users' machine code area between &HF380 and &HF0000:

```
CLEAR ,&HF000
```



String Area

This area stores the contents of the string variables. The default size of the String Area is 200 bytes. This sets the limit of the maximum length of a string to 200 characters. However, the size can be increased using the CLEAR statement.

Example: Increase the String Area to 255 bytes:

```
CLEAR 255
```

Stack Area

This stores the stacks for FOR/NEXT loops and GOSUBs. It stores the addresses of where the BASIC must return to at a NEXT or RETURN statement.

Free Area

This is the unused area. Its size can be found using the FRE function as follows:

```
PRINT FRE(0)
28815          for 32K or 64K MSX systems at the power up.
12431          for 16K MSX systems at the power up.
```

Note that this area decreases as you increase the size of BASIC program and variables.

DIM Space

This area stores the DIM array variables and is increased upon execution of a DIM statement. If an array is a string array, pointers to the string area will be stored.

Variable Area

This area stores numerical variables and a string descripter pointer to the string area will be stored here.

The address of where the data of a particular variable is stored can be found by using the VARPTR function. (See VARPTR in BASIC REFERENCE section.)

BASIC Program Area

Your BASIC program will be stored here.

CHAPTER 21

USR FUNCTION AND MACHINE CODE

To execute machine code subroutines from BASIC, use the USR function. The USR function calls a machine code program at a location specified in the DEF USR statement. You can use up to 10 machine code routines simultaneously. It is possible to have more than ten of course, by redefining the USR function.

How to Define the USR Function

To call a machine code routine, the computer must know where the start address of the routine is. Use the DEF USR statement to assign the address to a USR function.

Example: say a machine code subroutine starts at &HF000

```
DEF  USR0=&HF000      (you may abbreviate 0
                      so DEF  USR=&HF000)
```

How to Execute a Machine Code Routine

The USR function can be used in the same manner as any ordinary function. This means that they can be in an expression, such as below:

```
X=USR(99)
Y$="M/C"+USR9("abx")
PRINT USR7(0)
```

The above executes the subroutine and returns a value depending on what the machine code does. The numerics or strings in the brackets are parameters to be passed to the machine code. Hence the way you use the USR function really depends on your machine code routine. You

can have an argument in it, or just a dummy parameter; it can also return a parameter or nothing.

If you just want to execute machine code without entry parameters or output, then use a dummy such as below:

DUMMY=USR0(0)
where DUMMY is a dummy variable and
(0) is a dummy parameter.

How to Pass a Parameter to a Machine Code Routine from BASIC

You may pass any type of parameter to machine code from BASIC using USR<number>(<parameter>). The parameter must be enclosed in a set of brackets. When MSX executes the function it checks the kind of argument used, i.e. whether the parameter is integer, single precision number, double precision number, or a string.

If there is a parameter, the machine code must know what type it is and where it is located in memory. MSX BASIC sorts out the parameter type before the machine code is executed.

To find out what type of parameter was passed, look at location &HF663:

&HF663 = 2 = 00000010	integer parameter
&HF663 = 4 = 00000100	single precision parameter
&HF663 = 8 = 00001000	double precision parameter
&HF663 = 3 = 00000011	string

Location of the parameter passed.

Integer (&HF663=2)
&HF7F8 = <lower byte>
&HF7F9 = <higher byte>
<integer parameter> = <lower byte>+256*<high byte>

Single Precision Number (&HF663=4)
&HF7F6 =]
&HF7F7 =] data stored in binary
&HF7F8 =] coded decimal from &HF7F6
&HF7F9 =]

Double Precision Number (&HF663=8)
&HF7F6 =]
&HF7F7 =]
&HF7F8 =]
&HF7F9 =] data stored in binary coded
&HF7FA =] decimal from &HF7F6
&HF7FB =]
&HF7FC =]
&HF7FD =]

String (&HF663=3)

&HF7F8 =] low]
&HF7F9 =] high] .... <address1> address of string description block.

<address1> = string length
<address1>+1 = low]
<address1>+2 = high] ....<address2>, string location.

How to Return a Parameter from Machine Code

To return a parameter from machine code to BASIC, set address &HF663 to the appropriate value according to the type of data and store your parameters into the relevant addresses before you exit machine code. The BASIC picks up whatever is in the parameter store and transfers it to a variable after leaving the machine code routine.

CHAPTER 22

MSX MEMORY MANAGEMENT AND CARTRIDGE SLOT MECHANISM

Introduction

MSX's CPU, the Z80 microprocessor, has a 16 bit address bus which can address up to 64K bytes of memory at any one time. 64K byte of memory is divided into a 4 16K 'pages' of memory. MSX can expand its memory by switching to different 'slots' for each page. A 'slot' is basically a 64K memory space and can be thought of as a bank of separate memory.

Physically, a slot can be a cartridge slot or a hardwired bank of memory like the MSX BASIC ROM with its system's RAM.

To select which slot is used for which page, the slot-select register has to be set. More details on this later.

Usually a basic MSX has the following memory configuration.

PAGE	SLOT 0 SYSTEM SLOT	SLOT 1, 2 OR 3 CARTRIDGE SLOT TO BE PLUGGED IN
page 3	16 K RAM for 16 K MSX computer	
page 2	16 K RAM for 32 K MSX computer	ROM cartridge for games software 8 or 16 K ROM, or 16 K expansion RAM for 16 K MSX system
page 1	MSX BASIC ROM	Used for BASIC expansion ROM or disc operating ROM or other languages.
page 0	MSX BASIC ROM (BIOS)	

Cartridge

All MSX computers have at least one MSX standard cartridge slot where you can plug in a variety of devices. Here is a list of possible cartridge devices:

1. Expansion RAM. Usually to expand a 16K machine to 32K.
2. Games program cartridge ROM. The cartridge contains either a machine code or a BASIC program.
3. BASIC extension ROM. This ROM contains routines to expand the MSX BASIC. The ROM routines are accessed using CALL statements. Some cartridges have their own RAM as a work area.
4. I/O device cartridge. This can be a floppy disc controller or printer interface or light pen cartridge. It usually comes with its own utility ROM to control the input/output.

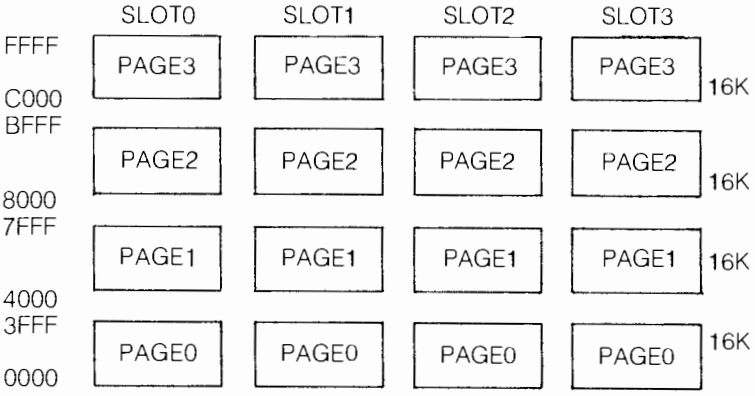
Any of the above can be plugged into any slot. MSX has a slot-select mechanism so it knows which cartridge to access.

Basic Slot Arrangement

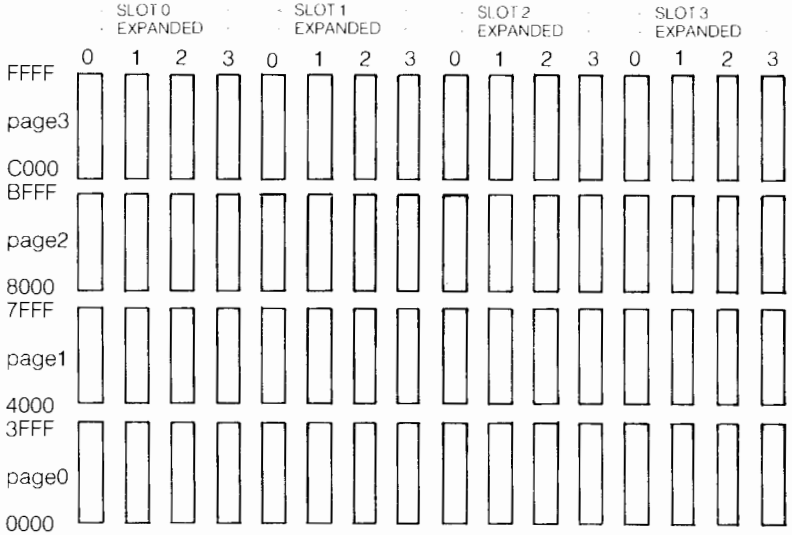
MSX can have four basic slots, slot number 0 being the one with the MSX BASIC ROM, and is called the system slot. Each of these 4 slots can be expanded to 4 expansion slots. Therefore, the total number of slots expandable is 16. These 16 slots can all be 64K of RAM giving you 1M

byte of memory space. This is the maximum RAM MSX can handle. However you should be reminded that you cannot access all 1 M byte from BASIC! You need either MSX-DOS or machine code program to make effective use of memory space larger than 64K.

Basic slot configuration



Expanded slot configuration



Slot Selector

MSX can have several of slots or banks of memory but unless they are controlled by some means, they can interfere with one another. In order

to sort out which slot is to be used for which page, MSX has a special mechanism to select slots.

At any one time, the CPU can access 64k or 4 pages of RAM. These 4 pages can each be in different slots. The pages in slots are selected using the 8255 PPI's 8 bit output port A (see slot selector circuit next page).

PA0 and PA1 give the slot number for page 0.
PA2 and PA3 give the slot number for page 1.
PA4 and PA5 give the slot number for page 2.
PA6 and PA7 give the slot number for page 3.

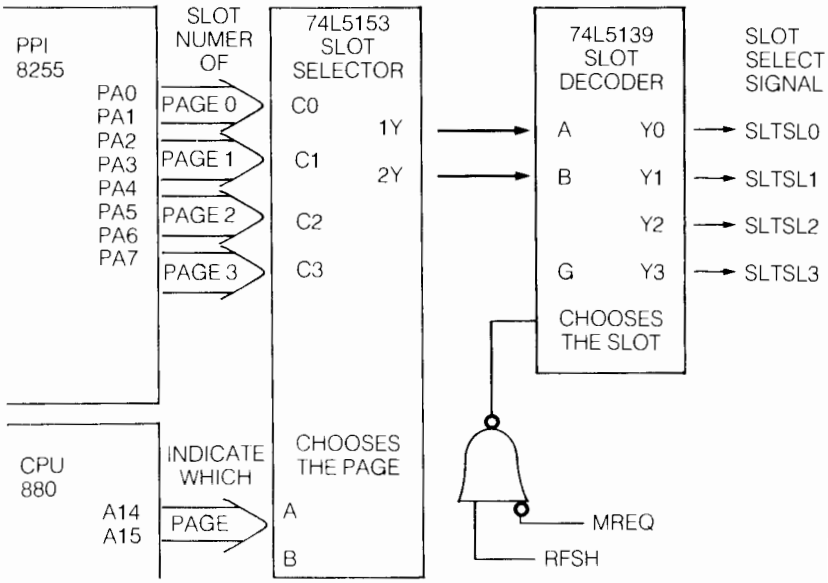
The signal from the PPI is sent to the slot selector 74LS153. This chip also receives signals from the Z80 CPU for the current page the CPU is reading from and writing to.

A15 signal A14 signal: The page number the CPU is reading and writing.

0	0	page 0
0	1	page 1
1	0	page 2
1	1	page 3

The selected slot number is then passed on to the 2 to 4 decoder, 74LS139, which gives the slot select signal, SLTSL, to the slot concerned for the 4 pages.

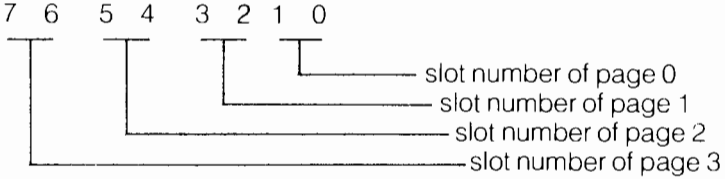
SLOT SELECTOR CIRCUIT DIAGRAM



How to Select and Enable a Slot

To select slots for each page from software, output to I/0 address &HA8, which is the output to port A of the PPI.

The value to be output is an 8 bit number.



Example:
Say you want to use slot 0 for page 0, 1, and 3, and slot 1 for page 2.

Bit No. 7 6 5 4 3 2 1 0
0 0 0 1 0 0 0 0 = 16

You can output 16 to I/0 address &HA8. However a better method is to use the BIOS call ENASLT (&H0024) from machine code.

Expansion of Slots

MSX has the option of having up to 4 basic slots, from 0 to 3. Not all of them are provided in every machine, but each basic slot can be expanded using an expansion box. The maximum number of slots, therefore, is 16. Since all can be filled with up to 64K RAM, the maximum RAM MSX can handle is 1M bytes.

These expansion slots cannot be expanded any further. They are always expanded from a basic slot, so there is no use plugging in an expansion box to an already expanded slot.

To select an expansion slot, you must first select the basic slot where the expansion slot is plugged in, using port A of 8255 PPI. The slot select register is at location &HFFFF in the expanded slot. This determines whether the selected page in that slot is used or not.

To find out whether a basic slot has an expansion box plugged in or not, write to location &HFFFF (POKE &HFFFF, &HOF). If you then read &HFFFF (PRINT PEEK (&HFFFF)) and get a complement of what you have written, this means there is an expansion box there.

Expansion Slot Buffer

Basic slots do not need buffer but the expansion slots do. Therefore, a cartridge slot expansion box requires a two-way data bus buffer within its circuits.

The buffer changes its direction depending on whether a read or write operation is being performed. The control signal which controls the direction of these buffers is BUSDIR. Those devices that send signals to the CPU, such as an I/0 cartridge, have to send the BUSDIR signal to

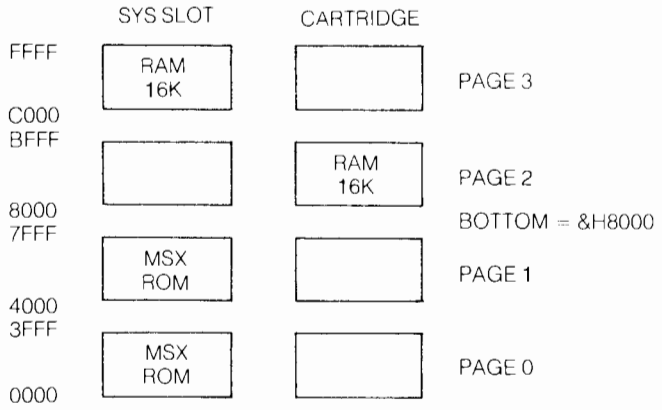
change the direction of the buffer from the expanded slots to the CPU. Those cartridges which contain either ROM or RAM only, do not have to manage the BUSDIR signal.

Ram Search Procedure

When the MSX computer is switched on it searches all the slots to select the system RAM.

1. It first searches for available RAM in page 2, from &H8000 to &HBFFF, then enables the slot with the largest RAM in page 2. If there is more than 1 slot with the same largest RAM space, then the slot with the smallest slot number is selected.
2. Then it does the same for page 3, from &HC000 to &HFFFF. Again, the largest available RAM with smallest slot number is selected.
3. Lastly it checks if the RAM is continuous from &H8000 to &HFFFF, and sets the system variable BOTTOM (&HFC48) to the address of the lowest available RAM.

16 K EXPANSION TO 32K MACHINE

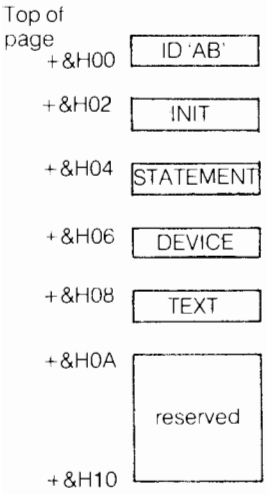


CARTRIDGE ROM SOFTWARE

ROM Search Procedure

Having selected its system RAM, the MSX then searches for ROM cartridges from &H4000 to &HBFFF, page 1 and page 2. It looks for a valid ID at the beginning of each page from slot 0 to slot 3, and the expansion slots when expanded.

The ID area is located at the beginning of a page and may have one of the following formats:



ID ... 2 byte code 'AB' to indicate that there is a ROM cartridge here.
INIT . holds the address of the initialisation routine for that cartridge.
STATEMENT . holds the address of the expanded statement handler.
DEVICE . holds the address of the expanded device handler routine.
TEXT . holds the start address of the BASIC program contained in that cartridge.

Note: ID, INIT, STATEMENT, DEVICE, TEXT will contain zeros if not applicable.

MSX BASIC takes the following action for the cartridge search procedure.

1. Checks the ID area and finds out what type of routine it has. It then passes on the information collected to the relevant work area.
2. Executes the INIT routine if there is one.
3. Executes the BASIC program in that cartridge if there is one.

Note: STATEMENT and DEVICE are not executed at this point because they are only used when the user requires the expanded statement or device.

INIT: Initialisation Routine

INIT holds the address of the initialisation routine specific for the cartridge. It usually initialises I/O devices connected to that cartridge, or sets up a hook (more on this in Hook section), or reserves a work area for the cartridge software.

The initialisation routine can change all registers except the stack pointer [SP]. It returns to BASIC upon the Z80 RET command.

The INIT does not have to be an initialisation routine. It can be a machine code program to be executed immediately upon power up. It could be a game.

If there is no initialisation routine then INIT contains O's.

TEXT: BASIC Program

A ROM cartridge does not necessarily have to be a machine code program: it could be written in BASIC. TEXT holds the start address of the BASIC program contained in the cartridge.

When programming a cartridge software in BASIC, the following notes should be remembered:

1. When there is more than one cartridge with BASIC software plugged in, the machine will only execute the one in the slot with the smallest slot number.
2. The BASIC program in the cartridge is stored in tokenised format.
3. The cartridge must be placed at page 2, &H8000 to &HBFFF. That means that the maximum memory of BASIC cartridge is 16K.
4. RAM placed at page 2 in any other slot is disabled and thus cannot be used from the BASIC cartridge program.
5. The address pointed to by the TEXT entry must contain a zero 0.
6. The line numbers for GOTO and GOSUB etc. should be changed to pointers for faster execution.

If there is no BASIC program in the cartridge, then TEXT contains zeros.

STATEMENT: Expanded Statement Routine

Using the CALL statement in MSX BASIC, you may use expanded statements called from outside the MSX BASIC ROM. A cartridge containing an expanded BASIC statement must contain the start address of the first expanded statement routine in the STATEMENT. The cartridge must be placed at page 1, but can be in any slot except for the system slot where BASIC resides.

The syntax for the expanded statement is:

CALL <statement-name>
CALL <statement-name> (arguments)

CALL can be abbreviated by an underline character '_'.

When BASIC encounters a CALL statement, it stores the expanded statement name in the system work area, PROCNM. PROCNM is located at &HFD89 onwards and is 16 bytes long. The statement name ends with a zero when stored in PROCNM, so the maximum length of an expanded statement is 15 characters.

Before the expanded statement is executed, the text pointer, the HL register, points to the address after the STATEMENT name.

The STATEMENT procedure in the cartridge then checks the content of PROCNM, and executes the routine which matches the statement name.

After it has executed the expanded statement, the carry flag is cleared and the text pointer [HL] is pointed to the position of the next statement.

If there is no expanded statement to match the name in PROCNM, then the carry flag and text pointer remain as they are, and return to BASIC with the 'Syntax Error' message.

If there is no expanded statement routine in the cartridge, then STATEMENT contains zeros.

DEVICE: Expansion Device Handler Routine

MSX BASIC has the ability of plugging in an expansion I/O device to a cartridge slot. The DEVICE contains the start address of the device handler routine.

The following notes should be remembered about DEVICE routines:

1. The cartridge must be placed at page 1, &H4000-&HBFFF.
2. One cartridge (16K) can have up to 4 logical devices connected.
3. Device name is stored in the system area PROCNM, as with STATEMENT. PROCNM is located at &HFD89 onwards and is 16 bytes long. The device name ends with a zero when stored in PROCNM, so the maximum length of an expanded statement is 15 characters.
4. When BASIC encounters a device name, in an OPEN statement, and others, which is not known by the MSX ROM, then it calls device entry with &HFF in register A. If there is no handler which matches the device name, the carry flag is set. If there is a device, ID (from 0 to 3) should be returned in register A and the carry flag reset. All registers can be altered in the device routine.
5. Real I/O operation is carried out when a DEVICE entry is made with one of following values in the A register:

    0 Open
    2 Close
    4 Random I/O
    6 Sequential output
    8 Sequential input
    10 LOC function
    12 LOF function
    14 EOF function
    16 FPOS function
    18 Back up character

    The system variable DEVICE should be set to the Device ID number (0-3).

    If there is no DEVICE routine in the cartridge, then DEVICE should contain zeros.

DESCRIPTIONS OF SYSTEM VARIABLES CONCERNING THE SLOT MECHANISM

Status of each Slot

EXPTBL — indicates which slot is expanded.
Location &HFCC1 — 4 bytes long

EXPTBL	&HFCC1	for slot 0
	&HFCC2	for slot 1
	&HFCC3	for slot 2
	&HFCC4	for slot 3
		&H80 indicates expanded slot
		&H00 not expanded

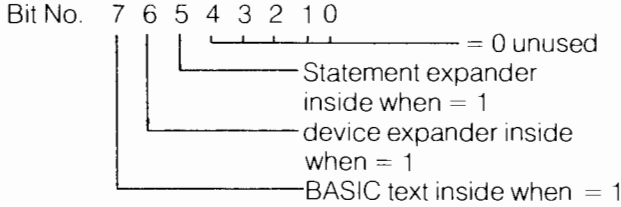
SLTTBL — indicates what value is currently output to the expansion slot select register. This is valid only when the corresponding EXPTBL holds &H80, i.e. when the slot is expanded.
Location &HFFC5 — 4 bytes long

SLTTBL	&HFCC5	for slot 0
	&HFCC6	for slot 1
	&HFCC7	for slot 2
	&HFCC8	for slot 3

Status of each Page

SLTATR — holds what is in each page for all possible pages.
Location &HFCC9 — 64 bytes long

SLTATR	&HFCC9	for basic slot 0 expansion slot 0 page 0
	&HFCCA	for basic slot 0 expansion slot 0 page 1
	. . . .	. . . .
	. . . .	. . . .
	&HFD07	for basic slot 3 expansion slot 3 page 2
	&HFD08	for basic slot 3 expansion slot 3 page 3



SLTWRK — work area for each page. The usage depends on what's in the page but 2 bytes per page allowed.
Location &HFD09 — 128 bytes long; 2 bytes per page

SLTWRK	&HFD09	for basic slot 0 expansion slot 0 page 0
	&HFD0A	for basic slot 0 expansion slot 0 page 0
	&HFD0B	for basic slot 0 expansion slot 0 page 1
	&HFD0C	for basic slot 0 expansion slot 0 page 1
	. . . .	. . . .
	. . . .	. . . .
	&HFD85	for basic slot 3 expansion slot 3 page 2
	&HFD86	for basic slot 3 expansion slot 3 page 2
	&HFD87	for basic slot 3 expansion slot 3 page 3
	&HFD88	for basic slot 3 expansion slot 3 page 3

CHAPTER 23

PERIPHERAL DEVICES

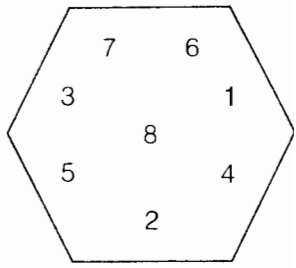
CASSETTE

The following statements and functions are associated with the use of a cassette (detailed descriptions are given in the BASIC reference section).

CLOAD
CSAVE
MOTOR
SAVE
LOAD
BSAVE
BLOAD

Connector: 8 pin DIN plug, type 45326.
Signal table

NO.	SIGNAL	DIRECTION
1	GND	. . . . .
2	GND	. . . . .
3	GND	. . . . .
4	CMTOUT	OUTPUT
5	CMTIN	INPUT
6	REM+	OUTPUT
7	REM−	OUTPUT
8	GND	. . . . .



PRINTER

If your MSX computer is equipped with a printer interface, then you can use any type of printer which is equipped with a Centronics interface. If your computer does not have a built-in printer interface, then you must buy a printer interface cartridge which plugs into the cartridge slot.

Some printers are equipped with a set of MSX standard characters. These MSX compatible printers are available from some of the manufacturers, and give you the advantage of being able to print out all of the MSX graphics characters. A non-MSX printer will not be able to print out the graphics characters; it will instead print out a space for each graphics character.

If you are using an MSX printer, then you must tell the computer that you are using one by using the SCREEN statement:

SCREEN ,,,,0 selects MSX printer
SCREEN ,,,,1 selects non-MSX printer

LPRINT
LPRINT USING

To write to the printer, you must use the LPRINT and LPRINT USING statements, instead of PRINT and PRINT USING. Their syntax is virtually the same except LPRINT and LPRINT USING will not give any screen display. If you want something to be printed on the screen as well as the printer, you must use both PRINT and LPRINT in your program

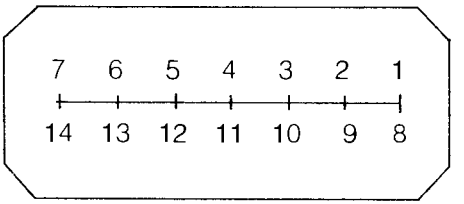
LLIST lists the program to the printer. Its syntax is the same as the LIST command.

The LPOS function returns the position of the printer head, and is analogous to POS function, which gives the cursor position on the text screen.

More details on the printer will be provided to you when you buy your printer. Invariably, the manual will include demonstration programs, written in Microsoft BASIC which is compatible with MSX BASIC.

Connector: Centronics type, 14 pin Aphenol

PIN NO.	SIGNAL NAME
1	PSTB
2	PDB0
3	PDB1
4	PDB2
5	PDB3
6	PDB4
7	PDB5
8	PDB6
9	PDB7
10	NC
11	BUSY
12	NC
13	NC
14	GND



JOYSTICK PORT

Most MSX computers have two joystick ports, while some just have one: the BASIC can support up to two. Apart from joysticks, you can plug in other devices such as touch pads and games paddles.

Connector: AMP 9 pin

Joystick

Joysticks are mainly used for playing games. If your computer has two joystick ports but you only have one, then make sure you plug it into the port marked JOYSTICK 1, as most commercial programs assume that when you are using one joystick it is plugged into this port.

To read the status of the joystick from BASIC, use the STICK and STRIG statements. STICK returns which direction the joystick is in, and STRIG tells you the status of joystick trigger button.

The ON STRIG GOSUB statement provides you with a trigger interrupt. See the Event handling and Interrupt section.

Games Paddle

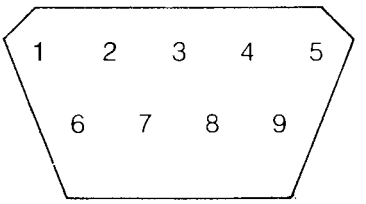
You may plug in up to 12 games paddles, 6 to each of the joystick ports. The status of the paddle can be read using the PDL function (see PDL.)

Touch Pad

You can plug in one touch pad per joystick port.

PAD returns the status of touch pad (see PAD in BASIC reference section).

	SIGNAL	DIRECTION
1	FWD	INPUT
2	BACK	INPUT
3	LEFT	INPUT
4	RIGHT	INPUT
5	+5V	. . . . .
6	TRG 1	INPUT
7	TRG 2	INPUT
8	OUTPUT	OUTPUT
9	GND	. . . . .



SECTION 3

REFERENCE SECTION

CHAPTER 1

BASIC KEYWORDS

EXPLANATION

BASIC KEYWORD FORMAT

This chapter contains all the MSX BASIC keywords in alphabetical order, and everything you need to know about them. The descriptions of the keywords are given in a standard way, making it easy to find their meaning. The description of each keyword is broken down as follows:

KEYWORD

This is sometimes followed by a few words on its derivation.

DESCRIPTION

This section explains what the KEYWORD does in simple terms.

SYNTAX

Every possible syntax is listed with explanations on the actions and the differences of each syntax format.

The following symbols will be used:

<num-const>	is an ordinary number such as 100 or 1.414.
<num-var>	is a numeric variable such as X or AMOUNT.
<num-exp>	means an expression which results in a number such as 2*PI*R*COS(Y). <num-const> and <num-var> can be considered as simple expressions and can be used whenever an expression is required.
<numeric>	can be any of the above.

<string-const>	indicates a string of characters enclosed in quotation marks, e.g. ''MSX COMPUTER 64K''.
<string-var>	means a string variable such as B$ or WORD$.
<string-exp>	means an expression which results in a string e.g. B$+''LONDON BRIDGE'', ''4'', STR$(1234)
<variable>	can be either 1) <string-var> or 2) <num-var>
<condition>	indicates a TRUE or FALSE condition test such as A<0 or B$=''LONDON BRIDGE''.
<statement>	can be any BASIC statement or a group of BASIC statements.
<line>	means the line number.
<name>	can be the name of a program or the name of a function.

EXAMPLES

This section includes various one line examples and short programs to illustrate how the KEYWORD may be used. A brief explanation of the example will be given.

For short example programs the following symbols are used:

```
Anything inside this
square is an example
program which can be
RUN on your computer.
```

```
The result of the example
program will be displayed in
this box. A supplementary
explanation may be given
inside this box on the same line.
```

POINTS TO REMEMBER

This section gives details of the do's and don'ts of the KEYWORD. A comprehensive list of hints and advice will be given, as well as any supplementary description of the KEYWORD.

BUG HUNT

This section gives possible causes of errors associated with the KEYWORD and it is intended to help you debug your program easily.

ASSOCIATED KEYWORDS AND OTHER REFERENCES

This section lists all the associated KEYWORDS which are usually used in conjuction with the KEYWORD.

If details of, or an example use of the KEYWORD is mentioned in any other section, such as the Advanced Programming section, then the list of these chapters will be given.

ABS ABSsolute value

DESCRIPTION

This function returns the absolute value, which is the difference between the number in the brackets and 0, which basically means that all negative numbers become positive, and positive numbers are left as they are. For example, the absolute value of – 2.6 is 2.6.

It is also used to calculate the difference between two numbers.

SYNTAX

ABS(<num-const>)
ABS(<num-var>)
ABS(<num-exp>)

EXAMPLES

```
10 PRINT "ABSOLUTE VALUE OF -1000 IS ";
   ABS(-1000)
20 A=1984
30 B=1960
40 PRINT ABS(B-A)
```

```
ABSOLUTE VALUE OF -1000 IS 1000
 24
```

POINTS TO REMEMBER

ABS is a function whose argument and result are both numeric.

The ABS value of an undefined variable is zero.

BUG HUNT

Mixing of an ASB function with a string will result in a Type mismatch error.

ASSOCIATED KEYWORDS AND REFERENCES

SGN
Section 1 Chapter 15 : FUNCTIONS

AND logical AND operator

DESCRIPTION

AND is one of the logical operators which performs a multiple condition test using Boolean algebra. Boolean algebra is a very necessary part of computer science. For more details, look up the section on Boolean Algebra.

AND is often used within the IF .. THEN .. ELSE statements to test more than two conditions before resultant action is taken. For example:

IF A=5 AND B$="GODZILLA" THEN PRINT "AAARGH!" ELSE PRINT "OK!"

The Boolean algebraic operation of AND can be listed in a truth table as below:

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

Therefore 100 AND 50 can be calculated as follows:

	X	100	0000000001100100
AND	Y	50	0000000000110010
		32	0000000000100000

SYNTAX

IF <condition> AND <condition> ..... THEN ..... ELSE .....

<num-var>=<num-const> AND <num-const>
<num-var>=<num-var> AND <num-exp>
or other numerical combinations.

EXAMPLES

Example (1) of IF THEN ELSE condition testing:

```
10 T=12 : HUNGER$="VERY"
20 INPUT "DO YOU WANT LUNCH (Y/N)";AN$
30 IF T=12 AND HUNGER$="VERY" AND AN$="Y"
   THEN PRINT "HAVE SOME SANDWICHES" :
   END
40 PRINT "FINE. HAVE SOME LATER THEN."
```

RUN this program twice. See what happens when you enter yes (Y) and when you enter no (N).

```
RUN
DO YOU WANT LUNCH (Y/N)? Y
HAVE SOME SANDWICHES
Ok
RUN
DO YOU WANT LUNCH (Y/N)? N
FINE. HAVE SOME LATER THEN.
```

Example (2) for Boolean algebra:

```
10 A%=185 : B%=85
20 C%=A% AND B%
30 PRINT A%;" AND ";B%;"=";C%
40 PRINT "MULTIPLY THAT BY 100 THEN YOU
   GET";100*(A% AND B%)
```

```
185 AND 85 = 17
MULTIPLY THAT BY 100 THEN YOU GET 1700
```

POINTS TO REMEMBER

Logical AND can be used for bit testing a particular number. For example if $Y = 2^n$, where n is the test bit between 0 and 7, then X AND Y = Y if the test bit n is true (i.e.1) and X AND Y = 0 if the test bit n is false (i.e.0).

BIT TEST the 5th bit (n=5) for X=117.

Y=2 ^ 5=32

	X	117	0000000001110101
AND	Y	32	0000000000100000
		32	0000000000100000

therefore true.

Boolean operands must be in the integer range of – 32768 to 32767. Real numbers are truncated to integers.

BUG HUNT

An overflow error occurs when one of the operands lies outside the integer range.

A Type mismatch error occurs when one of the operands is a string.

ASSOCIATED KEYWORDS AND REFERENCES

IF
THEN
ELSE
OR
EQV
XOR
NOT
IMP
Section 1 Chapter 6 : THE USE OF CONDITIONS
Section 2 Chapter 6 : BOOLEAN ALGEBRA
Section 2 Chapter 7 : BOOLEAN II : THE IF THEN ELSE

ASC ASCII code

DESCRIPTION

Every character has a corresponding code number, called ASCII code. (ASCII stands for American Standard Code for Information Interchange.) The computer converts all strings into numbers because it understands numbers better than strings. The ASCII standard was set up so that all computers using the ASCII code have the same number codes for their alphanumeric characters. This makes communication between computers easier.

It is sometimes necessary to find out the ASCII code when programming; ASC converts a particular character or the first character of a string into the corresponding ASCII code.

SYNTAX

ASC(''<string>'')	gives the ASCII code of the first character of the <string>. The <string> can be more than one character long.
ASC(<string-var>)	gives the ASCII code of the first character of the string variable.

EXAMPLES

PRINT ASC(''A'')
will result in the display of ASCII code of ''A'' which is 65.

```
10 A$="FRIDAY"
20 B=ASC(A$)
30 PRINT "ASCII VALUE OF THE FIRST
   LETTER OF ";A$;" IS";B
```

The above program will give:

```
ASCII VALUE OF THE FIRST LETTER OF
FRIDAY IS 70
```

POINTS TO REMEMBER

All characters, including the control codes such as <carriage return> and the graphics characters, have their own unique ASCII value and these can all be calculated using ASC.

Only the first character of a string is significant in ASC. The remaining characters are ignored.

The reverse process of generating a character from an ASCII number can be achieved using CHR$, but this cannot give more than one character at a time.

BUG HUNT

If a string variable has nothing in it, i.e. if it is a null string, this will result in an Illegal Function Call error.

For example, 10 PRINT ASC(" ") will result in an Illegal Function Call at 10.

A type mismatch error occurs if the argument is numeric.

ASSOCIATED KEYWORDS AND REFERENCES

CHR$
SeeAPPENDIX for the ASCII character code table.
Section 1 Chapter 20 : THE ASCII CODES

ATN arc-tangent

DESCRIPTION

This function returns the arc-tangent of the argument in radians between −PI/2 and PI/2

SYNTAX

ATN(<num-const>)
ATN(<num-var>)
ATN(<num-exp>)

ATN will give the arc-tangent in radians in double precision.

EXAMPLES

```
PRINT ATN(1.5)
  .98279372324731
PRINT ATN(3/6)
  .46364760900081
```

PI=4*ATN(1) .... 4*ATN(1) calculates the value of PI to 14 decimal places.

POINTS TO REMEMBER

ATN always gives a double precision number.
The argument can be any numeric type.

BUG HUNT

As it is a trigonometry function, it should obviously not be mixed with strings. If it is, a Type mismatch error will result.

ASSOCIATED KEYWORDS AND REFERENCES

TAN
SIN
COS
Section 1 Chapter 19 : MATHEMATICAL FUNCTIONS

AUTO AUTOmatic line numbering

DESCRIPTION

BASIC programs require a line number for every program line. When typing a long program, the AUTO command will put you into the automatic line numbering mode which will offer you a new line number every time you press the <RETURN> key.

This makes life a lot easier for the programmer. The line numbers will have a constant step size.

SYNTAX

AUTO	will give line numbers from 10 in steps of 10.
AUTO <num-const>	will give line numbers from the specified number in steps of 10.
AUTO <num-const>,<num-const>	will give line numbers from the specified number in steps of the second number.
AUTO,<num-const>	will give line numbers from 0 in the specified step size.
AUTO <num-const>,	will give line numbers from the specified line number with the same step size as previously used.

EXAMPLES

AUTO will give line numbers 10, 20, 30, 40, and so on.

AUTO 1000,5 will give line numbers 1000, 1005, 1010, 1015, and so on.

POINTS TO REMEMBER

Every time the <RETURN> is pressed the computer will offer you a new line number.

There are two ways to exit the AUTO mode:

1) Press <CTRL> and <STOP> keys at the same time.
2) Press <CTRL> and <C> keys at the same time.

The range of line numbers you can have is from 0 to 65529 and you can select the step size from 1 to 65529. When the computer reaches line 65529, it automatically ceases to give you line numbers and puts you into command mode.

If there is an existing line number within the program you are editing, the computer will give you a warning signal in form of a star sign "*" after the line number. If you do not want to ruin that line, press the return key but if you are replacing that line, ignore the warning "*"

The function key (F-2) is initialised to "AUTO" when the computer is switched on so you can get into the AUTO mode at the touch of a button.

The line numbers must be integers.

BUG HUNT

If either the line number or the step size specified is not an integer, then a Syntax error will result.

It is quite easy to erase part of a program unwittingly with the AUTO mode so watch out for the "*" warning.

AUTO can be included within a program but it is rather silly. Don't do it.

ASSOCIATED KEYWORDS AND REFERENCES

RENUM

Section 1 Chapter 7: USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

Section 2 Chapter 1: ADVANCED PROGRAM EDITING

BASE

DESCRIPTION

This system variable gives the current base address position of the various screen tables in the video RAM. BASE can be evaluated or assigned like an ordinary variable, but it should only be used in advanced graphics programs. BASE is a 16 bit integer.

SYNTAX

BASE(<numeric>)

BASE(0)	—	base address of name table of text mode 0.
BASE(1)	—	unused
BASE(2)	—	base address of pattern generator for text mode 0.
BASE(3)	—	unused
BASE(4)	—	unused
BASE(5)	—	base address of name table of text mode 1.
BASE(6)	—	base address of colour table of text mode 1.
BASE(7)	—	base address of pattern generator of text mode 1.
BASE(8)	—	base address of sprite attribute for text mode 1.
BASE(9)	—	base address of sprite pattern for text mode 1.
BASE(10)	—	base address of name table of high resolution graphics mode 2.
BASE(11)	—	base address of colour table of high resolution graphics mode 2.
BASE(12)	—	base address of pattern generator of high resolution graphics mode 2.
BASE(13)	—	base address of sprite attribute table of high resolution graphics mode 2.
BASE(14)	—	base address of sprite pattern table of high resolution graphics mode 2.

BASE(15)	—	base address of name table of multi colour graphics mode 3.
BASE(16)	—	unused.
BASE(17)	—	base address of pattern generator table of multi colour graphics mode 3.
BASE(18)	—	base address of sprite attribute table of multi colour graphics mode 3.
BASE(19)	—	base address of sprite pattern table of multi colour graphics mode 3.

EXAMPLES

```
PRINT BASE(0)
0
```

BUG HUNT

The argument can only be between 0 to 19 and must be an integer. No other numbers are recognised and an Illegal function call will result.

ASSOCIATED KEYWORDS AND REFERENCES

Section 2 Chapter 16: THE ADVANCED GRAPHICS VI: THE VIDEO RAM

BEEP BEEP from the speaker

DESCRIPTION

BEEP gives exactly the same beep as PRINT CHR$(7) for 0.04seconds.

SYNTAX

BEEP

EXAMPLES

BEEP

POINTS TO REMEMBER

BEEP is equal to CHR$(7)

ASSOCIATED KEYWORDS AND REFERENCES

SOUND
PLAY
Section 1 Chapter 28: THE MUSIC MACRO LANGUAGE

BIN$ BINary string

DESCRIPTION

BIN$ returns the binary equivalent of a decimal argument in a string form. It has the range of −32768 to 65535 and the argument must be an integer, or a numeric expression which will result in an integer.

SYNTAX

BIN$(<num-const>)
BIN$(<num-var>)
BIN$(<num-exp>)

EXAMPLES

```
10 PRINT BIN$(255)
20 PRINT BIN$(9999)
```

```
RUN
11111111
10011100001111
```

POINTS TO REMEMBER

If the argument is negative, then two's complement form is used, i.e. BIN$(−n)=BIN$(65536−n).

To convert a binary number to decimal, use the &B prefix to represent a binary number and equate it to a decimal variable such as below:

A%=&B00001111.

BUG HUNT

The argument must not be a string or real number. If it is a string or a real number, a type mismatch error will result.

Overflow error occurs when integer is outside the range of −32678 to 65535.

ASSOCIATED KEYWORDS AND REFERENCES

Section 2 Chapter 5: SURVEY OF NUMBER SYSTEM USED IN MSX

BLOAD LOAD Bytes

DESCRIPTION

BLOAD is used to load machine code programs and data from a specified device such as a cassette. It has the option of automatically executing the loaded machine code program, and also of loading the program to a different address from which it was saved.

BLOAD is used for autorunning machine code software.

At the moment the device option is restricted to a cassette but it will support discs in the future.

SYNTAX

BLOAD "<device-name>:" for loading file with unknown name.

BLOAD "<device-name>:<name>" for loading file with a known name.

BLOAD "<device-name>:<name>",R. This automatically executes the machine code program just loaded. R stands for RUN.

BLOAD "<device-name>:<name>",<num-const>. When the offset <num-const> is specified, all addresses specified in the BSAVE are offset by that value.

BLOAD "<device-name>:<name>",R,<num-const> loads with offset and then executes.

EXAMPLES

```
BLOAD "CAS:GAME",R
BLOAD "CAS:ADVENT",&HOOFF
```

POINTS TO REMEMBER

If the execution address is omitted when the code is BSAVEd, then the computer assumes the execution address is the start address when the program is BLOADed with the R option to auto execute the program.

Most machine code games which are recorded on cassette are likely to be auto executed using this command.

The file name must be less than or equal to 6 characters.

BUG HUNT

Device I/O Error will occur if the file is corrupted on tape.

If you get the file name wrong, obviously you are not going to load it into the computer. This is a common mistake. If you aren't sure of the name then omit the file name, as the computer will then load the first machine code bytes found.

ASSOCIATED KEYWORDS AND REFERENCES

BSAVE
Section 2 Chapter 11: CASSETTE SAVING AND LOADING

BSAVE SAVE Bytes

DESCRIPTION

This saves a chunk of memory at a specified memory on to cassette or disc. You must specify the start address and the end address. This command is used for saving machine code programs and data in the form of bytes. You can also specify the execution address which is used when automatically executing the machine code program when loading with BLOAD.

SYNTAX

BSAVE "<device-name>:<name>",<start address>,<end address>
BSAVE "<device-name>:<name>",<start address>,<end address>,<execution address>

EXAMPLES

BSAVE "CAS:GAME",&HA100,&HA200,&HA1FF
BSAVE "CAS:ADVENT",&HC000,&HCFFF

POINTS TO REMEMBER

The file name must be less than or equal to 6 characters long.

If the execution address is omitted, then the computer assumes the start address is the execution address when the program is BLOADed with the R option, to auto execute the machine code.

BUG HUNT

Good SAVEing of memory can only be achieved if the recording device is reliable. Common sense really. If you can't SAVE properly, check your cassette recorder.

A file name with more than 6 characters will cause a Syntax error.

ASSOCIATED KEYWORDS AND REFERENCES

BLOAD
Section 2 Chapter 11: CASSETTE SAVING AND LOADING

CALL CALL expanded statement

DESCRIPTION

This statement calls an expanded statement held within an expansion ROM cartridge. The abbreviation of call is "_", the underline mark. CALL is ROM cartridge dependent and therefore when used in a BASIC program the program will lose MSX compatibility.

SYNTAX

CALL <name>(<list of arguments>)
_<name>(<list of arguments>)

The list of arguments can be anything. It really depends on what the expanded statement does.

EXAMPLES

CALL SPEECH(loudness%,voice%,"HELLO")

POINTS TO REMEMBER

Some computers use CALL to execute user's machine code routine but this is not the case in MSX. To do that you'll need to use USR.

The maximum length of the expanded statement is 15 characters.

Unless you have an expansion ROM in one of the cartridge slots, this function is useless. Also what CALL does is totally ROM dependent.

BUG HUNT

Do not use this for commercial software or the program will lose MSX compatibility.

A non-existent call name will cause a Syntax error.

ASSOCIATED KEYWORDS AND REFERENCES

Section 2 Chapter 22 : MSX MEMORY MANAGEMENT AND CARTRIDGE SLOT MECHANISM

CDBL — Convert to DouBLe precision number

DESCRIPTION

The CDBL function converts integer and single precision numbers, variables and expressions, into double precision numbers.

SYNTAX

CDBL(<num-const>)
CDBL(<num-var>)
CDBL(<num-exp>)

EXAMPLES

B# = CDBL(A!)
PRINT CDBL(NO%)

POINTS TO REMEMBER

Double precision numbers are 8 bytes long and are accurate to 14 decimal places.

BUG HUNT

A Type Mismatch Error will result if a string argument is used or if CDBL is equated to a string variable.

ASSOCIATED KEYWORDS AND REFERENCES

CINT
CSNG
Section 2 Chapter 3 : TYPE CONVERSION

CHR$ CHaRacter string

DESCRIPTION

CHR$ generates a character from the number given according to the ASCII code. This is complimentary to the ASC function.

Since some of the ASCII codes correspond to control characters, such as clear screen, you can activate these using PRINT CHR$.

SYNTAX

CHR$(<num-const>)
CHR$(<num-var>)
CHR$(<num-exp>)

The above will all return a character string.

EXAMPLES

PRINT CHR$(66) will print the character "B" on the screen.

```
10 C1=65  : REM ASCII code of A is 65
20 C2=72  : REM ASCII code of H is 72
30 C3=84  : REM ASCII code of T is 84
40 PRINT CHR$(C2);CHR$(C1);CHR$(C3)
50 A$=CHR$(C2+32)+CHR$(C1+32)+CHR$(C3+32)
60 PRINT A$
```

If you run the above program, you will find that adding 32 to the ASCII code of capital letters turns them into lower case characters.

```
RUN
 HAT
 hat
```

POINTS TO REMEMBER

When the number inside the brackets is between 0 and 31, CHR$ will act according to the control code as defined by the the ASCII standard. For example, PRINT CHR$(7) will give a beep from the speaker. Look up the control code table. You will find that you can embed various control codes within a string.

When the number inside the brackets is between 32 and 255, CHR$ will give the corresponding ASCII character.

BUG HUNT

If the number inside the brackets is either less than 0, or more than 255, then an Illegal Function Error will result.

Always remember that CHR$ is a string, and therefore cannot be equated to a numeric. Otherwise a Type Mismatch Error will result.

ASSOCIATED KEYWORDS AND REFERENCES

ASC
See APPENDIX for ASCII character code table
Section 2 Chapter 20 : THE ASCII CODES

CINT Convert to INTeger

DESCRIPTION

This function converts single and double precision numbers, variables and expressions, to integer numbers. It has a range between – 32768 to 32767. Anything outside this range will cause problems.

Fractional parts of a real number will be truncated.

SYNTAX

CINT(<num-const>)
CINT(<num-var>)
CINT(<num-exp>)

EXAMPLES

A%=CINT(B!*C#)
PRINT CINT(1234.56789) will give 1234.

POINTS TO REMEMBER

Integer numbers are 2 bytes long.

BUG HUNT

The argument must strictly be within the range of – 32768 to 32767. If not, then an Overflow error will result.

A Type Mismatch Error will result if a string argument is used.

ASSOCIATED KEYWORDS AND REFERENCES

CDBL
CSNG
INT
FIX
Section 2 Chapter 3 : TYPE CONVERSION

CIRCLE draw a CIRCLE

DESCRIPTION

CIRCLE draws the whole or a part of either a circle or an ellipse in the graphics mode. You can specify the position of the centre of the circle, colour, and even how much of the circle you want the computer to draw.

SYNTAX

CIRCLE <coordinate-specifier>,<radius>,<colour>,<start angle>, <end angle>,<aspect ratio>

All can be <num-const>, <num-var> or <num-exp> but within their allowed range. (See below.)

Explanation of <coordinate-specifier>.

There are two ways of specifying the coordinate of the centre, one relative and the other absolute.

The <coordinate-specifier> can be either of the following:

1) (<x-coordinate>,<y-coordinate>)
   These coordinates specify the absolute position of the centre of the circle on the screen.

2) STEP (<x-offset>,<y-offset>)
   These coordinates, <x-offset>, <y-offset>, are relative to the last point referred to by the computer. If the resultant offset is outside the screen, then the computer will draw up to the edges of the screen.

<colour>, <start angle>, <end angle>, and <aspect ratio> are optional.

Range:

<x-coordinate>	−32768 to 32767 : To place the centre on the screen use the range 0 to 255
<y-coordinate>	−32768 to 32767 : To place the centre on the screen use the range 0 to 191
<radius>	0 to 32767 but the circle won't fit in the screen if the <radius> is too large.
<colour>	0 to 15 See colour code section in COLOR.
<start angle>	0 to 2*PI in radians
<end angle>	0 to 2*PI in radians

A "−" minus prefix for the <start angle> and the <end angle> specifies a line to be drawn from the centre to each end.

<aspect ratio> 0 to 32767
<aspect ratio> = (vertical radius)/(horizontal radius)

If the aspect ratio is more than 1, then the ellipse will be elongated vertically.
If the aspect ratio is less than 1, then the ellipse will be elongated horizontally.
When the aspect ratio is unspecified the computer assumes an aspect ratio of 1 thus drawing a circle.
If <aspect ratio> is large all you will get is a vertical line.
If 0 then you will get a horizontal line.

EXAMPLES

```
10 SCREEN 2
20 CIRCLE (100, 100), 50, 8
30 FOR I=1 TO 5
40 CIRCLE STEP (I*5, 0),50,8
50 NEXT
60 GOTO 60 : REM THIS LOCKS THE SCREEN
   MODE
```

POINTS TO REMEMBER

This command only works in graphics modes, so be sure to select SCREEN 2 or SCREEN 3.

If the colour is not specified the computer will draw in the current foreground colour.

The start and end angles are in radians.

BUG HUNT

An Overflow error occurs when the value of the <coordinate specifier> or <radius> is out of range.

An Illegal function call error occurs if you get the colour code number wrong.

An Illegal function call when <start angle> or <end angle> is out of range.

ASSOCIATED KEYWORDS AND REFERENCES

COLOR
SCREEN
Section 1 Chapter 25 : DRAWING CIRCLES AND ELLIPSES

CLEAR CLEAR memory area

DESCRIPTION

CLEAR tells the computer to forget all variables currently resident in memory. This command clears the variable memory space, nullifying all string and numerical variables.

You can also reserve memory space with this command for string variables and machine code in the system work area.

You can reserve memory space by changing the highest memory location available for use by BASIC. The highest memory location on default is &HF380. You can lower this using CLEAR ,<high-mem>. The memory space created can be used to store machine-code programs and data.

SYNTAX

CLEAR <string space>	Sets the size of the string space.
CLEAR ,<high-mem>	Sets the highest memory location available for BASIC.
CLEAR <string space>, <high-mem>	Sets the size of the string space as well as the highest memory location available for BASIC.

EXAMPLES

CLEAR	
CLEAR 255	expands the string work area to 255 bytes.
CLEAR ,&HF300	creates space for user's machine code program between &HF300 and &HF380

POINTS TO REMEMBER

The initial string space is 200 bytes. This sets the maximum length of a string variable to 200 characters. If you want the computer to handle string variables longer than 200 characters, you must first execute a CLEAR 255 statement.

The memory area between &HFFFF and &HF380 is used by the BASIC ROM as the system work space.

ASSOCIATED KEYWORDS AND REFERENCES

Section 1 Chapter 7 : USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS
Section 2 Chapter 2 : CONSTANTS AND VARIABLES
Section 2 Chapter 20 : MEMORY MAP

CLOAD/CLOAD?

LOAD from Cassette

DESCRIPTION

Use CLOAD to load in a BASIC program from a cassette tape. The computer will tell you if it has found a program and whether it is skipping over it or loading the program. You don't need to know the name of the program for, if this is unspecified, CLOAD will load in the first BASIC program found.

Whether the computer successfully loads in a program or not is really dependent on the quality of cassette tape recorder.

CLOAD? (cload question mark) is to verify the BASIC program just loaded from cassette with the one in the computer's memory. The computer will either tell you OK, or give a verify error if the current program is different from the one on tape.

SYNTAX

CLOAD
CLOAD ''<name>''
CLOAD?
CLOAD? ''<name>''

EXAMPLES

Let us say there are two programs called ''SNAKE'', and ''LADDERS'' on the cassette. Then, if you type:
CLOAD<return>

```
CLOAD
Found : SNAKE
OK
```

Try CLOAD ''LADDERS''<return>

```
CLOAD "LADDERS"
Skip  : SNAKE
Found : LADDERS
OK
```

POINTS TO REMEMBER

The program name must not be more than 6 characters long.

When a program is loaded with CLOAD, any program currently held in memory will be deleted.

The function key F7 is CLOAD''.

The recorded BAUD rate can be either 1200 BAUD or 2400 BAUD but the CLOAD will automatically sense which rate the recording is in and loads in accordingly.

If there is an open file (using function OPEN), then CLOAD will automatically close that file and proceed.

BUG HUNT

Device I/O Error occurs when you try to load in a badly recorded program. Alway use a reliable cassette tape recorder.

If you get the file name wrong, then obviously you won't get anything loaded. Be careful with your spelling.

ASSOCIATED KEYWORDS AND REFERENCES

CSAVE
Section 1 Chapter 11 : SAVING YOUR PROGRAM ON TAPE

CLOSE CLOSE channel

DESCRIPTION

This command is used to close an opened channel and is the opposite to the OPEN command. The buffer associated with the specified channel is also released.

SYNTAX

CLOSE ..... close all
CLOSE #<file number>,<file number>

EXAMPLES

CLOSE #2

BUG HUNT

A Bad file Number error results when you enter a non-existent file number in the argument.

ASSOCIATED KEYWORDS AND REFERENCES

OPEN
Section 2 Chapter 19 : HOW TO USE FILES

CLS CLear the Screen

DESCRIPTION

CLS clears the screen in any mode.

In a graphics mode, the screen is cleared to the current background colour.

SYNTAX

CLS

EXAMPLES

The following clears the screen to bright red:

```
10 SCREEN 2
20 COLOUR ,9
30 CLS
40 GOTO 40
```

LINE 10 CHOOSES HI-RES GRAPHICS MODE
LINE 20 SETS BACKGROUND COLOUR TO 9 (BRIGHT RED)
LINE 30 CLEARS THE SCREEN
LINE 40 LOCKS ONTO THIS SCREEN MODE

>Result : Bright red screen.

POINTS TO REMEMBER

The text cursor will be repositioned to the top left hand corner.

When the function keys are in KEY ON mode, then the text screen will still display the function key list at the bottom of the screen after the screen has been cleared with CLS. To get rid of the key list you'll need to execute KEY OFF.

Control L : <CTRL> <L> has the same function as CLS

PRINT CHR$(12) has the same function as CLS, i.e. you can embed a CLS statement within a string.

ASSOCIATED KEYWORDS AND REFERENCES

COLOR
Section 1 Chapter 3 : WRITING A PROGRAM

COLOR colour

DESCRIPTION

This statement sets the colour of background, foreground and border. There are 15 colours and one transparent.

SYNTAX

COLOR <foreground colour> , <background colour> , <border colour>

All can be <num-exp> but must evaluate to between 0 and 15. Also they are all optional so some of the arguments can be missed out.

COLOUR CODE

0	Transparent	8	Medium red
1	Black	9	Light red
2	Medium green	10	Dark yellow
3	Light green	11	Light yellow
4	Dark blue	12	Dark green
5	Light blue	13	Magenta
6	Dark red	14	Grey
7	Cyan	15	White

EXAMPLES

COLOR 11	Sets foreground colour to light yellow.
COLOR 15,,15	Sets border and foreground colour to white.
COLOR 1,5,1	Sets background to light blue and border and foreground to black.

POINTS TO REMEMBER

When the machine is switched on the COLOR is set at COLOR 15,4,7, i.e. a white foreground, light blue background and cyan border.

Function key 1 is predefined as the "COLOR" statement when the machine is switched on. Also F6 is "COLOR 15,4,7" which is the default colour.

The background colour will not change unless the CLS command is executed.

If the colour is not specified in either PSET, or LINE, or CIRCLE, then these graphics commands will draw according to COLOR.

BUG HUNT

The arguments must be between 0 and 15; anything else will cause an Illegal function call error.

ASSOCIATED KEYWORDS AND REFERENCES

DRAW
PRESET
PSET
LINE
CIRCLE
CLS
Section 1 Chapter 22 : COLOUR
Section 2 Chapter 13 : ADVANCED GRAPHICS II : COLOUR IN HIGH RESOLUTION GRAPHICS MODE 2

CONT CONTinue

DESCRIPTION

This command tells the computer to continue the execution of a program from where it left off by either a STOP statement or <CTRL> <STOP>. The computer remembers the last line executed before the <CTRL> <STOP> and carries on from the next statement. If the break was in the middle of an INPUT statement, then the computer carries on from the INPUT.

SYNTAX

CONT

EXAMPLES

```
10 PRINT "GIVE ME YOUR
   BIRTHDAY"
20 INPUT YEAR,MONTH,DAY
30 PRINT "THANK YOU"
```

```
RUN
GIVE ME YOUR BIRTHDAY
?1970
??13                          PRESS <CTRL><STOP>, HERE
BREAK IN 20
OK
CONT
?1970
??12
??1
THANK YOU
OK
```

POINTS TO REMEMBER

CONT will work in the following cases:

1) After the program is stopped by a STOP statement. The program restarts from the next line.

2) After the program is stopped by an END statement. The program restarts from the next line.

3) After the program is stopped by <CTRL> <STOP>. The program restarts from the next line.

BUG HUNT

CONT will not work in the following cases:

1) After the program has been edited.

2) After stopping the printer.

3) After halting cassette input/output commands such as INPUT#.

The resultant error message from the above will be Cant Continue.

ASSOCIATED KEYWORDS AND REFERENCES

Section 1 Chapter 7: USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

COS COSine

DESCRIPTION

COS returns the cosine of an angle.

COS is calculated to double precision.

The argument, i.e. the angle, must be in radians. MSX does not recognise degrees.

SYNTAX

COS(<num-const>)
COS(<num-var>)
COS(<num-exp>)

EXAMPLES

```
PRINT COS(1.4445432)
 1259179846524
```

POINTS TO REMEMBER

COS returns in double precision.
The argument must be in radians.

BUG HUNT

This is a trigonometric function, and therefore does not mix with strings.

A Type Mismatch error will result if you use, for example, COS(A$).

ASSOCIATED KEYWORDS AND REFERENCES

SIN
TAN
ATN
Section 1 Chapter 19: MATHEMATICAL FUNCTIONS

CSAVE Cassette SAVE

DESCRIPTION

CSAVE is used to record a BASIC program which is in the computer's memory on to a cassette tape. The saved program must have a file name which is 6 or less characters long. CSAVE saves the BASIC program in tokens instead of an ASCII file, to save time. The baud rate of either 1200 or 2400 can be specified. If you do not specify the baud rate, the computer will record in 1200 baud.

SYNTAX

CSAVE "<name>"
CSAVE "<name>",<baud rate>

<baud rate>
1 = 1200 baud
2 = 2400 baud

EXAMPLES

CSAVE "ADVENT"
CSAVE "games",2

POINTS TO REMEMBER

The program name must be 6 or less characters long.

The baud rate can also be set by the SCREEN command.

Any programs recorded with CSAVE cannot be MERGEd because the program is recorded in a tokenised format. To merge programs you must record the program using the SAVE command which saves BASIC programs in an ASCII file.

BUG HUNT

The success of CSAVE really depends on the cassette tape and the reliability of the recorder.

Bad recording results in the Device I/O error when you try to load the program using the CLOAD statement.

ASSOCIATED KEYWORDS AND REFERENCES

CLOAD
SAVE
Section 1 Chapter 11: SAVING YOUR PROGRAM ON TAPE

CSNG — Convert to SiNGle precision number

DESCRIPTION

This function converts the argument into a single precision number.

SYNTAX

CSNG(<num-const>)
CSNG(<num-var>)
CSNG(<num-exp>)

EXAMPLES

```
PRINT COS(0.7656)
     .72096670541357
PRINT CSNG(COS(0.7656) )
     .720967
```

POINTS TO REMEMBER

CSNG rounds off to the nearest 6th decimal place.

BUG HUNT

A Type mismatch error results if the argument is a string.

ASSOCIATED KEYWORDS AND REFERENCES

CDBL
CINT
Section 2 Chapter 3: TYPE CONVERSION

CSRLIN CurSoR LINe position

DESCRIPTION

This system variable returns the position of the text cursor.

SYNTAX

CSRLIN

EXAMPLES

PRINT CSRLIN

POINTS TO REMEMBER

CRSLIN is O for the top line.

BUG HUNT

You cannot assign this system variable.
A Syntax error occurs if you try to do so.

ASSOCIATED KEYWORDS AND REFERENCES

POS(0)
Section 2 Chapter 12: ADVANCED GRAPHICS I:
CHARACTERISTICS OF EACH SCREEN MODE

DATA

DESCRIPTION

In DATA statements you can store data required for your BASIC program, which is then accessed via READ statements. You can store any number of strings or numbers which fit into one program line. These data items must be separated by commas. The position of the DATA statements is unimportant as the statements are nonexecutable, i.e. the computer ignores a DATA statement when it encounters one; therefore, they can be placed anywhere in the program.

The READ statement is used to access, sequentially, items held in the DATA statement. The computer remembers which items in the DATA statements have already been used. The next READ statement will start from the first unread item in the statement.

The items contained in a DATA statement can be either numbers or strings. The length of the list, though, must be less than 255 characters to fit into one line. The numeric constants can be an integer, single or double precision number. The string constant need not be surrounded by " " quotation marks unless the string contains commas, colons, semicolons, etc. You cannot store variables or expressions in a DATA statement.

The corresponding variable in the READ statement must be of the same type as the data item, i.e. a string variable for string data.

SYNTAX

DATA <list of constants>

EXAMPLES

```
10 PRINT "LIST OF TELEPHONE NUMBERS"
20 FOR I = 1 TO 4
30 READ NME$,AREA%,TEL%
40 PRINT NME$,AREA%;"-";TEL%
50 NEXT I
60 DATA TOM,123,12345,PAUL,999,22222
70 DATA BELLA,555,11111,JANE,456,10001
```

```
RUN
LIST OF TELEPHONE NUMBERS
TOM              123-12345
PAUL             999-22222
BELLA            555-11111
JANE             456-10001
```

POINTS TO REMEMBER

Quotation marks around string data are required only in the following cases:

1) If the string data contains commas.
2) If the string data contains colons.
3) If the string data contains spaces before or after the characters.

To point to a particular DATA statement, use the RESTORE statement to set the computer to READ from the data in the line referred to in RESTORE.

BUG HUNT

If the character type of the READ variable and DATA item disagree, then the Type Mismatch Error will result. Therefore string variables must read string data and numeric variables, numbers.

If the computer runs out of DATA to READ, then an Out of DATA Error will result.

ASSOCIATED KEYWORDS AND REFERENCES

READ
RESTORE
Section 1 Chapter 12: READING DATA INTO ARRAYS

DEFDBL DEFine DouBLe precision

DESCRIPTION

The DEFDBL statement declares that any variable name beginning with the specified range of characters is to be treated as a double precision number.

SYNTAX

DEFDBL<range(s) of letters>

EXAMPLES

5 DEFDBL A,B,C ..... double precision for variables starting with A, B and C.

POINTS TO REMEMBER

The declaration of variable type by #, $, %, and ! takes precedence over DEFDBL.

ASSOCIATED KEYWORDS AND REFERENCES

DEFINT
DEFSNG
DEFSTR
Section 2 Chapter 2: CONSTANTS AND VARIABLES

DEF FN DEFine a FuNction

DESCRIPTION

Using DEF FN you can define your own function which you can call by name anywhere in the program. It is designed to save memory on expressions which turn up more than once in one program.

You can give your function any name you like as long as it does not clash with reserved words.

The DEF FN statement must be executed to let the computer know that the function exists, before the function is called upon.

You can pass parameters into the function to be processed. Any variables inside the function are local, i.e. their value will remain unchanged ouside the function.

To call a function, you must attach FN in front of the name of the function.

The DEF FN function is limited to one line.

SYNTAX

DEF FN<name>=<expression>
DEF FN<name>(<parameters>)=<expression>

<name> must have a "$" suffix for string function. Other types of specifiers can be used, if necessary.

EXAMPLES

```
10 DEF FNCUBE(X)=X^3
20 INPUT "GIVE ME A NUMBER";A%
30 PRINT "CUBE OF ";A%;"IS";FNCUBE(A%)
40 B%=FNCUBE(A%+1)
50 PRINT "CUBE OF ";A%+1;"IS";B%
60 C%=FNCUBE(FNCUBE(A%))
70 PRINT "CUBE OF ";FNCUBE(A%);"IS";C%
```

```
RUN
GIVE ME A NUMBER? 3
CUBE OF 3 IS 27
CUBE OF 4 IS 64
CUBE OF 27 IS 19683
```

POINTS TO REMEMBER

The items in the parameter list, when defining the function, can be either numeric, strings, expressions or variables but they must be separated by commas.

You can also include variables which are used outside the function.

You can set up a nested function by using a function in the parameter as in the example above.

BUG HUNT

Most errors associated with DEF FN occur when you call the function. Here are some of them:

1) Undefined user function, if a function is called before the DEF FN has been executed.
2) If the variable type does not match between the variable assigned to the function and the result of the function, then Type Mismatch Error will result, e.g. A$=FNCUBE(2)
3) If there are not enough parameters supplied when calling the function, an error will result.
4) A Type Mismatch Error will result if the parameter types do not match.

ASSOCIATED KEYWORDS AND REFERENCES

Section 1 Chapter 16: DEFINING YOUR OWN FUNCTIONS

DEFINT DEFine INTeger

DESCRIPTION

The DEFINT statement declares that any variable names beginning with the specified range of characters are to be treated as integers.

SYNTAX

DEFINT<range(s) of letters>

EXAMPLES

10 DEFINT I,J,K .. Variables starting with I, J, and K are to be integers.

POINTS TO REMEMBER

The declaration of variable type by #, $, %, and ! takes precedence over DEFINT.

ASSOCIATED KEYWORDS AND REFERENCES

DEFDBL
DEFSNG
DEFSTR
Section 2 Chapter 2: CONSTANTS AND VARIABLES

DEFSNG DEFine SiNGle precision

DESCRIPTION

The DEFSNG statement declares that any variable names beginning with the specified range of characters are to be treated as single precision numbers.

SYNTAX

DEFSNG<range(s) of letters>

EXAMPLES

10 DEFSNG X,Y,Z .... Single precision for variables starting with X, Y and Z.

POINTS TO REMEMBER

The declaration of variable type by #, $, %, and ! takes precedence over DEFSNG.

ASSOCIATED KEYWORDS AND REFERENCES

DEFDBL
DEFINT
DEFSTR
Section 2 Chapter 2: CONSTANTS AND VARIABLES

DEFSTR DEFine STRing

DESCRIPTION

The DEFSTR statement declares that any variable names beginning with the specified range of characters are to be treated as string variables.

SYNTAX

DEFSTR<range(s) of letters>

EXAMPLES

DEFSTR E,R,T

POINTS TO REMEMBER

The declaration of variable type by #, $, %, and ! takes precedence over DEFSTR.

ASSOCIATED KEYWORDS AND REFERENCES

DEFDBL
DEFINT
DEFSNG
Section 2 Chapter 2: CONSTANTS AND VARIABLES

DEF USR — DEFine USeR's machine code routine address

DESCRIPTION

To use machine code subroutines from BASIC, use the USR function. The USR function calls a machine code program at the address specified in the DEF USR statement.

You may define up to 10 USR functions. Each USR should be followed by a digit which is 5 in the example below:

DEF USR5=&HFF80

When you want to call the machine code subroutine defined by DEF USR, use the USR function, i.e.:

PRINT USR5("parameter")

where "parameter" in brackets is the parameter to be passed from BASIC to the machine code.

SYNTAX

DEFUSR=<starting address> .... when [digit]=0
DEFUSR[digit]=<starting address>

<starting address> can be either <num-const>, <num-var>, or <num-exp> but must be an integer.

[digit] must be between 0 and 9.

EXAMPLES

DEFUSR5=&HF300
DEFUSR=&HF300+255

To call USR5

10 DUMMY=USR5(9): (9) is the parameter to be passed to machine code.

POINTS TO REMEMBER

You can have up to ten USR calls at any one time but they can be redefined so you can have as many machine code routines as necessary.

DEFUSR is the same as DEFUSR0

You have to know Z80 machine code fairly well to use DEFUSR and USR.

BUG HUNT

The starting address must be within the RAM and must be an integer. Anything else will result in an error.

ASSOCIATED KEYWORDS AND REFERENCES

Section 2 Chapter 21: USR FUNCTION AND MACHINE CODE

DELETE

DESCRIPTION

This is an editing command which erases part of a program.

SYNTAX

DELETE <line>
DELETE <line>-<line>
DELETE -<line>

EXAMPLES

DELETE 20	deletes line 20.
DELETE 20– 100	deletes lines 20 to 100 inclusive.
DELETE – 100	deletes up to line 100 inclusive.

POINTS TO REMEMBER

When deleting only one line, it is easier to type in the line number and then press <RETURN> rather than using the DELETE command.

The DELETE command cannot be used in a program.

BUG HUNT

An Illegal function call will occur if you point to a non-existent line number.

If you delete part of a program by mistake, you will obviously mess up the program. Make sure that you are not deleting necessary lines.

ASSOCIATED KEYWORDS AND REFERENCES

Section 1 Chapter 7: USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS
Section 2 Chapter 1: ADVANCED PROGRAM EDITING

DIM DIMensional array

DESCRIPTION

The DIM statement is used to set up arrays. An array is a whole group of variables which all have the same name but differ from each other by a subscripted number.

SYNTAX

DIM <variable>(<num-const>)
DIM <variable>(<num-const>,<num-const>,...)
<variable> is the name of the array set up and can be integer, single precision, double precision or a string. <num-const> is the size of the array and must be a positive integer. To set up a multi-dimensional array, place more than one <num-const> in the brackets.

EXAMPLES

DIM A(5,5,5) is a 3 dimensional array
DIM B$(100),C$(100,2),D%(100),E!(100),F#(100)
You can define any kind of array and as many as you like within one DIM statement.

POINTS TO REMEMBER

If an array is not defined by DIM and if the computer encounters such an array, then the computer automatically assumes that there is an array of that name with 10 elements. However, undefined arrays cannot have more that 10 elements or a Subscript wrong error will occur. Looking at this the other way round, it means that you don't have to define a single dimensional array whose number of elements is less than ten, i.e. DIM A%(5) is not necessary!

The maximum dimension of an array is 255.

The minimum value of a subscript is 0. Therefore, the array starts from the zeroth element.

DIM A(5) gives A(0), A(1), A(2), A(3), A(4), A(5) i.e. 6 elements.

When an array is initially defined by a DIM statement, all the elements are set to zero.

BUG HUNT

Subscript out of range error will occur if you don't stick to the limits defined in the DIM statement.

ASSOCIATED KEYWORDS AND REFERENCES

Section 1 Chapter 12 : READING DATA INTO ARRAYS
Section 1 Chapter 13 : DATA HANDLING AND SORTING
Section 2 Chapter 2 : CONSTANTS AND VARIABLES

DRAW

DESCRIPTION

This enables you to draw pictures according to the Graphics Macro Language (GML). The graphics macro language is written in the form of strings, and the DRAW commands tell the computer to draw according to this sub language.

You should remember that there is a graphics cursor which can move anywhere on the screen. The Graphics Macro Language controls the graphics cursor's movements and actions according to how you program in GML.

SYNTAX

DRAW "<string>"
DRAW <string-var>
DRAW <string-exp>

Graphics Macro Language commands:

Up, Down, Left and Right.

U<n>	:	draws up
D<n>	:	draws down
L<n>	:	draws left
R<n>	:	draws right

<n> in each case is the the distance to move. This depends however on the scaling factor S (see later).

How to draw diagonally

E<n>	:	draws diagonally up and right
F<n>	:	draws diagonally down and right
G<n>	:	draws diagonally down and left
H<n>	:	draws diagonally up and left

<n> in this case is slightly different from drawing horizontally and vertically. E<n>, for instance, draws to up <n> pixel and right <n> pixel coordinate, and not to <n> length diagonally.

How to draw to specific coordinates

To draw to specific coordinates on the screen, use the M command which moves the graphics cursor from its last position to the x and y coordinates while drawing.

M<x>,<y>	:	draws to coordinates x,y

You may draw relative from the last point by using the + or − prefixes to the x and y coordinates.

M+<x>,<y> : M−<x>,<y>

M+<x>,−<y> : M−<x>,−<y>

How to move without drawing (Blank command)

There are times when you don't want the computer to draw a line, such as when you are moving the graphics cursor to a specific location. Use the B prefix which stands for Blank. It will stop the computer drawing while moving the graphics cursor.

Prefix B for all above drawing commands;

B : Moves without drawing.

The following combinations can be used:

BU<n> : moves up

BD<n> : moves down

BL<n> : moves left

BR<n> : moves right

BE<n> : moves diagonally up and right

BF<n> : moves diagonally down and right

BG<n> : moves diagonally down and left

BH<n> : moves diagonally up and left

BM<x>,<y> : moves absolute to point with coordinates x,y

BM+<x>,<y> : BM−<x>,<y> : moves relative

BM+<x>,−<y> : BM−<x>,−<y> : by step x and y.

How to return to starting position, after drawing a line

If you want to move back to where you started to draw from after you have drawn a line, use the prefix N. It will return you to the coordinates at which you started.

Prefix N : Draws but returns cursor to the starting point.

The following combinations can be used:

NU<n>	:	draws up, then moves back
ND<n>	:	draws down, then moves back
NL<n>	:	draws left, then moves back
NR<n>	:	draws right, then moves back
NE<n>	:	draws diagonally up and right, then moves back
NF<n>	:	draws diagonally down and right, then moves back
NG<n>	:	draws diagonally down and left, then moves back
NH<n>	:	draws diagonally up and left, then moves back
NM<x>,<y>	:	draws absolute to point with coordinates x,y then moves back
NM+<x>,<y>	:	NM−<x>,<y> : moves relative by step x and y
NM+<x>,−<y>	:	NM<x>,−<y> : then moves back

How to rotate the direction of relative movement

Using the angle command A, you may rotate the axis of the relative movement commands such as U, D, L, R, F, E, G, and H.

Angle Command A

A<n>	:	where <n> can be 0,1,2, or 3.
A0	:	by 0 degrees.
A1	:	rotates anti clockwise 90 degrees.
A2	:	rotates anti clockwise 180 degrees.
A3	:	rotates anti clockwise 270 degrees.

How to set colour within GML

You may use C command to change the colour within GML. You can change the colour of drawing as often as you like within a single DRAW statement.

Colour command C prefix

C<n> : sets the colour when drawing.
<n> is an integer between 0 and 15.

C0	Transparent	C8	Medium red
C1	Black	C9	Light red
C2	Medium green	C10	Dark yellow
C3	Light green	C11	Light yellow
C4	Dark blue	C12	Dark green
C5	Light blue	C13	Magenta
C6	Dark red	C14	Grey
C7	Cyan	C15	White

How to change the scale of drawing

Scale Command S prefix.

S <n> : where <n> can be an integer between 0 to 255.
<scale facter> = <n>/4
Therefore, S1 draws 1/4th of length specified by U, D, L, R etc.
S4 and S0 are the same and result in no scaling.
S8 will set the scale to twice the size of default value (S4).

How to use substrings

X<string-var>;

means execute what's inside the string variable. In graphics macro language you can have a string, which contains the drawing command within a string. The semi-colon is always necessary.

DRAW ''X<string-var>;''

How to use numerical variables within GML

In all the graphics macro commands, <n>, <x>, and <y> can be variables, but they have to be prefixed with ''='' sign and require a semicolon at the end. i.e.:

=<num-var>; =<n>;
=<x>;
=<y>;

Example numerical variable G%
DRAW ''U=G%;''

EXAMPLES

```
10 SCREEN 2
20 A$="R50U50L50D50"
30 DRAW "BM100,150"
40 DRAW "S0XA$;"
50 IF INKEY$="" THEN 50
60 FOR I=1 TO 10
70 DRAW "S=I;XA$;"
80 NEXT I
90 GOTO 90
```

LINE 10 SELECTS HI-RES SCREEN
LINE 20 A$ CONTAINS THE GRAPHICS MACRO LANGUAGE
LINE 30 POSITIONS GRAPHICS CURSOR TO 100, 150
LINE 40 DRAWS A BOX A$ SIZE NORMAL
LINE 50 WAITS FOR A KEY TO BE PRESSED
LINE 70 DRAWS BOX A$ IN SIZE I
LINE 90 HOLDS GRAPHICS SCREEN

Note A$ reads "Right 50 pixels, Up 50 pixels, Left 50 pixels, Down 50 pixels"

POINTS TO REMEMBER

The X command is very useful since you can define part of a drawing which can be incorporated anywhere. The concept is rather like having a graphics subroutine within the graphics command.

The X command will allow you to do a drawing which would exceed 255 characters if placed explicitly in the one DRAW command.

S command: If you change S from 4 then the computer will remember the new value and draws anything after it to that scale. If you want to return to the normal scale, you must put S4 in a DRAW command.

BM moves the graphics cursor to a specified point in the screen without drawing. This is the best way to position an object, though there are many other ways to do it. If the point of origin is not specified by BM, then the computer will start drawing from the last point visited, i.e. from where the graphics cursor finished after the last DRAW command.

You can set the origin by using PSET or the M command.

Once an angle command, A, is executed, all further DRAWing will be rotated unless you put it back to the original direction with another A command. The angle is not set to default upon the end of a DRAW command or even at the end of a program.

The current foreground colour will remain the same after execution of the DRAW command which contains a C command.

BUG HUNT

You can only use DRAW in graphics modes.

If you get the syntax of GML wrong, an Illegal function call error will result.

ASSOCIATED KEYWORDS AND REFERENCES

COLOR
SCREEN
Section 1 Chapter 26: THE GRAPHICS MACRO LANGUAGE

ELSE

DESCRIPTION

ELSE is a part of the IF/THEN/ELSE structure. ELSE tells the computer that if the <condition> in IF is not satisfied then skip the statements after THEN and execute the statements after ELSE.

ELSE GOTO <line> can be abbreviated to ELSE <line>.

Multiple IF/THEN/ELSE statements can be set up as well as nested IF/THEN/ELSE statements. They are only limited in length by the maximum number of characters allowed in one line (which is 255.)

SYNTAX

IF <condition> THEN <statements> ELSE <statements>

IF <condition> THEN <statements> ELSE <line>

EXAMPLES

Simple IF/THEN/ELSE structure.

```
10 INPUT D$
20 IF D$="NORTH" THEN PRINT "YOU MEET
   A NASTY MONSTER!" ELSE
   PRINT "YOU GO ";D$
30 GOTO 10
```

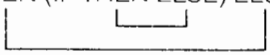
```
RUN
? NORTH
YOU MEET A NASTY MONSTER!
? SOUTH
YOU GO SOUTH
```

POINTS TO REMEMBER

If there are fewer ELSEs than THENs in a nested IF/THEN/ELSE structure, each ELSE is always matched to the closest THEN.

Example:

IF THEN (IF THEN (IF THEN ELSE) ELSE)



IF/THEN/ELSE statements tends to get too long to fit in a single line because you can have multiple statements after THEN and ELSE. If this is the case, then it is a good idea to use subroutines using the GOSUB statement.

IF THEN ELSE can create a spaghetti program if you don't watch out.

BUG HUNT

You tend to find bugs in the <condition> section of IF/THEN/ELSE

ASSOCIATED KEYWORDS AND REFERENCES

AND
OR
NOT
IF
THEN
Section 1 Chapter 6: THE USE OF CONDITIONS
Section 2 Chapter 7: BOOLEAN II : THE IF/THEN/ELSE

END

DESCRIPTION

END tells the computer that it has come to the end of the program and puts it back into command mode.

You may use END statements in any part of a program and as often as necessary.

You may need an END statement in front of a subroutine to prevent the program crashing into it accidentally.

You don't require an END right at the end of a program, because the computer will assume that it is at the end of the program when it runs out of lines to execute.

When the computer encounters an END statement it closes all files opened during the program. This is different from the STOP statement which, although halting the execution of the program, does not close files opened during the program.

SYNTAX

END

EXAMPLES

9999 END

POINTS TO REMEMBER

The difference between STOP and END is that STOP gives a "Break" message but END doesn't. Also, STOP does not close any unclosed files.

BUG HUNT

Let us state the obvious: if you place an END statement where you don't want the program to end, then the program will end prematurely. Right!

ASSOCIATED KEYWORDS AND REFERENCES

STOP

Section 1 Chapter 7: USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

EOF End Of File

DESCRIPTION

EOF tells you if an end of file has been reached. It returns -1 (true) if it has, if not it returns 0 (false).

This is a vital part of file handling and is used with OPEN etc.

SYNTAX

EOF (<file number>)

EXAMPLES

IF EOF(4)=-1 THEN PRINT"END OF FILE":END

BUG HUNT

If you run out of file because you did not use the EOF function, then an "Input Past end" error will result.

If you refer to an unopened file in <file number> then "File not open" error will result.

ASSOCIATED KEYWORDS AND REFERENCES

OPEN
CLOSE
INPUT#
Section 2 Chapter 19: HOW TO USE FILES

EQV equivalent

DESCRIPTION

EQV is one of the Boolean logical operators used in binary manipulation. Its operation can be listed in the following truth table.

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

Example: 51 EQV 74

	51	0000000000110011
EQV	74	0000000001001010
	-122	1111111110000110

SYNTAX

<numeric> EQV <numeric>

<numeric> must be in the integer range between -32768 and 32767. All fractional parts of decimal numbers are rounded off.

EXAMPLES

```
10 A=51
20 B=74
30 PRINT A;" EQV ";B
40 PRINT A EQV B,BIN$(A EQV B)
```

```
RUN
 51 EQV 74
-122          1111111110000110
```

POINTS TO REMEMBER

The following relations are true:

```
 X EQV X  = -1
-X EQV X  =  1
```

BUG HUNT

An Overflow error occurs when the argument is out of integer range (-32768 to 32767).

ASSOCIATED KEYWORDS AND REFERENCES

AND
OR
XOR
IMP
NOT
Section 2 Chapter 6: BOOLEAN ALGEBRA

ERASE ERASE an array

DESCRIPTION

This statement allows you to erase arrays created by DIM statements. This then allows you to redimensionalise the erased array.

SYNTAX

ERASE <name of array>
ERASE <name of array>,<name of array>,.....

EXAMPLES

```
10 A%=99
20 DIM A%(150)
30 FOR I = 1 TO 150
40 A%(I)=I
50 NEXT I
60 ERASE A%
70 DIM A%(1000)
80 FOR I = 1 TO 1000
90 A%(I)=I
100 NEXT I
110 PRINT A%
```

```
RUN
 99
```

POINTS TO REMEMBER

A variable with the same name as one of an array's is not affected by the command, as demonstrated in the above program.

Only the name of the array to be erased is needed in the ERASE statement. This means that you do not have to write ERASE A%(150): ERASE A% will do.

BUG HUNT

ERASEing a non-existent array will cause an "Illegal function call" error.

If you redimensionalise an array without first using ERASE, you will get a "Redimensioned array" error.

ASSOCIATED KEYWORDS AND REFERENCES

DIM
Section 1 Chapter 12: READING DATA INTO ARRAYS

ERL ERror Line

DESCRIPTION

ERL is a reserved variable which contains the line number of the line containing the error found in the program. It is used in error trapping routines.

SYNTAX

<num-var> = ERL
PRINT ERL

EXAMPLES

```
10 ON ERROR GOTO 1000
20 PRINT "Z80"
30 PRINT "TMS 9918"
40 PRONT "MISTAKE"
50 END
1000 E%=ERL
1010 PRINT"A silly mistake at line";E%
1020 END
```

```
RUN
Z80
TMS 9918
A silly mistake at line 40
Ok
```

POINTS TO REMEMBER

If an error occurs when in direct mode, the ERL will contain 65535.
ERL is a reserved variable so you cannot assign a number to it.

BUG HUNT

Use ERL in an error trapping routine to find the exact location of your bugs.

ASSOCIATED KEYWORDS AND REFERENCES

ON ERROR GOTO
ERROR
ERR
Section 2 Chapter 2: ERROR HANDLING

ERR ERRor code number

DESCRIPTION

This is a reserved variable which contains the error code number after the computer has detected an error in a program. It is mainly used in error trapping subroutines.

It has a value between 1 to 255 and it can be used in conjunction with user-defined error messages.

ERR is usually used in creating new errors using the ERROR statement (see example below).

SYNTAX

<num-var>=ERR

EXAMPLES

In this routine all commands inputted with KILL in them will be treated as a new error (No. 255) and are dealt with in the error trapping routine using the ERR function.

```
10 ON ERROR GOTO 1000
20 INPUT "MY LORD, WHAT IS THY COMMAND";A$
30 IF INSTR(A$,"KILL") THEN ERROR 255
40 PRINT "OK.":END
..
..
1000 IF ERR=255 THEN PRINT"KILLING
IS ILLEGAL: END OF PROGRAM": END
1010 END
```

```
RUN
MY LORD, WHAT IS THY COMMAND? KILL HIM
KILLING IS ILLEGAL: END OF PROGRAM
OK
```

POINTS TO REMEMBER

ERR is a reserved variable; therefore you cannot assign a value to it.

BUG HUNT

Use this function to find bugs in your program.

ASSOCIATED KEYWORDS AND REFERENCES

ERL
ERROR
ON ERROR
RESUME
Section 2 Chapter 10: ERROR HANDLING

ERROR

DESCRIPTION

There are basically two ways of using the ERROR statement,

1) To simulate the occurrence of an error.
2) To create user-defined errors using ON ERROR GOTO statements.

1) An ERROR statement with an integer argument will fool the computer into thinking that there is an error, and will terminate the program and print out the error message with the line number.
2) An ERROR statement, together with the use of an ON ERROR GOTO error trapping statement, will let you create your own customised errors.

There are about 36 errors, between error code numbers 1 and 60, in the present version of MSX BASIC. Code numbers 61 to 255 can be used by the programmer.

To program your own error, first of all you must put the computer into the error trapping mode by executing an ON ERROR GOTO <line> statement at the beginning of the program. Then, if in the middle of the program, a condition arises such that you want to call the error trapping routine, you can do so by:

IF <condition> THEN ERROR <error code>.

ON ERROR GOTO will be activated, and the computer will immediately start executing the error trapping subroutine. Within the subroutine you must have a line saying :

IF ERR=<error code> THEN PRINT "Error message"

The error trapping routine can either terminate the program by using the END statement or RESUME operation at any specified line.

SYNTAX

ERROR <error code>
where <error code> = 0 to 255

EXAMPLES

1) A short program containing an error simulation.

```
10 ERROR 11
```

```
RUN
Division by zero in 10
```

2) In this routine all the commands inputted with KILL will be treated as a new error (No. 255) and are dealt with in the error trapping routine using the ERR function.

```
10 ON ERROR GOTO 1000
20 INPUT "MY LORD, WHAT IS THY
   COMMAND";A$
30 IF INSTR(A$,"KILL") THEN ERROR 255
40 PRINT "OK.":END
..
..
1000 IF ERR=255 THEN PRINT"KILLING
IS ILLEGAL IN THIS ADVENTURE.":RESUME 20
1010 END
```

```
RUN
MY LORD, WHAT IS THY COMMAND? KILL HIM
KILLING IS ILLEGAL IN THIS ADVENTURE
MY LORD, WHAT IS THY COMMAND? GOTO EAST
OK
```

POINTS TO REMEMBER

It is the usual practice in MSX BASIC, when defining your own error codes, to work your way down from 255 so that you can avoid clashes with any future enhancement to the error messages made by the manufacturers of MSX computers.

If the error code has no previously defined error message, when the computer encounters this error in the program it will print "Unprintable Error" and halt the program.

No error trapping is carried out during execution of an error trapping routine. If there is an error within the error trapping routine, then it will give the error message and halt the program.

Once the error trapping is enabled, the computer will trap errors in direct mode, too.

ASSOCIATED KEYWORDS AND REFERENCES

ON ERROR GOTO
ERL
ERR
RESUME
Section 2 Chapter 10: ERROR HANDLING

EXP EXPonent

DESCRIPTION

This mathematical function gives the exponential value of the given argument, X.

e^ x

SYNTAX

EXP(<num-const>)
EXP(<num-var>)
EXP(<num-exp>)

EXAMPLES

```
PRINT EXP(1)
  2.7182818284588
```

POINTS TO REMEMBER

This function calculates in double precision up to 14 digits.

BUG HUNT

If the exponent is too large for the computer to handle, an Overflow Error will result.

ASSOCIATED KEYWORDS AND REFERENCES

LOG
Section 1 Chapter 19: MATHEMATICAL FUNCTIONS

FIX Return Integer part

DESCRIPTION

This function returns the integer part of the argument. The fractions are truncated.

FIX is equivalent to SGN(x)*INT(ABS(x))

FIX gives the nearest lower integer for positive arguments and the nearest higher integer for negative arguments.

FIX(1.87) = 1
FIX(– 12.88)= – 12

SYNTAX

FIX(<num-const>)
FIX(<num-var>)
FIX(<num-exp>)

EXAMPLES

PRINT FIX(– 1.2209)
will print – 1
and
A%=FIX(10.99)
will make A%=10

POINTS TO REMEMBER

The INT function gives the nearest lower number with a negative as well as with a positive argument. For example INT(– 1.3)= – 2. So there is a difference between INT and FIX.

BUG HUNT

This is a numeric function so do not mix it up with strings or a Type Mismatch error will result.

ASSOCIATED KEYWORDS AND REFERENCES

INT
Section 1 Chapter 15: FUNCTIONS

FOR FOR/TO/NEXT loop

DESCRIPTION

FOR is one of the words which are used to create loops. The range of the variable counter is defined. The amount by which the counter is incremented each time the loop is completed may be defined by STEP.

For example:

```
10  FOR I=1 TO 3
20  PRINT I
30  NEXT I
```

will give the numbers 1,2,3.

The counter I initially takes on the value 1, and executes the statement between FOR and NEXT, i.e. line 20 PRINT I. When it comes to line 30 NEXT I the computer goes back to line 10, increments the counter I by one and goes through the loop again, until I is equal to the limiting value 3.

So far we have seen the FOR/TO/NEXT format, but we can introduce STEP into the line which contains the FOR statement to control the size of the increment. Therefore, we can modify the above program as follows:

```
10  FOR I=1 TO 3 STEP 0.5
20  PRINT I
30  NEXT I
```

will print 1, 1.5, 2, 2.5, 3.

It is possible to have a nested loop, i.e. loops within a loop, such as below:

```
10  FOR I=1 TO 5
20  FOR J=1 TO 5
30  PRINT I,J
40  NEXT J
50  NEXT I
```

which will print out 1,1 1,2 1,3 ....etc.

There are certain rules for nested loops:

1) Each loop must have a different variable number for its counter.
2) The NEXT for the inside loop must be executed before that of the outside loop.
3) You can have a nested loop as deep as you like. The only limiting factor is the memory.
4) A single NEXT statement with a list of counters can replace a whole series of NEXT statements.

SYNTAX

FOR <num-var> = <numeric> TO <numeric>
FOR <num-var> = <numeric> TO <numeric> STEP <numeric>

EXAMPLES

You can have a negative increment such as shown below:

```
10 FOR F=10 TO 5 STEP -1
20 PRINT F
30 NEXT F
```

```
RUN
10
 9
 8
 7
 6
 5
```

POINTS TO REMEMBER

If you jump out of the middle of a loop, then you must jump back in again. Therefore, you can use GOSUB which will return you into the loop at RETURN. Jumping out with a GOTO command is very bad programming and should never be done.

BUG HUNT

You must have a corresponding NEXT statement for every FOR statement, otherwise a NEXT without FOR error will result.

ASSOCIATED KEYWORDS AND REFERENCES

NEXT
STEP
TO
Section 1 Chapter 5: THE USE OF LOOPS

FRE Amount of memory FREe

DESCRIPTION

This system variable tells you how much memory you have left in the BASIC program area, and string storage area.

SYNTAX

FRE(0) gives the memory left in the BASIC programming area.
FRE("") gives the memory left in the string area.

EXAMPLES

```
PRINT FRE(0)
28815
PRINT FRE("")
200
```

POINTS TO REMEMBER

FRE(0) is equivalent to the memory area indicated by FREE in the memory map.

BUG HUNT

You need the dummy argument (0) and ("") to use the function FRE.

ASSOCIATED KEYWORDS AND REFERENCES

Section 1 Chapter 7: USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS
Section 2 Chapter 20: MEMORY MAP.

GOSUB GO to SUBroutine

DESCRIPTION

In BASIC programming, it is often the case that you need the same routine at a number of different places. You can place a subroutine at a different position from where it is needed, and you can use GOSUB to call the subroutine at the line number specified as often as you like.

When BASIC encounters the GOSUB statement, it immediately jumps to the line number specified and continues the execution from there. When the computer reaches the RETURN statement, it then returns to the original main routine from where it left off.

You can even call a subroutine from a subroutine. In fact you can have nested subroutines as deep as you like, as long as the memory does not run out. It is possible to have a recursive subroutine which calls itself.

SYNTAX

GOSUB <line>

EXAMPLES

An example of recursive GOSUB. The subroutine calls itself at line 110 until the condition is satisfied.

```
10 PRINT "START"
20 A=1
30 GOSUB 100
40 PRINT A
50 END
90 REM SUBROUTINE
100 A=A+1
110 IF A<>100 THEN GOSUB 100
120 RETURN
```

```
RUN
START
 100
```

POINTS TO REMEMBER

It is bad practice to jump a line with a REM statement because there might be a need to delete REM statements to save memory at a later date.

It is not possible to have a variable expression for <line> number, i.e. GOSUB A%*10 is wrong.

It is standard practice in computer programming to make the subroutine readily distinguishable from the main program and preferably noted with a REM statement.

Should the logic dictate, it is perfectly possible to have more than one RETURN for one subroutine.

Make sure that when you are programming, you do not accidentally enter the subroutine; use an END statement before subroutines.

GOSUBS are also used widely with event trapping statements such as ON SPRITE etc. See the relevant section for more details.

BUG HUNT

If the GOSUB <line> points to a non-existent line number, an Undefined Line Number error will result.

ASSOCIATED KEYWORDS AND REFERENCES

RETURN
ON .. GOSUB
ON INTERVAL GOSUB
ON KEY .. GOSUB
ON SPRITE GOSUB
ON STOP GOSUB
ON STRIG GOSUB
Section 1 Chapter 17 STRUCTURING YOUR PROGRAMS

GOTO — GOTO line number

DESCRIPTION

GOTO tells the computer to jump to the line number specified, then continue the execution from that line.

In an IF/THEN/GOTO type statement, you can replace THEN GOTO with just GOTO or THEN. Your computer will understand what you mean.

SYNTAX

GOTO <line>

EXAMPLES

```
10 PRINT "WELCOME TO MSX"
20 GOTO 10
```

```
RUN
WELCOME TO MSX
WELCOME TO MSX
WELCOME TO MSX
WELCOME TO MSX
WELCOME TO MSX
WELCOME TO MSX
Break in 10
```

CONTINUES TO PRINT OUT WELCOME TO MSX UNTIL YOU PRESS CTRL + STOP

POINTS TO REMEMBER

You can GOTO the same line which contains the GOTO. This is used to retest certain conditions, and to lock onto the graphics screens, e.g.:

```
100 IF INKEY$ <> "A" GOTO 100
```

BUG HUNT

If the GOTO <line> points to a non-existent line number, an Undefined Line Number error will result.

ASSOCIATED KEYWORDS AND REFERENCES

ON GOTO
ON ERROR GOTO
Section 1 Chapter 3: WRITING A PROGRAM

HEX$ HEX string

DESCRIPTION

HEX$ gives the hexadecimal value of a decimal integer number in a string form. For example, HEX$(255) gives the string FF.

SYNTAX

HEX$(<integer-numeric>)
It can be used in the range of −32768 to 65535.
For negative integers, twos complement is used, i.e. HEX$(65535−x) = HEX$(−x).

EXAMPLES

```
PRINT HEX$(255)
A$=HEX$(NUM%)
```

POINTS TO REMEMBER

&H<number> gives the decimal value of a hex number.

BUG HUNT

The argument should be an integer, whereas the result is a string.

ASSOCIATED KEYWORDS AND REFERENCES

OCT$
BIN$
Section 2 Chapter 5: SURVEY OF NUMBER SYSTEMS USED IN MSX.

IF

DESCRIPTION

The IF statement tests a condition, or a combination of conditions, and reacts accordingly.

Condition testing statements involving IF have the following formats:

IF condition is true THEN do everything after the word THEN.
IF condition is false THEN do not react to the statements after the words THEN, but carry on from the next line.

ELSE is a part of the IF/THEN/ELSE structure. ELSE tells the computer that if the <condition> is not satisfied (i.e. it is false), then skip the statements after THEN, and execute the statements after ELSE.

Multiple IF/THEN/ELSE statements can be set up as well as nested IF/THEN/ELSE statements. They are only limited by the length of the program line which is 255 characters. If there are less ELSEs than THENs, each ELSE is matched to the nearest THEN.

IF THEN (IF THEN ELSE)

IF THEN (IF THEN (IF THEN ELSE) ELSE) ELSE

The IF ... THEN GOTO <line> format can be abbreviated into either of the following two forms:

1) IF ... THEN <line>
2) IF ... GOTO <line>

ELSE GOTO <line> can be abbreviated to ELSE <line>.

SYNTAX

IF <condition> THEN <statements>
IF <condition> THEN <line>
GOTO
IF <condition> THEN <statement or line> ELSE <statement or line>
GOTO GOTO

EXAMPLES

Line 50 is a simple IF THEN structure
Line 60 is an IF THEN ELSE IF THEN ELSE structure

```
10 PRINT"PLEASE INPUT THREE DIFFERENT
   NUMBERS"
20 INPUT"A";A%
30 INPUT"B";B%
40 INPUT"C";C%
50 IF A%=B% OR A%=C% OR B%=C% THEN 10
60 IF A%<B% AND A%<C% THEN PRINT "A" ELSE
   IF B%<A% AND B%<C% THEN PRINT "B" ELSE
   PRINT "C"
80 PRINT " IS THE SMALLEST NUMBER"
```

```
RUN
PLEASE INPUT THREE DIFFERENT NUMBERS
A? 8
B? 4
C? 3
C
IS THE SMALLEST NUMBER
```

POINTS TO REMEMBER

IF/THEN/ELSE statements tend to get too long to fit on a single line, because you can have multiple statements after THEN and ELSE. If this is the case, then it is a good idea to use subroutines with the GOSUB statement.

It is advisable that you avoid nested IF/THEN/ELSE statements as they can get into a bit of a muddle. You can create a spaghetti program if you are not careful.

BUG HUNT

You tend to find bugs in the <condition> section of IF/THEN/ELSE statements.

ASSOCIATED KEYWORDS AND REFERENCES

THEN
ELSE
AND
OR
NOT
Section 1 Chapter 6: THE USE OF CONDITIONS
Section 2 Chapter 7: BOOLEAN II: THE IF/THEN/ELSE

IMP IMPlication

DESCRIPTION

One of the boolean logical operators.

Boolean algebraic operation can be listed in an IMP truth table:

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

100 IMP 50 can be calculated as follows:

	X	100	0000000001100100
IMP	Y	50	0000000000110010
		−69	1111111110111011

SYNTAX

<num-var>=<num-const> IMP <num-const>
<num-var>=<num-var> IMP <num-exp>
and other numerical combinations.

EXAMPLES

```
10 S=99
20 PRINT S IMP S
30 PRINT (-S IMP S)
40 PRINT (S IMP -S)
```

```
RUN
-1
 99
-99
```

POINTS TO REMEMBER

The following relationships are true:

X IMP X = −1
−X IMP X = X
X IMP −X = −X

IMP is a 16 bit operation

BUG HUNT

An Overflow error results if one of the arguments is out of the integer range.

ASSOCIATED KEYWORDS AND REFERENCES

AND
OR
NOT
XOR
EQV
Section 2 Chapter 6: BOOLEAN ALGEBRA

INKEY$

DESCRIPTION

This function tests the keyboard to see if a key is being pressed, and if so returns the character being pressed. If there is no key being pressed, it will return a null string.

SYNTAX

<string-var> = INKEY$

EXAMPLES

This example program demonstrates how to make the computer wait until a key is pressed. Line 20 tests if a key has been pressed and the character stored in A$. If a key has not been pressed, it tells the computer to go back to the beginning of the same line and try again until one is pressed.

```
10 PRINT "PRESS A KEY TO EXIT THE PROGRAM"
20 A$=INKEY$:IF A$="" THEN 20  ELSE STOP
```

```
RUN
PRESS A KEY TO EXIT THE PROGRAM
Break in 20                          PRESS A KEY TO STOP
```

POINTS TO REMEMBER

There are several differences between INKEY$ and INPUT:

1) INKEY$ will not give a question mark "?" on the screen.
2) INKEY$ does not wait for you to press a key. Either you have a key pressed while the computer executes the INKEY$ statement or you will get null string returned.
3) INKEY$, unlike INPUT, does not display the returned character on the screen.
4) INKEY$ recognises TAB and DEL keys.
5) Unlike INPUT, INKEY$ does not put you into text mode when executed in a graphics mode.
6) INKEY$ returns only one character, whereas INPUT can return a whole line of characters.

BUG HUNT

A Type Mismatch error will occur, if INKEY$ is equated to a numeric variable.

ASSOCIATED KEYWORDS AND REFERENCES

INPUT$
Section 1 Chapter 10: INTERACTIVE PROGRAMMING

INP IN from Port

DESCRIPTION

This returns the byte read in at the specified port. The port number must be in the range of 0 to 255. Anything above 255 is not used by MSX.

Do not use this statement for commercial software. Use BIOS — it is easier.

SYNTAX

<num-var>=INP(<port number>)

EXAMPLES

PRINT INP(1)

POINTS TO REMEMBER

INP is the opposite of the OUT statement.

BUG HUNT

Unless you know exactly what you are doing, try not to use it.

ASSOCIATED KEYWORDS AND REFERENCES

OUT
WAIT

INPUT

DESCRIPTION

INPUT allows you to input numbers and strings into variables while the program is running, so that you can make the program interactive with the user.

The INPUT statement halts the program, and waits for the necessary data to be entered via the keyboard.

It is possible to include a prompt before the question mark which appears on the screen.

Any number of numeric or string variables can be input one after the other in the same INPUT statement, as long as they are separated by commas. However, the type of input and type of variable specified in the INPUT statement must agree. Failure to do so will result in a

?Redo from start

prompt appearing and you will have to re-input from the very first variable asked for.

SYNTAX

INPUT <list of string or numeric variables separated by commas>
INPUT "message";<list of string or numeric variables>

EXAMPLES

```
10 INPUT"YOUR NAME";N$
20 INPUT"YOUR AGE AND RELIGION";A%,R$
30 PRINT N$
40 PRINT A%
50 PRINT R$
```

```
RUN
YOUR NAME? GODZILLA
YOUR AGE AND RELIGION? 12,BUDDHIST
GODZILLA
12
BUDDHIST
```

POINTS TO REMEMBER

In graphics mode, INPUT will force you back into the text mode.

The INPUT statement responds in three ways when the input is not as it should be:

1) A Type mismatch error will cause
   ?Redo from start.
   You will have to re-input from the star.

2) Too many inputs will cause
   ?Extra ignored
   then execution of the program continues.

3) Not enough input will cause
   ??
   two question marks to be displayed, and the computer will wait until the missing variable is inputted.

The only way to escape from INPUT is to <CTRL> <C> or <CTRL><STOP>. It is possible to get back into the program after such an escape, using the CONT command, but you will have to reinput from the start.

If the RETURN key only is pressed the input variable will be unchanged (i.e. it will still contain its previous value).

Each variable must be separated from the next by a comma in the list of variables. The items input must also be separated by commas.

BUG HUNT

Typing mistakes while inputting will be dealt with by the computer as above.

If the length of an inputted string exceeds 200 characters it will cause out of string space error unless you increase the string work space using the CLEAR statement.

ASSOCIATED KEYWORDS AND REFERENCES

INKEY$
INPUT#
INPUT$
LINE INPUT
Section 1 Chapter 3: COMMAND MODE
Section 2 Chapter 10: INTERACTIVE PROGRAMMING

INPUT# INPUT from file

DESCRIPTION

This statement allows you to read in data from devices other than the keyboard, such as a cassette.

Before this statement is executed, a specified channel must be opened using the OPEN statement and an assigned file number.

The input format must be exactly the same as it would be for the keyboard.

During the input for numeric variables, leading spaces, carriage returns and line feeds are ignored. The same goes for string variables Quotation marks are also ignored.

SYNTAX

INPUT #<file number>,<list of string or numeric variables>

EXAMPLES

INPUT #1,A$

POINTS TO REMEMBER

To open a file to the cassette tape recorder, use:
OPEN "CAS:" FOR INPUT AS #1

BUG HUNT

A Bad file number error will result if the file number referred to is not OPENed.

ASSOCIATED KEYWORDS AND REFERENCES

OPEN
CLOSE
INPUT
INPUT$
Section 2 Chapter 19: HOW TO USE FILES

INPUT$

DESCRIPTION

This returns a string of a given number of characters, read from the keyboard. None of the characters will be displayed on the screen. <CTRL> <C> terminates the execution of the INPUT$ function.

SYNTAX

<string-var>=INPUT$(<num-const>)

EXAMPLES

A$=INPUT$(5)

POINTS TO REMEMBER

Unlike the INKEY$ function, INPUT$ waits for the characters to be typed in. Also it can handle more than one character at a time.

ASSOCIATED KEYWORDS AND REFERENCES

INKEY$
INPUT$(#)
INPUT
Section 1 Chapter 10: INTERACTIVE PROGRAMMING

INPUT$(#)

DESCRIPTION

This returns a string of a given number of characters, read from a device other than the keyboard. No character will be displayed on the screen. The file number must be given and a file can be opened using the OPEN statement.

SYNTAX

<string-var>=INPUT$(<num-const>,<file number>)
<string-var>=INPUT$(<num-const>,#<file number>)

EXAMPLES

W$=INPUT$(5,#1)

POINTS TO REMEMBER

To open a file to the cassette tape recorder, use:
OPEN "CAS:" FOR INPUT AS #1

BUG HUNT

A Bad file number error will result if the file number referred to is not OPENed.

ASSOCIATED KEYWORDS AND REFERENCES

OPEN
CLOSE
INPUT
INPUT#
INPUT$
Section 2 Chapter 19. HOW TO USE FILES

INSTR IN STRing

DESCRIPTION

INSTR searches for a specified string within a string, and then returns the position of this specified string if found.

INSTR(A$,B$) . . . A$ is the main string, and B$ is the string to be searched for.

If the string is not found INSTR returns 0.

The search starts from the begining of the string, but you can specify the start position of the search. The position number must lie between 1 and 255 inclusive.

SYNTAX

<numeric>=INSTR(<string-var>,"<string>")
<numeric>=INSTR(<string-var>,<string-var>)
<numeric>=INSTR(<numeric>,<string-var>,<string-var>)

EXAMPLES

```
10 PRINT "MAIN STRING IS AS FOLLOWS:"
20 PRINT "TRAPPED IN A TWISTY LITTLE
   PASSAGE, ALL ALIKE."
30 A$="TRAPPED IN A TWISTY LITTLE
   PASSAGE, ALL ALIKE."
40 INPUT "INPUT A CHARACTER TO BE
   SEARCHED";B$
50 C%=0
60 C%=INSTR(C%+1,A$,B$)
70 IF C%=0 THEN END
80 PRINT "CHARACTER ";C%;" IS ";B$
90 GOTO 60
```

```
RUN
MAIN STRING IS AS FOLLOWS:
TRAPPED IN A TWISTY LITTLE PASSAGE,
ALL ALIKE
INPUT A CHARACTER TO BE SEARCHED? T
CHARACTER  1 IS T
CHARACTER  14 IS T
CHARACTER  18 IS T
CHARACTER  23 IS T
CHARACTER  24 IS T
```

POINTS TO REMEMBER

For INSTR(Z,X$,Y$), you get 0 returned in the following cases:

1) If Y$ cannot be found in X$
2) If Z is larger than the length of X$
3) If X$ is a null string i.e. X$=""
4) If X$ and Y$ are null strings.

If you try to search for a null string, "", then you will get 1 from INSTR.

BUG HUNT

If the search position number is 0 or more than 255, an illegal function call error will result.

ASSOCIATED KEYWORDS AND REFERENCES

LEFT$
RIGHT$
MID$
LEN
Section 1 Chapter 14: MANIPULATING STRINGS

INT INTeger

DESCRIPTION

This returns the integer part of a real number. It rounds the real number down to the nearest smaller whole number for both positive and negative numbers.

SYNTAX

INT(<num-const>)
INT(<num-var>)
INT(<num-exp>)

EXAMPLES

```
PRINT INT(2.76)
2
PRINT INT(10.1111)
10
PRINT INT(-1.234)
-2
```

POINTS TO REMEMBER

Note that INT of a negative number returns the nearest smaller whole number than the number supplied. Therefore, you get INT(−0.2) returning −1.

INT is different from FIX. FIX gives the nearest lower integer for a positive argument, and the nearest higher integer for a negative argument.

BUG HUNT

A type mismatch error will occur if INT is used with strings.

ASSOCIATED KEYWORDS AND REFERENCES

FIX
Section 1 Chapter 4: MORE ON ARITHMETIC
Section 1 Chapter 15: FUNCTIONS

INTERVAL ON/OFF/STOP

DESCRIPTION

MSX BASIC can time itself, and at specified time intervals you can make it perform a certain routine using the ON INTERVAL statement (see ON INTERVAL).

INTERVAL ON/STOP/OFF statements enable and disable the ON INTERVAL event trapping.

To use ON INTERVAL GOSUB, you must activate it with INTERVAL ON. After INTERVAL ON, the computer checks each time it executes a new line to see if it is the right time to call the ON INTERVAL GOSUB routine.

INTERVAL OFF tells the computer to forget about checking the elapsed time interval trapping, and disables the ON INTERVAL GOSUB.

INTERVAL STOP tells the computer not to perform the ON INTERVAL GOSUB but to remember to do so when INTERVAL ON is executed if the time interval has elapsed while it is in STOP mode.

SYNTAX

INTERVAL ON
INTERVAL OFF
INTERVAL STOP

EXAMPLES

Press <y> to start the interval trapping. Press <n> to stop the interval trapping. You should hear beeps when the interval interrupt is enabled.

```
10 ON INTERVAL=60 GOSUB 50
20 IF INKEY$="y" THEN INTERVAL ON
30 IF INKEY$="n" THEN INTERVAL OFF
40 GOTO 20
45 REM INTERVAL SUBROUTINE
50 BEEP
60 RETURN
```

LINE 10 SETS THE INTERVAL TO 1 SEC. SUBROUTINE AT LINE 50
LINE 20 ENABLES INTERRUPT
LINE 30 DISABLES INTERRUPT
LINE 40 INFINITE LOOP BACK TO LINE 20
LINE 60 RETURNS TO WHEREVER IT LEFT THE MAIN ROUTINE

POINTS TO REMEMBER

The time interval is set in units of 1/60th of a second.

Timed interrupts are disabled in the command mode, i.e. when the program is not running and during error trapping routines.

ASSOCIATED KEYWORDS AND REFERENCES

ON INTERVAL GOSUB

Section 2 Chapter 9: EVENT HANDLING AND INTERRUPT BY BASIC

KEY

DESCRIPTION

This command is used to define the action of a function key. All MSX computers have five function keys, with each key representing two functions — one accessed by pressing the function key, and the other by pressing <SHIFT> and the function key at the same time.

When you switch on the computer, the function keys are already initialised to the following:

F1 = color [s]	colour
F2 = auto [s]	auto
F3 = goto [s]	goto
F4 = list [s]	list
F5 = run [r]	runs the program
F6 = colour 15,4,7[r]	sets the color to the default colour
F7 = cload"	loads from cassette
F8 = cont [r]	continues the program
F9 = list. [r][u][u]	lists the last line referred to and moves the cursor to the begining of that line
F10 = [cls] run [r]	clears the screen then runs the program

[s] = space
[r] = RETURN
[cls] = clear screen
[u] = cursor up one line

You can redefine any of the function keys using the KEY statement. The string, however, must be less than 15 characters long.

If you want to define a function so that it is executed immediately after pressing the function key, then add a carriage return character CHR$(13) to the string. You can also include the CLS command and other control character codes to the function string.

SYNTAX

KEY <num-const>,<string>
KEY <num-const>,<string-exp>
where the <num-const> must be between 1 and 10.

EXAMPLES

KEY 2, "PRINT FRE(0)" + CHR$(13) . . .	prints the number of bytes free for BASIC.
KEY 4, "RENUM" + CHR$(13)	renumber
a$ = "KEY LIST"	
KEY 5,a$+CHR$(13)	returns "KEY LIST" [R].

Note: CHR$(13) is the control character for the <RETURN> key.

POINTS TO REMEMBER

When the machine is switched on, the contents of the function keys are displayed on the bottom line of the screen. If you want to get rid of this display, then use the KEY OFF statement.

Remember that the control characters can be incorporated into the function key, so you can make the function keys clear the screen, or beep etc.

BUG HUNT

Do not forget the quotation marks around the string.

ASSOCIATED KEYWORDS AND REFERENCES

KEY ON/OFF
KEY LIST
Section 1 Chapter 8: THE FUNCTION KEYS

KEY LIST

DESCRIPTION

This statement lists the contents of all the function keys without actually executing them.

All the control characters are converted to spaces when listed.

SYNTAX

KEY LIST

EXAMPLES

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

POINTS TO REMEMBER

See KEY statement for how to change the contents of the function keys

ASSOCIATED KEYWORDS AND REFERENCES

KEY
KEY ON/OFF
Section 1 Chapter 8: THE FUNCTION KEYS

KEY ON/OFF

DESCRIPTION

When the computer is switched on, the 24th line on the screen displays the contents of the function keys. This is intended to remind you of what each function key does, but sometimes when you are running a program the key list gets in the way. The KEY ON/OFF will allow you to switch on and off the 24th line function key list.

SYNTAX

KEY ON
KEY OFF

EXAMPLES

KEY ON
KEY OFF

POINTS TO REMEMBER

KEY LIST will give you a list of the contents of the function keys.

ASSOCIATED KEYWORDS AND REFERENCES

KEY
KEY LIST
Section 1 Chapter 8: THE FUNCTION KEYS

KEY () ON/OFF/STOP

DESCRIPTION

The KEY () ON/OFF/STOP statement enables/disables trapping of a specified function key. Once the function key trapping is enabled, the computer will jump to the subroutine specified by the ON KEY GOSUB statement when that function key is pressed.

Individual function keys can be enabled or disabled separately. Function keys which are not specified in KEY () ON are assumed to be disabled.

If KEY () STOP is executed, then the computer will not jump to the subroutine specified when the specified function key is pressed, but the computer will immediately jump to the subroutine as soon as KEY () ON is executed for that function key.

KEY () OFF will completely stop the interrupt for that function key.

SYNTAX

KEY (<num-const>) ON
KEY (<num-const>) OFF
KEY (<num-const>) STOP

EXAMPLES

If you press the function key F0 the computer will beep twice; if you press F1 it will beep only once.

```
10 ON KEY GOSUB 50,60
20 KEY (0) ON
30 KEY (1) ON
40 GOTO 40
50 BEEP
60 BEEP
70 RETURN
```

LINE 10 SETS SUBROUTINE FOR F0 AT LINE 50 AND F1 AT LINE 60
LINE 20 ENABLES FUNCTION KEY F0
LINE 30 ENABLES FUNCTION KEY F1
LINE 40 INFINITE LOOP TO WAIT FOR A FUNCTION KEY
LINE 50 BEEP (FOR F0)
LINE 60 BEEP (FOR BOTH)
LINE 70 RETURNS TO WHEREVER IT CAME FROM IN THIS CASE LINE 40

Note that the above program does nothing when any other function key is pressed.

POINTS TO REMEMBER

The function key interrupt is disabled during the error trapping routine.

BUG HUNT

Do not forget to execute the ON KEY GOSUB statement first.
This statement is totally different to KEY ON/OFF.

ASSOCIATED KEYWORDS AND REFERENCES

ON KEY GOSUB
Section 2 Chapter 9: EVENT HANDLING AND INTERRUPT BY BASIC

LEFT$

DESCRIPTION

LEFT$ is a string manipulation function, which returns the left n characters of a given string. If the source string is shorter than the given value, then the entire string is returned.

SYNTAX

LEFT$(<string-var>,<num-const>)
LEFT$(<string-var>,<num-var>)
LEFT$(<string-var>,<num-exp>)
The number must be in the range of 0 to 255.

EXAMPLES

```
10 A$="JOHN SMITH"
20 P=INSTR(A$," ")
30 B$=LEFT$(A$,P-1)
40 PRINT B$
```

LINE 20 FINDS THE POSITION OF THE SPACE IN A$
LINE 30 GETS THE LEFT P-1 CHARACTERS FROM A$ FOR THE FORENAME

```
RUN
JOHN
Ok
```

POINTS TO REMEMBER

If the number of the argument is 0, then a null string is returned.

To get the surname SMITH in the above program, you have to use MID$ as follows:

```
PRINT MID$(A$,P+1)
```

BUG HUNT

The number in the argument must be an integer between 0 and 255 otherwise a type mismatch error will result.

An illegal function call error occurs when a non-existent variable is referred to.

ASSOCIATED KEYWORDS AND REFERENCES

INSTR
RIGHT$
MID$
LEN
Section 1 Chapter 14: MANIPULATING STRINGS

LEN LENgth of a string

DESCRIPTION

LEN returns the length of a given string.
Control characters and blanks are counted.

SYNTAX

LEN(<string>)

EXAMPLES

```
10 A$="HELLO"+CHR$(7)
20 B=LEN(A$)
30 PRINT A$
40 PRINT "THE LENGTH OF A$ IS ";B
```

LINE 10 CHR$(7) IS A BEEP
LINE 30 PRINTS HELLO WITH A BEEP

```
RUN
HELLO                                    YOU HEAR A BEEP
THE LENGTH OF A$ IS 6
```

POINTS TO REMEMBER

This function always returns an integer.

BUG HUNT

This returns zero when a non-existent string variable is referred to.

ASSOCIATED KEYWORDS AND REFERENCES

RIGHT$
LEFT$
MID$
INSTR
Section 1 Chapter 15: FUNCTIONS

LET

DESCRIPTION

When you assign some value to a string or numeric variable, you use the LET statement as in:

```
LET A=100
```

LET is optional, and it is better not to use it because it is unnecessary and wastes memory space, i.e. A=100 is equivalent to the above.

SYNTAX

LET <variable>=<expression>

EXAMPLES

```
LET A=100
LET W$="WEEK"+"END"
```

ASSOCIATED KEYWORDS AND REFERENCES

Section 1 Chapter 2: COMMAND MODE

LINE draws a LINE

DESCRIPTION

This graphics statement draws lines and rectangles. The rectangles may be filled in with any colour.

SYNTAX

LINE-<coordinate specifier>
draws a line in current foreground colour from the last point referred to by the computer, to the point given by the coodinates specifier.

LINE-<coodinate specifier>,<colour>
draws a line in the colour specified from the last point referred to by the computer, to the point given by the coordinates specifier.

LINE <coordinate specifier>-<coordinate specifier>
draws a line from the first coordinate to the second in the current foreground colour.

LINE <coordinate specifier>-<coordinate specifier>,<colour>
draws a line from the first coordinate to the second, in the colour specified.

LINE <coordinate specifier>-<coordinate specifier>,<colour>,B
draws a rectangle in the colour specified with the top left corner of the rectangle given by the first coordinates specifier and the bottom right corner given by the second coordinate specifier. (B stands for box.)

LINE <coordinate specifier>-<coordinate specifier>,<colour>,BF
draws a rectangle in the colour specified with the top left corner of the rectangle given by the first coordinates specifier and the bottom right corner given by the second coordinate specifier. Then, the rectangle is filled with the colour specified. (BF stands for box filled.)

Explanation of <coordinates specifier>

There are two formats, one relative and the other an absolute coordinate format:

1) (<x-coordinate>,<y-coordinate>)
   These specify an absolute position on the screen. X can be between 0 and 255 and y between 0 and 191.
2) STEP (<x-offset>,<y-offset>)
   These coordinates are of a point relative to the last point referred to. If the resultant offset is outside the screen, the computer will draw up to the edges of the screen.

   <x-offset> and <y-offset> may be negative depending in which direction you wish to draw.

NOTES: <x-offset>, <y-offset>, <x-coordinates>, <y-coordinates> can be variables, expressions or just simple numerical constants.

When you require two coordinate specifiers, it is perfectly acceptable to mix the relative and the absolute format, as long as they are joined with a "-" (negative) sign.

EXAMPLES

```
10 SCREEN 2
20 COLOR 4,1,1
30 CLS
40 LINE (50,50)-(100,100)
50 LINE (50,50)-(100,100),4,B
60 LINE STEP(-50,20)-STEP(50,50),9,BF
70 X=10:Y=10
80 LINE (15*X,10*Y)-STEP(50,20),6,B
90 GOTO 90
```

```
  YOU SEE A BOX WITH A DIAGONAL LINE
A PURPLE SQUARE AND A RECTANGLE
```

POINTS TO REMEMBER

Real number coordinates are truncated to integers. The colour code must be a valid number between 0 and 15 (see COLOUR).

BUG HUNT

The coordinates can be bigger than or smaller than the screen range, but if they are outside the integer range (−32768 to 32767) then an Overflow error will result (e.g. LINE (−100000,100000) will give Overflow.)

ASSOCIATED KEYWORDS AND REFERENCES

PAINT
COLOR
DRAW
Section 1 Chapter 24: DRAWING LINES AND BOXES
Section 2 Chapter 13: ADVANCED GRAPHICS II: COLOUR IN HIGH RESOLUTION GRAPHICS MODE 2

LINE INPUT

DESCRIPTION

This statement is similar to INPUT but allows you to input an entire sentence up to 200 characters, including commas.

You can have prompts, as in the INPUT statement, but a question mark will not be printed automatically, unlike the INPUT statement.

SYNTAX

LINE INPUT <string-var>
LINE INPUT "<prompt>";<string-var>

EXAMPLES

Adventure game subroutine

```
10 LINE INPUT "What shall I do now?";S$
20 PRINT "OK YOU ";S$
```

```
RUN
What shall I do now? GET LAMP, SWORD,
AND WATER
OK YOU GET LAMP, SWORD, AND WATER
```

POINTS TO REMEMBER

You can escape from LINE INPUT by <CTRL> <C> or <CTRL> <STOP>. CONT will let you continue from the LINE INPUT.

You cannot use this statement in a graphics mode. You must return to the text mode first.

The maximum number of characters in a line is 200. This limit is set by the size of the string work area of the system's RAM. If you want to extend the string work space, use the CLEAR statement.

BUG HUNT

You can only use this with a string variable. If you try to LINE INPUT a numeric variable, you will get a type mismatch error.

An out of string space error will occur if an inputted string is too long to fit in the string work area (maximum at default is 200 characters).

ASSOCIATED KEYWORDS AND REFERENCES

LINE INPUT #
INPUT
Section 1 Chapter 10: INTERACTIVE PROGRAMMING

LINE INPUT#

DESCRIPTION

This statement is similar to INPUT#, but this allows you to read an entire sentence up to 200 characters from a sequential file on an external device such as a cassette.

It basically means that you can read spaces and commas, as well as characters, from a cassette all into one string.

SYNTAX

LINE INPUT # <file number>,<string-var>

<file number> is the number of the file OPENed.

EXAMPLES

LINE INPUT#1,A$

POINTS TO REMEMBER

LINE INPUT# reads everything up to the carriage return.

It is useful when each line of a file has been broken into fields.

Also, a BASIC program saved in ASCII codes can be read in, line by line, within a program using LINE INPUT#.

BUG HUNT

You can only use this with a string variable. If you try to LINE INPUT a numeric variable, you will get a type mismatch error.

An out of string space error will occur if an inputted string is too long to fit in the string work area (maximum at default is 200 characters).

ASSOCIATED KEYWORDS AND REFERENCES

INPUT
INPUT#
INPUT$(#)
OPEN
LINE INPUT
Section 2 Chapter 19: HOW TO USE FILE

LIST

DESCRIPTION

This command lists the BASIC program held within the computer's memory. You can list the whole or part of a program.

SYNTAX

LIST	lists the entire program
LIST <line>	lists the line specified
LIST <line 1>-<line 2>	lists the program from line 1 to line 2
LIST <line>-	lists from the line specified to the end of the program
LIST -<line>	lists from the begining to the line specified
LIST.	lists the last line referred to by the computer.

EXAMPLES

LIST 100–200	list from line 100 to 200 inclusive
LIST–100	list up to and including line 100

POINTS TO REMEMBER

The function key F4 is defined as "LIST" when the computer is switched on.

All lower case characters, apart from those within quotation marks, will be converted to upper case characters upon listing the program.

ASSOCIATED KEYWORDS AND REFERENCES

LLIST

Section 1 Chapter 7: USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

Section 2 Chapter 1: ADVANCED PROGRAM EDITING

LLIST

DESCRIPTION

This command lists the BASIC program held within the computer's memory to the printer. You can list the whole or part of a program.

SYNTAX

LLIST	prints the entire program
LLIST <line>	prints the line specified
LLIST <line 1>-<line 2>	prints the programs from line 1 to line 2
LLIST <line>-	prints from the line specified to the end of the program
LLIST-<line>	prints from the begining to the line specified
LLIST.	prints the last line referred to by the computer.

EXAMPLES

LLIST 100–200	prints from line 100 to 200 inclusive
LLIST –100	prints up to and including line 100

POINTS TO REMEMBER

All lower case characters, apart from those within quotation marks, will be converted to upper case characters upon listing the program.

BUG HUNT

If the printer is not connected, nothing happens.

ASSOCIATED KEYWORDS AND REFERENCES

LIST
Section 2 Chapter 23: PERIPHERAL DEVICES

LOAD

DESCRIPTION

This loads in a BASIC file, saved with SAVE in ASCII format, from a specified device.

For the moment, only the cassette option (CAS:) is implemented.

LOAD has an auto run option, R, which will leave the file open and run the program when loaded.

SYNTAX

LOAD "<device-name>"	loads the first BASIC file found, saved in ASCII format.
LOAD "<device-name><file-name>"	loads a BASIC file with the specified name.
LOAD "<device-name><file-name>",R	loads a BASIC file with the specified name, then runs it

<device-name>=CAS: at present

EXAMPLES

LOAD "CAS:GAME",R
Automatically loads and runs the program GAME, saved in ASCII code.

POINTS TO REMEMBER

When LOAD is executed, the computer will close all files left open and start loading in a new BASIC program which will erase the previous program in the memory.

<CTRL> <Z> is treated as EOF, i.e. end-of-file.

BUG HUNT

Do not load in machine code bytes using LOAD; use BLOAD instead.

ASSOCIATED KEYWORDS AND REFERENCES

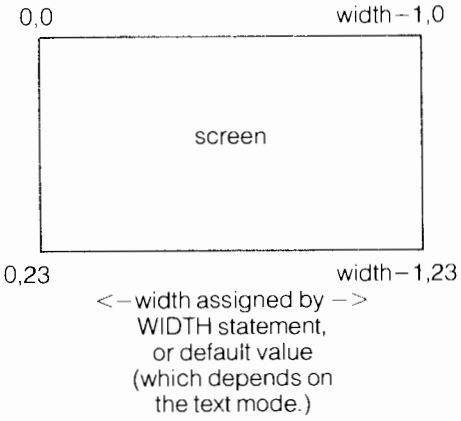
SAVE
CLOAD
CSAVE
BLOAD
BSAVE
Section 2 Chapter 2: CASSETTE SAVING AND LOADING

LOCATE

DESCRIPTION

LOCATE moves the cursor to the specified position. It also has the ability to turn the cursor display on and off. This statement is usually used in conjunction with PRINT statements to print at a specific part of the text screen.

You can move the cursor using LOCATE to any point within the limit imposed by the text mode and the WIDTH statement.



Default width for both text modes is as follows:
MODE 0: WIDTH=37 (maximum width = 40)
MODE 1: WIDTH=29 (maximum width = 32)

SYNTAX

LOCATE <x-coordinate>,<y-coordinate>	moves cursor to specified coordinates
LOCATE <x-coordinate>,<y-coordinate>,0	moves cursor to specified coordinates but disables the cursor.
LOCATE <x-coordinate>,<y-coordinate>,1	moves cursor to specified coordinates and switches the cursor on.

<x-coordinate> and <y-coordinate> can be either variables, constants or expressions but must be within the screen limits. All real numbers are truncated to integers.

EXAMPLES

```
10 FOR Y=0 TO 6
20 FOR X=0 TO 20
30 LOCATE X,Y
40 PRINT "E"
50 NEXT:NEXT
60 PRINT "LOTS OF Es"
```

```
EEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEE
LOTS OF Es
```

POINTS TO REMEMBER

LOCATE will not work at all in the graphics modes.

LOCATE can be used for locating the position of an INPUT prompt.

There is a similar function, TAB, which is used in the form PRINTTAB but this is very restricted. It only allows you to move in the x direction, hence LOCATE is often more useful.

BUG HUNT

Make sure that the coordinates are not out of the limits; an illegal function call error results if they are.

ASSOCIATED KEYWORDS AND REFERENCES

PRINT
INPUT
TAB
Section 1 Chapter 9: MORE ON PRINT AND THE SCREEN

LOG

DESCRIPTION

LOG returns the natural log of the number given in double precision.

SYNTAX

LOG(<num-const>)
LOG(<num-var>)
LOG(<num-exp>)

EXAMPLES

```
PRINT LOG(10)
  2.302585092994
```

POINTS TO REMEMBER

This is the natural log to the base e, which is different from log to the base 10.

The argument must be greater than 0.

BUG HUNT

An illegal function call error will result if the argument is negative.

ASSOCIATED KEYWORDS AND REFERENCES

EXP
Section 1 Chapter 19: MATHEMATICAL FUNCTIONS.

LPOS Line POSition

DESCRIPTION

LPOS returns the position of the printer head as held within the printer buffer.

LPOS requires a dummy variable X.

LPOS does not necessarily represent the physical position of the printer head.

SYNTAX

LPOS(X)

EXAMPLES

PRINT LPOS(X)
or
A=LPOS(X)

POINTS TO REMEMBER

You cannot assign a value to this system variable.

ASSOCIATED KEYWORDS AND REFERENCES

Section 2 Chapter 23: PERIPHERAL DEVICES

LPRINT

DESCRIPTION

LPRINT is the "PRINT" command for printing out to the printer, and is basically the same as PRINT.

SYNTAX

LPRINT

EXAMPLES

LPRINT "this is a test";123
will print "this is a test 123" to the printer.

POINTS TO REMEMBER

Remember to have a printer connected!

ASSOCIATED KEYWORDS AND REFERENCES

PRINT
LPRINT USING
Section 2 Chapter 23: PERIPHERAL DEVICES

LPRINT USING

DESCRIPTION

LPRINT USING is the same as PRINT USING, except that it prints out to a printer. See PRINT USING as they are identical in function.

SYNTAX

See PRINT USING

EXAMPLES

LPRINT USING "I LOVE MY LITTLE @ VERY MUCH";"MSX"
will print
I LOVE MY LITTLE MSX VERY MUCH

POINTS TO REMEMBER

Do not forget to connect the printer when you are using this statement.

BUG HUNT

See PRINT USING

ASSOCIATED KEYWORDS AND REFERENCES

LPRINT
PRINT USING
Section 2 Chapter 8: PRINT USING
Section 2 Chapter 23: PERIPHERAL DEVICES

MAXFILES — MAXimum number of FILES

DESCRIPTION

This sets the maximum number of files at any one time. The range is between 0 and 15. When MAXFILES is not set, then the computer assumes MAXFILES = 1. If MAXFILES is 0, then only SAVE and LOAD can be performed.

SYNTAX

MAXFILES = <num-const>
MAXFILES = <num-var>
MAXFILES = <num-exp>

Range 0 to 15

EXAMPLES

1000 MAXFILES = 5

POINTS TO REMEMBER

MAXFILES has an effect of increasing the size of the file control block in the memory. The file control block is used as a work area when you are accessing peripheral devices using files.

BUG HUNT

The most common error associated with this function is to miss out the S in MAXFILES.

An illegal function call error will result if the argument is out of range.

ASSOCIATED KEYWORDS AND REFERENCES

OPEN
CLOSE
Section 2 Chapter 19: HOW TO USE FILES

MERGE MERGE two programs

DESCRIPTION

This statement allows you to MERGE two BASIC programs into one.

Let us say that you have two BASIC programs, program 1 and program 2, and you want to merge program 2 with program 1, such that program 2 comes after program 1 when it is eventually merged. Let us say that both are initially saved on a cassette tape.

First, load program 2 and renumber it so that all line numbers are greater than those in program 1. Then save it in an ASCII file using SAVE.

Load program 1 from the cassette, then type in MERGE"program2" and play the cassette. The computer will add program 2 to the end of program 1.

If any line numbers coincide in the two programs, then they are replaced by those merged into it.

SYNTAX

MERGE"<device-name>"
MERGE"<device-name><file-name>"
<device-name>=CAS: for cassette.

EXAMPLES

How to merge two programs, 1 and 2.

PROGRAM 1

```
10 PRINT"HELLO"
20 PRINT"WELCOME"
30 PRINT"TO"
```

PROGRAM 2

```
10 FOR I = 1 TO 10
20 PRINT"MSX"
30 NEXT I
```

CLOAD IN PROGRAM 2 AND THEN RENUM 50 SO IF YOU LIST IT YOU WILL GET THIS

```
50 FOR I=1 TO 10
60 PRINT"MSX"
70 NEXT I
```

Save program 2 in ASCII format by using SAVE"CAS:PROG2".

Now load in program 1 using CLOAD.

MERGE"CAS:PROG2" will merge program 2 on the end of program 1, so that if you LIST the merged program you get the following:

```
10 PRINT"HELLO"
20 PRINT"WELCOME"
30 PRINT"TO"
50 FOR I =1 TO 10
60 PRINT"MSX"
70 NEXT I
```

POINTS TO REMEMBER

If a line number occurs in both the initial program (1) and the second program (2), the resulting MERGEd program will contain the line from the second program (2).

If the file name is omitted, then the next ASCII file encountered on the tape will be merged.

<CTRL>-Z tells the computer the end-of-file, EOF.

BUG HUNT

Get the name of program to be MERGEd right or you will miss it.

ASSOCIATED KEYWORDS AND REFERENCES

SAVE
Section 2 Chapter 11: CASSETTE SAVING AND LOADING

MID$

DESCRIPTION

This function returns the middle part of a given string.

SYNTAX

MID$(<string-var>,x,y)	returns the mid part of the string, length y characters, from position x.
MID$(<string-var>,x)	returns the remainder of the string from position x.

EXAMPLES

MID$ is often used in separating words within a sentence. Here is a good example.

```
10 A$="MARTIN LUTHER KING"
20 B1=INSTR(A$," ")
30 B2=INSTR(B1+1,A$," ")
40 PRINT LEFT$(A$,B1-1)
50 PRINT MID$(A$,B1+1,B2-B1-1)
60 PRINT MID$(A$,B2+1)
```

```
RUN
MARTIN
LUTHER
KING
```

POINTS TO REMEMBER

X and y must be in the range 1 to 255.

If x is greater than the length of the given string, then MID$ will return a null string "".

BUG HUNT

This is a string function. Try to avoid Type mismatch errors.

ASSOCIATED KEYWORDS AND REFERENCES

RIGHT$
LEFT$
LEN
INSTR
Section 1 Chapter 14: MANIPULATING STRINGS

MOD MODulus

DESCRIPTION

This carries out modulus arithmetic operations, that is, the operator MOD gives the remainder after integer division.

```
10 MOD 3 = 1
```

because

10/3 is 3 and the remainder is 1.

SYNTAX

<num-var> = <numeric> MOD <numeric>

EXAMPLES

```
10 PRINT 15 MOD 5
```

you get 0

POINTS TO REMEMBER

The operands are truncated to INTEGERs before the operation.

BUG HUNT

A Type mismatch occurs if used with a string variable.

An Overflow error occurs when the argument is out of the integer range (−32768 to 32767).

ASSOCIATED KEYWORDS AND REFERENCES

Y (div)
Section 2 Chapter 4: EXPRESSION AND OPERATORS

MOTOR

DESCRIPTION

This statement allows you to use the cassette tape recorder with remote control, by means of the remote control socket on the computer. This must be connected up to the remote socket on the cassette tape recorder.

MOTOR refers to the motor in the cassette tape recorder.

SYNTAX

MOTOR	flips the motor switch so ON if OFF, OFF if ON.
MOTOR ON	switches the motor ON.
MOTOR OFF	switches the motor OFF.

POINTS TO REMEMBER

This is useful only if you have a cassette player with a REMOTE socket.

ASSOCIATED KEYWORDS AND REFERENCES

See section on cassette.

NEW NEW program

DESCRIPTION

NEW deletes the entire program held in the computer's memory and resets all variables and stack pointers etc. You cannot recover the old program once you NEW the computer.

Its purpose is to make the computer ready for a new program.

SYNTAX

NEW

EXAMPLES

NEW

POINTS TO REMEMBER

NEW does not clear the screen.

Always NEW the computer before loading in another program. If you do not, the old program will mix with the one being loaded.

BUG HUNT

Fatal, if you NEW by mistake!

ASSOCIATED KEYWORDS AND REFERENCES

Section 1 Chapter 3: WRITING A PROGRAM

NEXT

DESCRIPTION

NEXT pairs with FOR, and it tells the computer to go back to the corresponding FOR statement, unless the FOR-NEXT loop is at the final iteration.

For more information on FOR/NEXT loop, see FOR.

SYNTAX

NEXT
NEXT [variable]
NEXT [variable][,variable]..., list of variables.

EXAMPLES

```
10 FOR I=1 to 9
20 PRINT I;
30 NEXT I
```

```
RUN
 1  2  3  4  5  6  7  8  9
Ok.
```

POINTS TO REMEMBER

[variable] after NEXT can be omitted.

BUG HUNT

The following is a common mistake — NEXT I and NEXT J are the wrong way round.

```
10 FOR I=1 TO 5
20 FOR J=1 TO 10
30 PRINT I,J
40 NEXT I
50 NEXT J
```

LINE 40 NEXT WITHOUT FOR ERROR

ASSOCIATED KEYWORDS AND REFERENCES

FOR
TO
STEP
Section 1 Chapter 5: THE USE OF LOOPS

NOT

DESCRIPTION

This is used in IF/THEN condition testing, and in Boolean operations. It returns the opposite, i.e. NOT (TRUE) gives (FALSE).

Truth table for NOT.

NOT	X	NOT X
	0	1
	1	0

SYNTAX

<num-var>=NOT(<numeric>)
IF NOT(<condition>) THEN

EXAMPLES

A=0
PRINT NOT(A)
will give −1. Why? Well A=&B00000000000000000 in binary. NOT(A) becomes &B1111111111111111 which is −1 in decimal.

IF NOT(A=100 AND B<900) THEN GOTO 10000

POINTS TO REMEMBER

Arguments must be within the integer range of −32768 to 32767.
Any fractional part of a real number will be truncated.

ASSOCIATED KEYWORDS AND REFERENCES

IF
THEN
ELSE
AND
OR
XOR
IMP
EQV
Section 2 Chapter 6: BOOLEAN ALGEBRA
Section 2 Chapter 7: BOOLEAN II: THE IF/THEN/ELSE

OCT$ OCTAL STRING

DESCRIPTION

OCT$ gives a string which represents the octal value of the argument in decimal.

Octal is the number system which uses eight as a base or radix. The octal system uses the digits 0, 1, 2, 3, 4, 5, 6, 7, and each digit position represents a power of eight.

The argument must be a numeric expression in the range – 32768 to 65535.

SYNTAX

<string-var>=OCT$(<numeric>)

EXAMPLES

```
10 PRINT OCT$(8)
20 PRINT OCT$(&HF)
30 PRINT OCT$(8^4)
40 PRINT OCT$(8^4-1)
```

```
RUN
10
17
10000
7777
```

POINTS TO REMEMBER

If the argument is negative, then two's complement form is used, i.e. OCT$(–x)=OCT$(65536–x)

BUG HUNT

An Overflow error results if the argument is out of range.

ASSOCIATED KEYWORDS AND REFERENCES

BIN$
HEX$
Section 2 Chapter 5: SURVEY OF NUMBER SYSTEMS USED IN MSX

ON <expression> GOSUB

DESCRIPTION

The ON <expression> GOSUB <line1>, <line2>, <line3>,... statement offers a multiple choice of subroutines. The option depends on the <expression> evaluated: if the <expression> returns 1, the program will GOSUB to the subroutine at <line1>; if it returns 2, <line2> and so on.

The expression must evaluate to between one and a number which is less than, or equal, to the number of line numbers listed.

For example:

ON X GOSUB 100,250,670,800
If X is 1 then it GOSUBs to the subroutine at line 100.
If X is 2 then it GOSUBs to the subroutine at line 250.
If X is 3 then it GOSUBs to the subroutine at line 670.
If X is 4 then it GOSUBs to the subroutine at line 800.

If X is zero or more than the number of the line number listed, i.e. more than 4 in the above example, then the BASIC continues to the next statement.

SYNTAX

ON <expression> GOSUB <list of line numbers>

EXAMPLES

```
10 INPUT "1, 2 OR 3";I
20 ON I GOSUB 40,60,80
30 GOTO 10
40 PRINT "SUBROUTINE 1"
50 RETURN
60 PRINT "SUBROUTINE 2"
70 RETURN
80 PRINT "SUBROUTINE 3"
90 RETURN
```

```
RUN
1, 2 OR 3?2                          INPUT 2
SUBROUTINE 2
1, 2 OR 3?2                          INPUT 1
SUBROUTINE 2
1, 2 OR 3?2                          INPUT 7
1, 2 OR 3?          CONTINUES TO LINE 30 (GOTO 10)
```

POINTS TO REMEMBER

If the evaluated expression is a real number, then the fractional part is truncated, i.e. if X=1.22 then X=1 and the first GOSUB is executed.

IF

<expression>= 0
OR
<expression>> number of items in the list AND less than 255
THEN
BASIC continues to the next statement.

BUG HUNT

If the expression gives a negative number or more than 255, then an illegal function call error will result.

ASSOCIATED KEYWORDS AND REFERENCES

ON ... GOTO
Section 1 Chapter 18: PROGRAM BRANCHING

ON <expression> GOTO

DESCRIPTION

ON <expression> GOTO <line1>,<line2>,<line3>,..... statement offers a multiple choice of GOTO jumps. The option depends on the <expression> evaluated: if the <expression> returns 1, the program will GOTO to <line1>; if it returns 2, <line2> and so on.

The expression must evaluate to between one and a number which is less than, or equal, to the number of line numbers listed.

For example:

ON X GOTO 100, 250, 670, 800
If X is 1 then it goes to the line number 100.
If X is 2 then it goes to the line number 250.
If X is 3 then it goes to the line number 670.
If X is 4 then it goes to the line number 800.

If X is zero or more than the number of the line number listed, i.e. more than 4 in the above example, then the BASIC continues to the next statement.

SYNTAX

ON <expression> GOTO <list of line numbers>

EXAMPLES

```
10 INPUT "HOW MANY TIMES";N
20 ON N GOTO 60,50,40
30 GOTO 10
40 PRINT "MSX"
50 PRINT "MSX"
60 PRINT "MSX"
```

```
RUN
HOW MANY TIMES?2
MSX
MSX
```

POINTS TO REMEMBER

If the evaluated value is a real number, then the fractional part is truncated, i.e. if X=1.22 then X=1 and the first line number in the list is jumped to.

IF

<expression>=0
OR
<expression>> number of items in the list AND less than 255
THEN
BASIC continues to the next statement.

BUG HUNT

If the expression gives a negative number or more than 255, then an illegal function call error will result.

ASSOCIATED KEYWORDS AND REFERENCES

ON ... GOSUB
Section 1 Chapter 18: PROGRAM BRANCHING

ON ERROR GOTO

DESCRIPTION

ON ERROR GOTO enables error trapping, and specifies which line to GOTO once an error has been detected. Once the computer is told to trap errors, it will jump to the specified line if an error occurs while the program is running.

To disable error trapping, you must execute ON ERROR GOTO 0. If this is executed in an error trapping routine it will cause the BASIC to stop and print out the error which has caused the trap.

Using the ERROR statement, you can create your own customised error. There are about 36 errors between error code number 1 and 60 in MSX BASIC. Codes above 60, up to 255, are available for your use.

To program your own error, you must first of all put the computer into the error trapping mode by executing the ON ERROR GOTO <line> statement at the beginning of your program. Then, if in the middle of the program a condition arises such that you want to call the error trapping routine, you can do so by:

IF <condition> THEN ERROR <error code>.

ON ERROR GOTO will be activated and the computer will immediately jump to that error handling subroutine. You must have a line within the subroutine saying:

IF ERR=<error code> THEN PRINT "error message"

The error trapping subroutine can either terminate the program, or RESUME operation from any point in the program.

SYNTAX

ON ERROR GOTO <line>

EXAMPLES

Simple error trapping.

```
10 ON ERROR GOTO 1000
20 PRINT "MSX"
30 PRONT "MISTAKE"
40 END
1000 PRINT "ERROR DETECTED AT LINE";ERL
1010 END
```

```
LINE 30  THERE IS A MISTAKE, HERE
```

```
RUN
MSX
ERROR DETECTED AT LINE 30
Ok
```

POINTS TO REMEMBER

The MSX computer will not trap errors once it is executing the error trapping subroutine. If there is an error within an error trapping routine, BASIC will stop and the error message is displayed.

Not all errors can be dealt with by error trapping routines. It is therefore recommended to end the error trapping routine with ON ERROR GOTO 0, which will stop the BASIC program execution and display the error message.

Once the error trapping is enabled, the computer will GOTO the error trapping subroutine, even when you are in command mode.

When you are defining your own errors, start from 255 and work your way downwards.

If the computer encounters an error with no predefined error message, it will print "Unprintable Error".

ON ERROR disables all event handling traps such as ON INTERVAL and ON STRIG.

BUG HUNT

If you miss out the line number, you will get an undefined line number error.

ASSOCIATED KEYWORDS AND REFERENCES

ERL
ERR
ERROR
RESUME
Section 2 Chapter 10: ERROR HANDLING

ON INTERVAL GOSUB

DESCRIPTION

ON INTERVAL sets the subroutine and timed interrupt in a BASIC program. When the interval interrupt is enabled (with INTERVAL ON), the computer will — after a certain time interval — jump to the specified subroutine. This is one of the event handling statements which the MSX specialises in. (More on its application in the Event handling and Interrupt section.)

The interval is calculated in 50ths of a second and can be any length of time.

Once the time interrupt occurs, an automatic INTERVAL STOP is executed. This prevents the time interrupt occuring during the current time interrupt subroutine. It remembers, however, if an interrupt has happened during the current subroutine and if this has occurred, the computer will immediately execute the subroutine again, unless the current subroutine disables the time interrupt altogether by executing INTERVAL OFF. After it leaves the interrupt subroutine, the computer will automatically execute INTERVAL ON to enable the interrupt.

SYNTAX

ON INTERVAL=<time interval in 50ths of a second> GOSUB <line>

EXAMPLES

```
10 ON INTERVAL=500 GOSUB 40
20 INTERVAL ON
30 GOTO 30
35 REM INTERVAL SUBROUTINE
40 PRINT "10 SECONDS ELAPSED"
50 PRINT "END OF PROGRAM"
60 END
```

LINE 10 SETS TIME INTERVAL TO 500 UNITS (10 SECONDS). SUBROUTINE AT 40
LINE 20 ENABLES THE INTERRUPT

When you RUN this program you should get the following.

```
RUN                                THERE IS A TEN SECONDS DELAY
10 SECONDS ELAPSED
END OF PROGRAM
Ok
```

POINTS TO REMEMBER

Time interrupt is disabled as soon as the BASIC gets out of a program, i.e. no time interrupt in direct mode.

It is also disabled in error trapping subroutines.

BUG HUNT

Do not forget to switch the time interrupt on with INTERVAL ON.

ASSOCIATED KEYWORDS AND REFERENCES

INTERVAL ON/OFF/STOP

Section 2 Chapter 9 : EVENT HANDLING AND INTERRUPTS BY BASIC

ON KEY GOSUB

DESCRIPTION

ON KEY GOSUB sets the line numbers for function key interrupt subroutines. KEY (<number>) ON statements enable a function key interrupt for each of the function keys to be detected by the computer. When there is an interrupt, the computer jumps to the subroutine specifed by ON KEY GOSUB.

In the ON KEY GOSUB statement you must list all the subroutine line numbers for a corresponding function key. If there is no subroutine for a particular function key, skip the line number and just place a comma.

Once the F-KEY interrupt occurs, an automatic KEY (<number>) STOP is executed. This prevents the key interrupt occuring during the current interrupt subroutine. It remembers, however, if a function key has been pressed during the current subroutine, and the computer will immediately go to the corresponding subroutine for that key, if this has happened, unless the current subroutine disables KEY interrupt altogether by executing KEY <number> OFF. After it leaves the interrupt subroutine, the computer will automatically execute KEY ON to enable the interrupt.

SYNTAX

ON KEY GOSUB <list of line numbers>

EXAMPLES

If you press the function key 1 the computer will beep twice; if you press F2 it will beep only once.

```
10 ON KEY GOSUB 50,60
20 KEY (1) ON
30 KEY (2) ON
40 GOTO 40
50 BEEP
60 BEEP
70 RETURN
```

LINE 10 SETS SUBROUTINE FOR F1 AT LINE 50 AND F2 AT LINE 60
LINE 20 ENABLES FUNCTION KEY F1
LINE 30 ENABLES FUNCTION KEY F2
LINE 40 INFINITE LOOP TO WAIT FOR A FUNCTION KEY
LINE 50 BEEP (FOR F1)
LINE 60 BEEP (FOR BOTH)
LINE 70 RETURNS TO WHEREVER IT CAME FROM: IN THIS CASE LINE 40

POINTS TO REMEMBER

KEY interrupt is disabled when the program is not running and also during error trapping routines.

BUG HUNT

You get an Undefined line number error if you put a wrong line number in the list.

Do not forget to KEY ON.

ASSOCIATED KEYWORDS AND REFERENCES

KEY ON/OFF/STOP
Section 2 Chapter 9 . EVENT HANDLING AND INTERRUPTS BY BASIC

ON SPRITE GOSUB

DESCRIPTION

ON SPRITE GOSUB sets the sprite collision interrupt subroutine. The SPRITE ON statement enables a trap which diverts the program to the subroutine specified by ON SPRITE GOSUB when two sprites collide.

Once the interrupt occurs, an automatic SPRITE STOP is executed. This stops an interrupt occuring during the current sprite subroutine. It remembers, however, if there is another sprite collision during the subroutine, and in this case the computer will immediately execute the subroutine again after it has left the current subroutine, unless the current subroutine disables the sprite interrupt altogether by executing SPRITE OFF. After it leaves the interrupt subroutine, the computer will automatically execute SPRITE ON to enable the interrupt.

SYNTAX

ON SPRITE GOSUB <line>

EXAMPLES

In this example, you will see two square sprites, one yellow and another white, approaching each other from opposite sides of the screen. When they collide, ON SPRITE GOSUB will come into effect, and make the BASIC jump to the sprite interrupt subroutine. The SPRITE OFF in the sprite interrupt routine prevents any further detection of sprite collisions. (Try this routine without SPRITE OFF and see what happens.)

```
10 ON SPRITE GOSUB 110
20 SCREEN 2,0
30 SPRITE$(0)=STRING$(8,CHR$(255))
40 SPRITE$(1)=STRING$(8,CHR$(255))
50 SPRITE ON
60 FOR I=10 TO 240
70 PUT SPRITE 0,(I,100),11,0
80 PUT SPRITE 1,(250-I,100),15,1
90 NEXT I
100 END
105 REM SPRITE COLLISION ROUTINE
110 SPRITE OFF
120 BEEP
130 RETURN
```

LINE 10 SETS TRAP SUBROUTINE TO 110
LINE 20 SETS GRAPHICS SCREEN

```
LINE  30    SPRITE 0 IS A SQUARE
LINE  40    SO IS SPRITE 1
LINE  50    SPRITE COLLISION DETECTOR ON
LINE  60    LOOP
LINE  70    SPRITE (YELLOW) MOVES FROM LEFT
LINE  80    SPRITE (WHITE) MOVES FROM RIGHT
LINE  90    NEXT LOOP
LINE 100    FINISH
LINE 110    KILLS TRAP (ONCE IS ENOUGH!)
LINE 120    NOISE INDICATOR
LINE 130    GO WHEREVER THE COMPUTER LEFT OFF
```

POINTS TO REMEMBER

SPRITE interrupt is disabled when the program is not running, and also during error trapping routines.

BUG HUNT

You get an Undefined line number error if you give a wrong line number for the interrupt subroutine.

Do not forget to SPRITE ON.

ASSOCIATED KEYWORDS AND REFERENCES

SPRITE ON/OFF/STOP
SPRITE$
PUT SPRITE
Section 2 Chapter 15 : ADVANCED GRAPHICS IV : SPRITE GRAPHICS

ON STOP GOSUB

DESCRIPTION

ON STOP GOSUB is an interrupt handling statement which prevents the user from breaking into a program already running, by pressing <CTRL><STOP>.

The only way to break out of a break-proofed program is to RESET the computer. Therefore you should remember to save your break-proofed program before you RUN it.

STOP ON/OFF/STOP enables/disables STOP KEY trapping. When STOP ON is executed, the BASIC starts to check if <CTRL><STOP> has been pressed each time it executes a new statement. If a <CTRL><STOP> has been detected, then the BASIC is diverted to the subroutine specified by the ON STOP GOSUB statement executed earlier in the program.

Once the interrupt occurs an automatic STOP STOP is executed. This stops the interrupt occuring during the current subroutine. It remembers, however, if <CTRL><STOP> is pressed again during the current subroutine, and the computer will immediately goto the subroutine again after it has left the current subroutine if this has happened, unless the current subroutine disables the STOP interrupt altogether by executing STOP OFF. After it leaves the interrupt subroutine, the computer will automatically execute STOP ON to enable the interrupt.

SYNTAX

ON STOP GOSUB <line>

EXAMPLES

This program shows you how ON STOP GOSUB can prevent the user from breaking into your program. If you press <CTRL><STOP> it will say "control stop disabled" and carries on executing the current program. This routine has a special exit routine: press <s> to exit: otherwise MSX will be printed out indefinitely.

```
10 ON STOP GOSUB 100
20 STOP ON
30 IF INKEY$="s" THEN END
40 PRINT "MSX"
50 GOTO 30
90 REM CTRL-STOP SUBROUTINE
100 BEEP
110 PRINT"CONTROL STOP DISABLED"
120 RETURN
```

```
LINE 10   STOP SUBROUTINE SET TO LINE 100
LINE 20   SWITCHES ON THE STOP DETECTOR
LINE 30   IF <s> IS PRESSED THEN END
LINE 40   PRINTS MSX
LINE 50   LOOPS BACK TO LINE 30
LINE 100  WARNING BEEP
LINE 110  MESSAGE
LINE 120  RETURN TO WHEREVER THE INTERRUPT WAS DETECTED
```

POINTS TO REMEMBER

Note that ON STOP trapping does not prevent the <STOP> key halting the program. It just prevents you from breaking into a program by <CTRL><STOP>. Try pressing the <STOP> key only in the above program. You will find that it will halt the program and display the cursor. If you press the <STOP> key for the second time, the program will continue.

Remember to execute STOP ON to enable the STOP trapping.

The STOP interrupt is disabled when the program is not running and also during error trapping routines.

BUG HUNT

You get an Undefined line number error if you give a wrong line number for the STOP interrupt subroutine.

ASSOCIATED KEYWORDS AND REFERENCES

STOP ON/OFF/STOP
Section 2 Chapter 9 : EVENT HANDLING AND INTERRUPTS BY BASIC

ON STRIG GOSUB

DESCRIPTION

ON STRIG GOSUB sets trigger interrupt subroutines for the joy stick and the space bar trigger. STRIG(<n>) ON activates the trigger trapping and is used in conjunction with ON STRIG GOSUB.

There are five triggers:
0=Space bar
1=trigger 1 of joystick 1
2=trigger 1 of joystick 2
3=trigger 2 of joystick 1
4=trigger 2 of joystick 2

In the ON STRIG GOSUB statement, you must list the subroutine line numbers for all the triggers (except when using the space bar only – see syntax section). If there is no subroutine for a particular trigger, you may omit it and just place a comma.

Once the interrupt occurs an automatic STRIG(<n>) STOP is executed (<n> is the trigger number). This stops the interrupt occuring during the current trigger subroutine. It remembers, however, if a trigger is pressed during this subroutine, and the computer will immediately goto the corresponding subroutine for that trigger after it has left the current subroutine, unless the current subroutine disables the trigger interrupt altogether by executing STRIG(<n>) OFF. After it leaves the interrupt subroutine, the computer will automatically execute STRIG(<n>) ON to enable the interrupt.

SYNTAX

ON STRIG GOSUB <list of line numbers>
ON STRIG GOSUB <line> only when the space bar is used as the only trigger.

EXAMPLES

Here is a very short program which demonstrates how the trigger trapping works with the space bar as a trigger. When the space bar is pressed, the program jumps to the trigger trap subroutine. Otherwise, it will print out MSX continuously until the "s" key is pressed.

```
10 ON STRIG GOSUB 100
20 STRIG(0) ON
30 IF INKEY$="s" THEN END
40 PRINT "MSX"
50 GOTO 30
90 REM SPACE TRIGGER ROUTINE
100 BEEP
110 PRINT "SPACE BAR PRESSED"
120 RETURN
```

LINE 10 SETS TRIGGER ROUTINE TO LINE 100
LINE 20 SWITCH ON
LINE 30 IF <s> IS PRESSED THEN END
LINE 40 PRINTS MSX
LINE 50 LOOPS BACK TO LINE 30
LINE 100 WARNING BEEP
LINE 110 GIVES A MESSAGE
LINE 120 RETURNS TO WHEREVER THE INTERRUPT WAS DETECTED

```
RUN
MSX
MSX
MSX
MSX
SPACE BAR PRESSED                    HIT <space>
MSX
MSX
MSX
ok                                   HIT <s>
```

POINTS TO REMEMBER

STRIG interrupt is disabled when the program is not running and also during error trapping routines.

BUG HUNT

A common mistake:
STRIG (0) ON ..... This is wrong and causes a Syntax error. There should be no space between STRIG and (0).

You get an Undefined line number error if you put a wrong line number in the list.

ASSOCIATED KEYWORDS AND REFERENCES

STRIG()
STRIG ON/OFF/STOP
Section 2 Chapter 9 : EVENT HANDLING AND INTERRUPTS BY BASIC

OPEN

DESCRIPTION

OPEN declares a specified device opened. It allocates a buffer for the input/output (I/O) and sets the mode of I/O operation for the buffer. If you are inputting or outputting to a file, you must first have an opened file. You need to execute OPEN before the following statements:

PRINT#	PRINT# USING
INPUT#	LINE INPUT#
INPUT$(#)	EOF

There are four devices supported in the current version of MSX.
<device descriptor>

CAS : cassette
CRT : text screen
GRP : graphic screen
LPT : line printer

The number of devices will be increased when floppy disc and other peripherals come out. The device descriptor can be increased by using a ROM cartridge. Don't forget the colons after the device description.

There are three modes of I/O.

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

<file number> is an integer whose value is between one and MAXFILES, the maximum number of files. File numbers are also quoted in PRINT# statements, etc. The <file number> is associated with the file as long as it is not CLOSEd.

SYNTAX

OPEN "<device descriptor>[<file name>]" [FOR <mode>] AS [#]<file number>

[<file name>], [FOR <mode>], and [#] are optional.

EXAMPLES

To write characters to the graphic screen, you must open a file to the GRP. Then, you can use PRINT# to print to the graphic screen at the graphics cursor position.

```
10 OPEN "GRP:" FOR OUTPUT AS #1
20 SCREEN 2
30 DRAW "BM30,145"
40 PRINT#1,"GRAPHICS PRINT"
50 GOTO 50
```

LINE 10 OPENS A FILE TO GRAPHICS SCREEN
LINE 20 GRAPHICS SCREEN
LINE 30 POINT TO 30, 145
LINE 40 PRINT#1
LINE 50 HOLDS GRAPHICS MODE

POINTS TO REMEMBER

MAXFILES is 1 on default, but if you are using a number of devices in a program it is a good idea to set MAXFILES to a higher number.

BUG HUNT

If the file number is greater than that set by MAXFILES, then a Bad file number error will result.

ASSOCIATED KEYWORDS AND REFERENCES

MAXFILES
PRINT#
PRINT# USING
INPUT#
LINE INPUT#
INPUT$
VARPTR
EOF
Section 2 Chapter 14 : ADVANCED GRAPHICS III :
HOW TO PRINT TO GRAPHICS SCREEN
Section 2 Chapter 19 : HOW TO USE FILES

OR Logical OR operator

DESCRIPTION

OR is one of the logical operators which performs bit manipulation and Boolean algebra. It is also used in IF THEN ELSE statements to test more than one condition, before resultant action is taken.

The truth table for OR is:

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

Example 34 OR 67

	34	0000000000100010
OR	67	0000000001000011
	99	0000000001100011

OR is used in IF/THEN/ELSE statements and gives a mutliple choice of the possible conditions.

SYNTAX

IF <condition> OR <condition> .... THEN ..... ELSE

<num-var>=<numeric> OR <numeric>

EXAMPLES

Boolean Algebra.

```
10 A=134
20 B=213
30 PRINT "A         ";A;BIN$(A)
40 PRINT "B         ";B;BIN$(B)
50 PRINT "A OR B    ";A OR B;BIN$(A OR B)
```

```
RUN
A         134   10000110
B         213   11010101
A OR B    215   11010111
```

IF <condition> OR <condition> THEN

```
10 INPUT "A=";A
20 INPUT "B=";B
30 IF A>100 OR B>100 THEN PRINT
"EITHER A OR B IS GREATER THAN 100"
40 END
```

```
RUN
A=? 78
B=? 107
EITHER A OR B IS GREATER THAN 100
```

POINTS TO REMEMBER

You can use OR to merge two bytes to create a particular value.

Boolean operands must be in the integer range of −32768 and 32767.

Real numbers are truncated to integers.

BUG HUNT

An Overflow error results when the number is out of the integer range.

Type mismatch occurs if the operands are strings.

ASSOCIATED KEYWORDS AND REFERENCES

IF
THEN
ELSE
AND
NOT
IMP
XOR
EQV
Section 1 Chapter 6 : THE USE OF CONDITIONS
Section 2 Chapter 6 : BOOLEAN ALGEBRA
Section 2 Chapter 7 : BOOLEAN II : THE IF/THEN ELSE

OUT

DESCRIPTION

This sends a byte to a specified output port . DO NOT use this statement in commercial software as it will be machine dependent and will lose the MSX compatibility.

SYNTAX

OUT <port>,<data>
<port> = port number between 0 and 255
<data> = data byte between 0 and 255

EXAMPLES

OUT 1,166

POINTS TO REMEMBER

Use BIOS instead of OUT statements, so that you will not lose the MSX compatibility.

BUG HUNT

Any negative or number greater than 255 is meaningless.

ASSOCIATED KEYWORDS AND REFERENCES

WAIT
INP

PAD touch PAD

DESCRIPTION

MSX can support two touch pads connected to the joystick ports. PAD returns the status values of the touch pads.

SYNTAX

PAD(<n>)
n=0 to 3 for touch pads connected to joystick port 1.
PAD(0)= − 1 if touched
0 if not touched
PAD(1)= x-coordinate (0 − 255) on pad 1
PAD(2)= y-coordinate (0 − 255) on pad 1
PAD(3)= − 1 if touch pad switch is pressed
0 if touch pad switch is pressed

n=4 to 7 for touch pad connected to joystick port 2.

PAD(4)= − 1 if touched
0 if not touched
PAD(5)= x-coordinate (0 − 255) on pad 2
PAD(6)= y-coordinate (0 − 255) on pad 2
PAD(7)= − 1 if touch pad switch is pressed
0 if touch pad switch is pressed

EXAMPLES

This is a short program to use the touch pad as a drawing board.

```
10 SCREEN 2,0,0
20 IF NOT PAD(0) THEN 20
30 X=PAD(1)
40 Y=INT(PAD(2)*192/255)
50 PSET (X,Y)
60 GOTO 20
```

LINE 10 HI-RES GRAPHICS MODE
LINE 20 IF NOT TOUCHED REPEATS SAME LINE
LINE 30 GET X
LINE 40 GET Y TO FIT THE SCREEN
LINE 50 PLOT A POINT

POINTS TO REMEMBER

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

ASSOCIATED KEYWORDS AND REFERENCES

Section 2 Chapter 23: PERIPHERAL DEVICES.

PAINT

DESCRIPTION

PAINT fills an area surrounded by a border line with the specified colour.

You must specify the coordinates of where you want the computer to start painting. If you are filling in a particular shape, then point the coordinates to anywhere within that shape.

SYNTAX

PAINT <coordinates specifier>
paints in the current foreground colour.

PAINT <coordinates specifier>,<colour>
paints with the specified colour.

PAINT <coordinates specifier>,<colour>,<border line colour>
paints, with the specified colour, the area surrounded by the specified border line color. (Multi-colour mode only.)

<coordinates specifier>

1) <x-coordinate>,<y-coordinate>
   x range = 0-255 y range = 0-191
2) STEP (<x-offset>,<y-offset>)
   These coordinates are of a point relative to the last point referred to.

NOTES: <x-offset>, <y-offset>, <x-coordinates>, <y-coordinates> can be <num-var>, <num-exp> or simply <num-con>.

COLOUR CODE

0 Transparent	8 Medium red
1 Black	9 Light red
2 Medium green	10 Dark yellow
3 Light green	11 Light yellow
4 Dark blue	12 Dark green
5 Light blue	13 Magenta
6 Dark red	14 Grey
7 Cyan	15 White

EXAMPLES

1) High resolution graphics mode (SCREEN 2)

In the hi-res mode the border line colour must be the same colour as the paint colour, otherwise the paint will "spill" over. In this example no colour is specified so it uses the default colour: this means the lines are drawn in white, onto a blue background and then the shape is painted white.

```
10 SCREEN 2
20 PSET (100,100)
30 LINE -STEP(-50,50)
40 LINE -STEP(100,0)
50 LINE -STEP(50,-50)
60 LINE -STEP(-100,0)
70 PAINT (110,110)
100 GOTO 100
```

2) In mode 3, the low resolution graphics mode, you can paint with a different colour from the border line's colour.

```
10 SCREEN 3
20 PSET (100,100)
30 LINE -STEP(-50,50)
40 LINE -STEP(100,0)
50 LINE -STEP(50,-50)
60 LINE -STEP(-100,0)
70 PAINT (110,110),9,15
100 GOTO 100
```

POINTS TO REMEMBER

You can only specify the border line colour in the multi-colour mode. You can only use one colour per shape painted. This means you cannot have multi-colour border lines.

BUG HUNT

If the coordinates specified are outside the screen, an Illegal function call error will result.

ASSOCIATED KEYWORDS AND REFERENCES

DRAW
SCREEN
Section 1 Chapter 27 : PAINTING

PDL PADDLE

DESCRIPTION

PDL returns the value for Paddle.
You can connect up to 12 games paddles, 6 on each joystick port.

SYNTAX

PDL(<n>)
Port 1 n=1,3,5,7,9,11
Port 2 n=2,4,6,8,10,12

PDL returns a value between 0 and 255.

EXAMPLES

P1=PDL(1)

ASSOCIATED KEYWORDS AND REFERENCES

Section 2 Chapter 23 : PERIPHERAL DEVICES.

PEEK

DESCRIPTION

PEEK returns a byte, read from a specified memory location.

SYNTAX

PEEK(<integer-numeric>)
Range – 32768 to 65535

EXAMPLES

```
10 FOR I=1 TO 10000
20 A=PEEK(I)
30 IF A>31 AND A<127 THEN PRINT
HEX$(I);" ";CHR$(A)
40 NEXT I
```

```
RUN
A  &
12  &
19  E
21  j
25  ^
..
```

POINTS TO REMEMBER

POKE is the complimentary statement to PEEK.
You cannot peek the video RAM with this: use VPEEK instead.

ASSOCIATED KEYWORDS AND REFERENCES

POKE
VPOKE
VPEEK
See SYSTEM VARIABLE CHART
Section 2 Chapter 20 : MEMORY MAP

PLAY PLAY music

DESCRIPTION

PLAY plays music according to the Music Macro Language which is entered in a string format in a similar manner to that of the Graphics Macro Language.

The music macro language:

A,B,C,D,E,F,G with optional #,+,-

Plays the note indicated in the current octave.
or + after the letter means SHARP
- <minus> means FLAT.
Note that the use of #, +, - is valid only if such a note exists, i.e. corresponds to a black note on the piano.

O <n> n=1 to 8

Selects the Octave from 1 to 8.
Each Octave goes from C to B.
Initial default octave is 4, in which Octave, C corresponds to "Middle C" on a piano.

N <n> n=0 to 96

This plays note n. This is an alternative way to play any note within the 8 octaves available

L <n> n=1 to 64

This sets the duration of the note.

L1 = whole note	Semibreve
L2 = half note	Minim
L4 = quarter note	Crotchet
L8 = 8th note	Quaver

etc....
The duration of a particular note may be changed with <n> following the note. For example, L8C is the same as C8. This cannot be used with the N command.
The default value is L4.

R <n> n=1 to 64

Rest or pause. The duration of the rest is set in a similar manner to L.
R1 = rests whole note
R2 = rests half note
R4 = rests quarter note
R8 = rests 8th note
etc....
Note: R and R0 are equal to the default rest period which is R4.

• (dot)

Dot after a note A to G, N, or rest R increases the length by 3/2, i.e. played as a dotted note.

You can put more than one dot after a note.

T <n> n=32 to 255 TEMPO

The tempo sets the number of quarter notes (or crotchets) in a minute.
n may range from 32 to 255. The default value is 120.

V <n> n=0 to 15

Volume. V sets the volume of output.
0 being the lowest volume, 15 the highest.
The default value is 8.

M <n> n=1 to 65535

Modulation. This sets the period of the envelope specified by S.
The default value is 255.

S <n> n=0 to 15

This gives the shape of the envelope. There are several predefined envelope patterns which you can chose from.
See SOUND for detail.

X<variable>;

X command executes what is in the string variable as the music macro language.

In all the above commands, n can be an integer numerical constant, or it can be a variable in the form of "=<variable>;" where the variable name is surrounded by "=" and ";".

When you execute a BEEP, you will reset the sound system to the default values.

SYNTAX

PLAY <string-exp for voice 1>,<string-exp for voice 2>,<string-exp for voice 3>

where voice 2 and 3 are optional.

EXAMPLES

```
10 A$="CDEFGAB"
20 FOR I= 1 TO 8
30 PLAY "O=I;XA$;"
40 NEXT I
```

POINTS TO REMEMBER

Do not get this mixed up with the function PLAY() which is rather different.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 28 : THE MUSIC MACRO LANGUAGE

PLAY () PLAY function

DESCRIPTION

This tells you if the computer is playing music from a particular channel.

Returns – 1 if still playing music.
0 if not.

PLAY(0) checks all three channels 1, 2, & 3, and if one or more of them is playing, it returns – 1. If none, then 0.

SYNTAX

<num-var>=PLAY(<play channel>)
<play channel>=0 to 3

POINTS TO REMEMBER

This is different from the PLAY statement.

ASSOCIATED KEYWORDS AND REFERENCE

PLAY
Section 1 Chapter 28: THE MUSIC MACRO LANGUAGE

POINT

DESCRIPTION

This finds the colour of the specified point.
It does not return the colour of a foreground sprite.

SYNTAX

POINT (<x-coordinate>,<y-coordinate>)

EXAMPLES

This is a short program which finds the colour of the graphics screen. Line 10 tells the computer to PRINT#1 to output to the graphics screen.

```
10 OPEN "GRP:" FOR OUTPUT AS #1
20 SCREEN 2
30 COLOUR 14,8,9
40 CLS
50 P=POINT(100,100)
60 PRINT#1,"COLOUR IS ";P
70 GOTO 70
```

You see the screen turn red and
COLOUR IS 8
displayed.

POINTS TO REMEMBER

Returns 0 in both text MODEs 0 and 1.
Returns -1 when the coordinates are off the screen.

ASSOCIATED KEYWORDS AND REFERENCE

COLOR
SCREEN
PSET
PRESET
Section 1 Chapter 23 : PLOTTING POINTS

POKE

DESCRIPTION

POKE writes a byte to a specified memory address.

POKE is mainly used for inserting machine code into a BASIC program. The most useful area for a machine code routine is in the user work area which is created by the CLEAR statement.

SYNTAX

POKE <address>,<integer-exp>

<address> can be between – 32768 and 65535. If the number is negative, the address is calculated by subtracting from 65535.

<integer-exp> must be a one byte number, i.e. between 0 and 255.

EXAMPLES

POKE 65535,0

POINTS TO REMEMBER

Be careful with this statement as you can crash the system by poking into certain memory locations.

To poke into the video RAM, use VPOKE.

BUG HUNT

An Overflow error will occur if <address> or <integer-exp> is out of range.

ASSOCIATED KEYWORDS AND REFERENCE

CLEAR
PEEK
VPEEK
VPOKE
See SYSTEM VARIABLE CHART

POS cursor POSition

DESCRIPTION

This returns the current text cursor position. You need a dummy variable (0).

SYNTAX

POS(0)

EXAMPLES

LET P=POS(0)

POINTS TO REMEMBER

Left most position is 0.

Right most position depends on the screen mode.

	WIDTH	
	default	maximum
screen 0	37	40
screen 1	29	32

BUG HUNT

A common mistake is to forget the (0) dummy variable.

ASSOCIATED KEYWORDS AND REFERENCE

CRSLIN
WIDTH
Section 2 Chapter 12: ADVANCED GRAPHICS I
CHARACTERISTICS OF EACH SCREEN MODE

PRESET Point RESET

DESCRIPTION

This resets a specific point to the background colour.

If a colour is specified, it plots that colour at the given point. (This is the same as PSET.)

SYNTAX

PRESET <coordinates specifier>
PRESET <coordinates specifier>,<colour>

<coordinates specifier>
1) (<x-coordinate>,<y-coordinate>)
2) STEP (<x-offset>,<y-offset>)

<x-offset>, <y-offset>, <x-coordinates>, <y-coordinates> can be <num-var>, <num-exp>, or simply <num-con>

EXAMPLES

```
10 SCREEN 2
20 CLS
30 PAINT (10,10),15
40 X=RND(1)*255
50 Y=RND(1)*255
60 PRESET (X,Y)
70 GOTO 40
```

LINE 10 HIGH RESOLUTION GRAPHICS MODE 2
LINE 20 CLEARS THE SCREEN TO BLUE
LINE 30 PAINTS SCREEN WITH WHITE (FOREGROUND COLOUR)
LINE 40 X COORDINATE
LINE 50 Y COORDINATE
LINE 60 POINT RESET TO BACKGROUND COLOUR (BLUE)
LINE 70 LOOP BACK FOR MORE

You see a blue screen painted white, then lots of blue dots appear at random.

POINTS TO REMEMBER

PRESET <coordinates specifier>,<colour> is the same as
PSET <coordinates specifier>,<colour>

BUG HUNT

PRESET will do nothing if the point is off the screen. However, it will give an Overflow error if the coordinates are out of the integer range (– 32678 to 32767).

ASSOCIATED KEYWORDS AND REFERENCE

PSET
POINT
COLOR
Section 1 Chapter 23 : PLOTTING POINTS

PRINT

DESCRIPTION

This statement is the most commonly used statement in BASIC. It simply PRINTs out what ever you want on the screen.

After the PRINT statement, you can have a whole list of items to be printed out. They may be numeric or string variables, or strings in quotation marks. Their position is determined by the punctuation used within the statement, or by using the TAB or the LOCATE statements.

The BASIC divides each line into zones, 14 characters long. A comma tells the computer to print each item in the list, at the beginning of each 14 character zone.

A semicolon, on the other hand, will cause each item in the list to be printed sequentially on the same line. Typing one or more <space>s between the items has exactly the same effect as a semicolon.

If a PRINT statement ends with a semicolon, the next PRINT statement will print on the same line with the same spacings as given above, i.e. there is no carriage return.

If the <list of items> to be printed ends with nothing, then a carriage return is executed and the next PRINT statement will PRINT on the next line.

If there are too many characters to fit on a line, then the computer will continue to print on the next line.

Printed numbers are always followed and preceded by a <space>. Printed negative numbers are preceded by a minus sign.

The abbreviation for PRINT is "?".

SYNTAX

PRINT <list of items>
? <list of items>

EXAMPLES

```
10 PRINT "MESSAGES MUST BE ENCLOSED IN
          QUOTATION MARKS"
20 PRINT "SEMICOLON MEANS CONTINUE AT
          THE LAST POINT ";
30 PRINT "WHERE YOU LEFT OFF"
40 A=100
50 B=300
60 PRINT "A=";A,"B=";B
70 PRINT "A+B=";A+B
```

```
RUN
MESSAGES MUST BE ENCLOSED IN
QUOTATION MARKS
SEMICOLON MEANS CONTINUE AT T
HE LAST POINT WHERE YOU LEFT
OFF
A=100            B=300
A+B= 400
ok
```

The above display assumes screen width to be 29 characters long.

POINTS TO REMEMBER

Note that the abbreviated print "?" becomes "PRINT" automatically when listed.

It is easier to use PRINT USING when printing out a table.

BUG HUNT

Always use quotation marks for strings.

ASSOCIATED KEYWORDS AND REFERENCE

PRINT USING
PRINT#
TAB
LOCATE
Section 1 Chapter 2 : COMMAND MODE
Section 1 Chapter 9 : MORE ON PRINT AND THE SCREEN

PRINT#

DESCRIPTION

To write (print out) to various devices and to the graphics screen, you must OPEN a channel and then use a special print statement, PRINT#. The "#" sign is followed by the file number which is specified in the OPEN statement.

PRINT# is also often used in storing files on cassette. (See example 2 for the full procedure.)

SYNTAX

PRINT#<file number>

EXAMPLES

```
10 OPEN "GRP:" FOR OUTPUT AS #1
20 SCREEN 2
30 DRAW "BM30,100"
40 PRINT#1,"PRINT IN HI-RES MODE"
50 GOTO 50
```

LINE 10 OPENS A FILE TO THE GRAPHICS SCREEN AS #1
LINE 20 SELECTS GRAPHICS SCREEN 2
LINE 30 MOVES GRAPHICS CURSOR TO 30,100
LINE 40 PRINTS TO GRAPHICS SCREEN
LINE 50 LOCKS GRAPHICS SCREEN

POINTS TO REMEMBER

A channel must be OPENed before you use PRINT#.

INPUT# is complementary to PRINT#.

BUG HUNT

A bad file number error occurs when the channel number referred to is not opened.

ASSOCIATED KEYWORDS AND REFERENCE

MAXFILES
INPUT#
OPEN
CLOSE
Section 2 Chapter 14: ADAVNCED GRAPHICS III
HOW TO PRINT TO GRAPHICS SCREEN USING FILES
Section 2 Chapter 19: HOW TO USE FILES

PRINT USING

DESCRIPTION

The PRINT USING statement allows you to print strings or numbers using a specified format. It will help you to tabulate information neatly.

SYNTAX

PRINT USING <string-exp>;<list of items>

<list of items> can be numbers or strings. They must be separated by a semicolon.

<string-exp> is one of the special formatting characters. It can be either a string or a string variable.

CHARACTER EXPLANATION

!
Specifies that the first character of the string is to be printed.
F$="MSX": PRINT USING "!";F$
M

@
Inserts specified string into the position given by @.
PRINT USING "ABC@EFG";"MSX"
ABCMSXEFG

#
The "#" sign indicates that a digit is to be printed, e.g:
PRINT USING "#.###";1.3 prints
1.300 i.e. it formats the number to be displayed.
As you can see in the above example, you may include a decimal point in the formatting string with the "#" sign. If the number has fewer digits than specified by the formatting characters, it will be right-justified with preceding spaces.
PRINT USING "###.##";65.87
65.87

+
A "+" plus sign at the begining or the end of the formatting string will print the sign of the number, i.e. "+" or "-" at the position specified.
PRINT USING "+###.#####";- 0.123422
- 0.12342
PRINT USING "#### #+";10.71
10.7+

**
The double asterisk sign causes the leading spaces in the numeric field to be filled with "*". They represent two more digit spaces where you can have numbers, in the field.
PRINT USING "**#.##";5.55, 5.55
**5.55* 5.55

$$** The double **$$** dollar sign puts a dollar sign in front of a number. (pound sign in U.K.?) **$$** specifies two more digit spaces in the field, one of which will be the **$ dollar sign.
PRINT USING "**$$**#####.##";10000,99999.99,– 100.55
$**10000.00 **$99999.99 –**$**100.55
You cannot use the exponential format in conjunction with **$$** sign.

****$** This sign is a combination of ** and the **$$** sign. All blank spaces are filled by * and the number is preceded with a **$** sign. ****$** specifies three more digit positions, one of which is the **$** sign.
PRINT USING "****$**###.##",34.99
*****$**34.99

, A comma placed to the left of the decimal point in the formatting string causes a comma to be printed every third digit to the left of the decimal point.
PRINT USING "#######,.##";2000000
2,000,000.00
A comma at the end of the formatting string causes a comma to be printed at the end of the number.
PRINT USING "##.##,";12.567
12.57,

^^^^ This is the exponent formatting character and allows the space for E+xx. You can specify the position of decimal point. Significant digits are left justified, and the exponent is adjusted.
PRINT USING "##.##^^^^";200.00
2.00E+02

EXAMPLES

PRINT USING "**$$**####..##";10000;2000.50;3000.555
$**10,000.00 **$2,000.50 **$**3,000.56

There are lots of examples given in the Advanced Programming Guide.

POINTS TO REMEMBER

If the number to be printed does not fit into the field specified by the format characters, a % percent sign is printed in front of the number. This also happens when the rounded number exceeds the size of the field.

E.g.
PRINT USING "##.##";1000
%1000.00

```
PRINT USING "##.##";99.999
%100.00
```

BUG HUNT

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

ASSOCIATED KEYWORDS AND REFERENCE

PRINT# USING
Section 2 Chapter 8: PRINT USING

PRINT# USING

DESCRIPTION

PRINT# USING is exactly the same as PRINT USING but this writes to various devices. You must execute an OPEN statement, to open a channel to a specific device, before using this statement.

SYNTAX

PRINT#<file number>,USING <string-exp>;<list of expression>

EXAMPLES

PRINT#2, USING "##.###";3.453729

POINTS TO REMEMBER

A channel must be OPENed before you use PRINT# USING.

BUG HUNT

A Bad file number error results when the channel number referred to is not OPENed.

ASSOCIATED KEYWORDS AND REFERENCE

PRINT USING
MAXFILES
PRINT#
OPEN
CLOSE
Section 2 Chapter 8: PRINT USING
Section 2 Chapter 14: ADAVANCED GRAPHICS III : HOW TO PRINT TO GRAPHICS SCREEN USING FILES
Section 2 Chapter 19. HOW TO USE FILES

PSET Point SET

DESCRIPTION

PSET plots a point at specified coordinates in the current foregound colour or a specified colour.

SYNTAX

PSET <coordinate specifier>
PSET <coordinate specifier>,<colour>

<coordinates specifier>
1) (<x-coordinate>,<y-coordinate>)
2) STEP (<x-offset>,<y-offset>)

<x-offset>, <y-offset>, <x-coordinates>, <y-coordinates> and <colour> can be <num-var>, <num-exp>, or simple <num-const>.

EXAMPLES

This program shows a black sky filled with mutlicolour stars.

```
10 COLOR 15,1,1
20 SCREEN 2
30 X=RND(1)*255
40 Y=RND(1)*255
50 Z=INT(RND(1)*16)
60 PSET (X,Y),Z
70 GOTO 30
```

LINE 10 BLACK BORDER AND BACKGROUND
LINE 50 RANDOM COLOUR

POINTS TO REMEMBER

PRESET with the colour specified is exactly the same as PSET with the colour specified.

BUG HUNT

PSET will do nothing if the point referred to is outside the screen. However, it will give an Overflow error if it is out of the integer range.

ASSOCIATED KEYWORDS AND REFERENCE

COLOR
PRESET
Section 1 Chapter 23: PLOTTING POINTS

PUT SPRITE

DESCRIPTION

This puts a sprite on to the screen. You can place one sprite per sprite plane. There are 32 sprite planes, therefore the maximum number of sprites displayed at any one time is 32.

The coordinates specifier tells the computer where to place the sprite. You can hide a sprite beyond the edge of the screen, or move behind and over another sprite.

You can use all of the 16 colours. However, you can only use one colour per sprite.

See the Sprites section in the Advanced Programming Guide on how to use sprites.

SYNTAX

PUT SPRITE <sprite plane number>[,<coordnate specifier>]
[,<colour>][,<pattern number>]

All arguments can be either <num-const>, <num-var>, or <num-exp> but within their alllowed range.

<sprite plane number> = 0 to 31

<coordinates specifier>
1) (<x-coordinate>,<y-coordinate>)
2) STEP (<x-offset>,<y-offset>)

Coordinates range: between −32768 to 32767.
Sprite Screen coordinates range: X = − 32 to 255
Y = − 32 to 191

If the coordinates go outside their sprite screen range, automatic wrap-around will take place. (See Advanced Programming Guide).

<pattern number> = 0 to 256 if sprite size is 8 by 8 pixels
= 0 to 64 if sprite size is 16 by 16 pixels

EXAMPLES

```
10 SCREEN 2
20 SPRITE$(0)=STRING$(8,CHR$(255))
30 X=100 : Y=120
40 C=7
50 PUT SPRITE 0,(X,Y),C,0
60 GOTO 60
```

LINE 10 SELECTS SCREEN MODE 2
LINE 20 DEFINES A SQUARE SPRITE (0)
LINE 30 X AND Y COORDINATES
LINE 40 COLOUR IS CYAN (7)
LINE 50 PUT SPRITE PATTERN 0 ON SPRITE PLANE 0 AT COORDINATES X, Y IN CYAN

POINTS TO REMEMBER

When either 208 (&HD0) or 209 (&HD1) is given to the Y coordinate, it has an effect of hiding one or more sprites temporarily. (See Advanced Programming Guide: Sprite section for more detail on its appplication.)

You cannot use sprites in text mode 0.

The sprite size is determined by the SCREEN statement.

If the <coordinate specifier> is ommitted, the computer PUTs the SPRITE at the last point referred to.

BUG HUNT

An Overflow error will result if the coordinate specifier is out of integer range.

ASSOCIATED KEYWORDS AND REFERENCE

ON SPRITE GOSUB
SPRITE$
SPRITE ON/OFF/STOP
SCREEN
COLOR
Section 1 Chapter 15: ADVANCED GRAPHICS IV: SPRITE GRAPHICS

READ

DESCRIPTION

READs data from DATA statements and assigns it to variables in the READ statements.

The READ statement is always used in conjunction with the DATA statement. The data is stored in DATA statements which are non-executable statements whose only purpose is to store information for the program. To access the information from a DATA statement, use the READ statement.

You can read a list of variables in one READ statement. They can be read into arrays, strings or numeric variables, but the data types must match or Syntax error will result.

The READ statement reads from the DATA statement with the smallest line number and proceeds to increasing line numbers. If you want to READ some data from a certain line, you must use the READ statement in conjunction with the RESTORE statement which points to the DATA statement READ should refer to.

SYNTAX

READ <list of variables>

EXAMPLES

```
10 READ A$,B
20 PRINT A$;B
30 READ C$,D
40 PRINT C$;D
50 DATA "ASTRONOMY",500,"PHYSICS",600
```

```
RUN
ASTRONOMY 500
PHYSICS 600
```

POINTS TO REMEMBER

As shown in the above program, the computer remembers the last item of DATA read. The next READ statement reads the first unread DATA item after the last read data item.

BUG HUNT

A Syntax error results when the READ variable does not match the DATA type.

An Out of DATA error occurs if you try to READ, when you have read all the data already. Remedy : use RESTORE.

ASSOCIATED KEYWORDS AND REFERENCE

DATA
RESTORE
Section 1Chapter 12: READING DATA INTO ARRAYS

REM REMark

DESCRIPTION

This enables the programmer to put comments into a program. A REM statement is not executed, i.e. the computer skips a REM statement when it sees one.

REM statements are used to document programs. When you are writing a program it is a good idea to put in REM statements to make a note of what each section does. Otherwise, it is easy to forget what is going on in a program. By well documenting your program with REM statements, you can remind yourself how the program works. REM statements will make your program much more understandable to other people.

REM statements are also used to title programs.

REM can be abbreviated to the apostrophe.

SYNTAX

REM comment.....
' comment .....

EXAMPLES

```
10 REM MY VERY FIRST PROGRAM
20 REM TITLE   : SPACE ZAP
30 REM DATE    : 31/6/1984
40 REM VERSION : 1
50 REM
.
.
rest of the program
```

POINTS TO REMEMBER

REM can be added at the end of a line.
(:REM ..)
Do not do this in DATA statements, however, because the computer will consider it to be more data.

REM statements make the program easier to understand, but they also use up precious memory. It is a good idea to use REM statements until you start running out of memory, then DELETE them as necessary

Never GOTO or GOSUB to a REM statement. If you delete a REM

statement which is referred to in a GOTO or a GOSUB, an undefined line number error will result.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 3: WRITING A PROGRAM

RENUM RENUMber

DESCRIPTION

This command renumbers the program lines. Quite often you will find your program in bit of a mess because the line numbers are not in a fixed increment. Use RENUM: It is the best cure for improving the listing of a program and makes inserting new lines easier.

RENUM on its own will renumber from 10 in increments of 10: this is the most widely used line numbering scheme, but you can renumber from a specific line with any increment.

RENUM will automatically change the line numbers associated with GOTO, GOSUB, RESTORE, THEN, ELSE, ON GOTO, ON GOSUB.

SYNTAX

RENUM
Renumbers from 10 in increments of 10.

RENUM <line 1>
Renumbers from new line number 1 in increments of 10.

RENUM <line 1>,<line 2>
Renumbers from old line number 2, starting with new line number 1, in increments of 10.

RENUM <line 1>,<line 2>,<increment>
Renumbers starting with old line number 2, starting with new line number 1, with the given increment.

RENUM ,<line 2>,<increment>
Renumbers from old line number 2, starting with new line number 10, with the given increment.

RENUM ,<line 2>
Renumbers from old line number 2, starting with new line number 10 in increment of 10.

RENUM ,,<increment>
Renumbers from old line number 10 starting with new line number 10, in given increment.

RENUM <line 1>,,<increment>
Renumbers starting from old line number 10, starting with new line number 1, in given increment.

EXAMPLES

RENUM ,,100
Renumbers from 10 with an increment of 100.
RENUM 1000,200,20

Renumbers from old line number 200, starting with new line number 1000, in increment of 20.

```
10 REM
20 REM
30 REM
40 REM
50 REM
```

```
RENUM ,10,20
OK.
LIST
10 REM
30 REM
50 REM
70 REM
90 REM
```

BUG HUNT

If a non-existent line number appears after GOTO or GOSUB or any other statement associated with having a line number after it, this will cause an "Undefined line NNNN in MMMM" error.
E.g.
10 GOTO 97
RENUM
Undefined line 97 in 10

Careless line renumbering such as RENUM 10,40 when the old lines have numbers 10,20,30 will not work. The result is an illegal function call error.

Negative increments are also not allowed.

You cannot have line numbers greater than 65535 so, if during the course of renumbering the line numbers go above this value, then an illegal function call error will result.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 7: USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS
Section 2 Chapter 1 : ADVANCED PROGRAM EDITING

RESTORE

DESCRIPTION

RESTORE DATA statements. When the computer READs DATA statements, it starts from the DATA statement with the smallest line number and proceeds to increasing line numbers until it runs out of DATA. To make the computer reREAD data from a specific line, you can point to the line number which contains the DATA statement you want by using the RESTORE statement.

After the RESTORE statement, the next READ statement will read the first data from the line pointed to by RESTORE. If there is no line number specified, then the computer will read from the very first DATA statement it finds in the program.

SYNTAX

RESTORE
RESTORE <line>

EXAMPLES

```
10 READ A$
20 PRINT A$
30 RESTORE
40 READ B$
50 PRINT B$
60 DATA "GOLDEN YEARS"
```

```
RUN
GOLDEN YEARS
GOLDEN YEARS
OK.
```

POINTS TO REMEMBER

Use RESTORE to prevent an out of data error.

BUG HUNT

If RESTORE points to a non-existent line number, an Undefined line number error will result.

ASSOCIATED KEYWORDS AND REFERENCE

READ
DATA
Section 1 Chapter 12: READING DATA INTO ARRAYS

RESUME

DESCRIPTION

RESUME means resume BASIC operation after an error trapping procedure has taken place, using the error trapping statement ON ERROR GOTO. After the error has been dealt with, RESUME tells the computer to continue the execution from wherever you want.

If there is no RESUME inside an error trapping routine, it will cause No RESUME error unless you halt the program using STOP or END statements within the error trapping routine.

There are three forms of RESUME.

SYNTAX

RESUME or RESUME 0
Resumes execution from the statement which caused the error.
RESUME NEXT
Resumes execution from the statement following the one which caused the error.
RESUME <line>
Resumes execution from the specified line number.

EXAMPLES

In this routine all commands inputted with KILL in them will be treated as a new error (No. 255) and are dealt with in the error trapping routine using the ERR function. RESUME 20 tells the computer to resume operation at line 20 after the warning message is displayed.

```
10 ON ERROR GOTO 1000
20 INPUT "MY LORD, WHAT IS THY COMMAND";
   A$
30 IF INSTR(A$,"KILL") THEN ERROR 255
40 PRINT "OK.":END
..
..
1000 IF ERR=255 THEN PRINT"KILLING
IS ILLEGAL IN THIS ADVENTURE.":RESUME 20
1010 END
```

```
RUN
MY LORD, WHAT IS THY COMMAND? KILL HIM
KILLING IS ILLEGAL IN THIS ADVENTURE
MY LORD, WHAT IS THY COMMAND? GOTO EAST
OK
```

POINTS TO REMEMBER

You cannot use RESUME to go back into BASIC from the command mode.

BUG HUNT

A RESUME statement that is not in an error trapping subroutine causes a RESUME without error.

An Undefined line number error results if RESUME refers to a non-existent line.

ASSOCIATED KEYWORDS AND REFERENCES

ON ERROR GOTO
ERL
ERR
ERROR
Section 2 Chapter 10. ERROR HANDLING

RETURN

DESCRIPTION

RETURN from GOSUB subroutine.

This makes the computer jump to the statement after the most recent GOSUB statement following execution of the specified subroutine.

SYNTAX

RETURN

EXAMPLES

```
10 PRINT 1000
20 GOSUB 50
30 PRINT 2000
40 STOP
50 PRINT "JUMP TO SUBROUTINE"
60 RETURN
```

```
RUN
1000
JUMP TO SUBROUTINE
2000
BREAK IN 40
```

POINTS TO REMEMBER

It is very easy to crash into a subroutine. It is therefore a good idea to make the subroutine distinguishable and have an END or STOP statement at the end of the main program to prevent accidental entry.

If the logic of your program requires it, it is perfectly possible to have more than one RETURN statement in any one subroutine.

RETURN is also used in various event handling subroutines.

BUG HUNT

A RETURN without GOSUB error results when the computer encounters an unexpected RETURN statement without a prior GOSUB statement.

ASSOCIATED KEYWORDS AND REFERENCE

GOSUB
ON .. GOSUB
ON ERROR GOSUB
ON INTERVAL GOSUB
ON KEY GOSUB
ON SPRITE GOSUB
ON STOP GOSUB
ON STRIG GOSUB
Section 1 Chapter 17: STRUCTURING YOUR PROGRAMS

RIGHT$

DESCRIPTION

This is one of the string manipulation functions. RIGHT$(A$,n) returns the rightmost n characters of string A$. If n is the same as the length of the A$, then quite obviously you will get the entire string A$ back. If n is zero, then a null string is returned.

SYNTAX

RIGHT$(<string-exp>,<integer-numeric>)

EXAMPLES

```
10 A$="GROUND CONTROL TO MAJOR TOM"
20 PRINT RIGHT$(A$,9)
30 B$="COMMENCING COUNT DOWN, ENGINE'S
   ON"
40 C=11
50 D$=RIGHT$(B$,C)
60 PRINT D$
```

```
RUN
MAJOR TOM
ENGINE'S ON
```

BUG HUNT

RIGHT$'s argument is a string followed by a number. If these are in reverse order, you will get a Type Mismatch error.

ASSOCIATED KEYWORDS AND REFERENCE

LEFT$
MID$
INSTR
Section 1 Chapter 14: MANIPULATING STRINGS

RND RaNDom number

DESCRIPTION

The RND function generates a random number between 0 and 1. The same series of random numbers will be generated each time the program is RUN. To generate truely random numbers which do not follow this series, use RND(-TIME)

SYNTAX

RND(<numeric>)

If <numeric> is negative, it makes the random number generator give a different series of random numbers, according to the number given.

If <numeric> is 0, then it repeats the last number given.

If <numeric> is positive, the next random number in the sequence is generated.

EXAMPLES

```
10 FOR I= 1 TO 5
20 PRINT RND(1)
30 NEXT I
```

```
RUN
 .59521943994623
 .10658628050158
 .76597651772822
 .57756392935958
 .73474759503023
Ok
```

Note: this program generates the same series of random numbers every time it is RUN.

To generate a random integer between 0 and 9, use this function.

```
X=INT(RND(1)*10)
```

POINTS TO REMEMBER

RND generates 14 digits, double precision numbers, between 0 and 0.99999999999999.

RND(-TIME) gives a truely random number, whereas RDN(1) gives the same series of random numbers.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 15: FUNCTIONS

RUN

DESCRIPTION

RUNs a program.

SYNTAX

RUN
RUN <line> .... RUN a program from the given line number.

EXAMPLES

```
10 PRINT 10000
20 A=300
30 B=200
40 PRINT A+B
```

```
RUN
10000
500
Ok.
```

```
RUN 20
500
Ok.
```

POINTS TO REMEMBER

How to end a program. END or STOP statements within the program will stop the execution.

<CTRL><STOP> terminates the execution from the keyboard.

On switching on the computer, the function key F4 is programmed as RUN<carriage return>.

BUG HUNT

If there is no BASIC program in the memory, RUN will result in "Ok." being displayed.

ASSOCIATED KEYWORDS AND REFERENCE

END
STOP
Section 1Chapter 3: WRITING A PROGRAM
Section 1 Chapter 7: USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

SAVE

DESCRIPTION

SAVEs a BASIC program to the specified device, in an ASCII file.
SAVE is used in merging two programs. See MERGE and Cassette Operation Section in Advanced Programming Guide.

SYNTAX

SAVE "<program name>"
SAVE "<device-descriptor>[<file name>]"

<device-descriptor> = CAS: for cassette. No other device is supported in the current version of MSX BASIC.

EXAMPLES

Let us say that the program below is in the computer's memory.

```
10 PRINT"HELLO"
20 PRINT"WELCOME"
30 PRINT"TO"
50 FOR I =1 TO 10
60 PRINT"MSX"
70 NEXT I
```

To save it in an ASCII format with program name "PROG" use:
SAVE "PROG" or SAVE "CAS:PROG"
To load it back in use:
LOAD "PROG" or LOAD "CAS:PROG"

POINTS TO REMEMBER

This is different from CSAVE because SAVE saves the BASIC program in ASCII file, whereas CSAVE saves in a tokenised format which is quite different.

ASSOCIATED KEYWORDS AND REFERENCE

MERGE
LOAD
Section 2 Chapter 11: CASSETTE SAVING AND LOADING

SCREEN

DESCRIPTION

SCREEN has a number of functions. It is used to set the various modes of operation for the display, cassette, keyboard, sprite and printer. It is mainly used to set the screen mode.

Using the SCREEN command, you may specify the sprite size but you can use only one size at a time.

Other options set by the SCREEN command are the key click switch cassette baud rate, and printer selection.

SYNTAX

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

	Examples
<mode> = 0,1,2,3	
0 = 40 x 24 text mode	SCREEN 0
1 = 32 x 24 text mode	SCREEN 1
2 = high resolution mode	SCREEN 2
3 = multi colour mode	SCREEN 3
<sprite size> = 0,1,2,3	
0 = 8 x 8 unmagnified	SCREEN ,0
1 = 8 x 8 magnified	SCREEN ,1
2 = 16 x 16 unmagnified	SCREEN ,2
3 = 16 x 16 magnified	SCREEN ,3
Note: when the sprite size is specified the contents of SPRITE$ will be cleared.	
<key click switch> = 0, non zero	
0 = disable the key click	SCREEN ,,0
non zero = enable the key click	SCREEN ,,1
<cassette baud rate> = 0,1	
0 = 1200 baud	SCREEN ,,,0
1 = 2400 baud	SCREEN ,,,1
Note: The baud rate can be changed using CSAVE statement as well.	
<printer option> = 0, non zero	
0 = printer with MSX printer which has MSX graphics capability.	SCREEN ,,,,0

non zero = non MSX standard printer, SCREEN ,,,,1
all graphics are changed to spaces.

EXAMPLES

This example shows you how the graphics MODE 2 (Hi-res) and 3 (low-res) differ. The program carries out the same task in each mode, showing you the advantages and disadvantages of each.

```
10 SCREEN 2
20 GOSUB 90
30 PAINT (105,120)
40 FOR I=1 TO 1000:NEXT
50 SCREEN 3
60 GOSUB 90
70 PAINT (105,120),8,15
80 GOTO 80
90 PSET (100,100)
100 DRAW "R50D50L50U50F50"
110 RETURN
```

LINE 10 HIGH RESOLUTION GRAPHICS MODE
LINE 20 GOTO SUBROUTINE
LINE 30 PAINTS WITH CURRENT FOREGROUND COLOUR
LINE 40 DELAY
LINE 50 MULTI COLOUR GRAPHICS MODE
LINE 60 GOTO SUBROUTINE
LINE 70 PAINTS WITH RED THE AREA SURROUNDED BY LINE
LINE 90 SUBROUTINE: POSITIONS GRAPHICS CURSOR
LINE 100 DRAWS A BOX WITH A DIAGONAL LINE
LINE 110 RETURN

POINTS TO REMEMBER

Characteristics of each screen mode are given in the Advanced computer Graphics section.

BUG HUNT

Using INPUT in screen modes 2 or 3 will force you back into the last text mode used.

Use of any graphics statement in a text mode, except PUT SPRITE in mode 1, results in an Illegal function call.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 8: MORE ON PRINT AND THE SCREEN
Section 1 Chapter 21: THE SCREEN MODES
Section 2 Chapter 11: CASSETTE SAVING AND LOADING
Section 2 Chapter 12: ADAVANCED GRAPHICS I:
CHARACTERISTICS OF EACH SCREEN MODE
Section 2 Chapter 15: ADAVANCED GRAPHICS II:
SPRITE GRAPHICS
Section 2 Chapter 23: PERIPHERAL DEVICES

SGN SiGN

DESCRIPTION

SGN returns the sign of a given number.

− 1 if argument is negative
0 if argument is zero
1 if argument is positive

SYNTAX

SGN(<numeric>)

EXAMPLES

PRINT SGN(− 1000)
− 1

ASSOCIATED KEYWORDS AND REFERENCE

ABS
Section 1 Chapter 15: FUNCTIONS

SIN SINe

DESCRIPTION

SIN returns the sine of the argument in radians.

SYNTAX

SIN(<numeric>)

EXAMPLES

```
PRINT SIN(1)
   .84147098480792
```

POINTS TO REMEMBER

SIN calculates in double precision, and gives up to 14 significant digits.

ASSOCIATED KEYWORDS AND REFERENCE

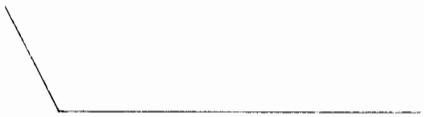
COS
TAN
ATN
Section 1 Chapter 19: MATHEMATICAL FUNCTIONS

SOUND

DESCRIPTION

SOUND writes directly to the Programmable Sound Generator Chip. PSG has 14 sound registers. For a step by step explanation of the SOUND statement, see the advanced sound effects section.

Register	Range	Description
0	0 – 255	Fine tune frequency on channel A
1	0 – 15	Coarse tune channel A
2	0 – 255	Fine tune frequency on channel B
3	0 – 15	Coarse tune channel B
4	0 – 255	Fine tune frequency on channel C
5	0 – 15	Coarse tune channel C
6	0 – 31	Frequency of the noise generator
7	0 – 63	Mixer (each bit has its unique control)
	bit 0	Tone on channel A 0=ON 1=OFF
	bit 1	Tone on channel B 0=ON 1=OFF
	bit 2	Tone on channel C 0=ON 1=OFF
	bit 3	Noise on channel A 0=ON 1=OFF
	bit 4	Noise on channel B 0=ON 1=OFF
	bit 5	Noise on channel C 0=ON 1=OFF
8	0 – 16	Volume control for channel A (uses envelope when 16)
9	0 – 16	Volume control for channel B (uses envelope when 16)
10	0 – 16	Volume control for channel C (uses envelope when 16)
11	0 – 255	Envelope period (low)
12	0 – 255	Envelope period (high)
	Envelope period (0 – 65535) = R12*256+R11	

REGISTER	ENVELOPE NO.	ENVELOPE PATTERN
13	0 – 15	
	0,1,2,3,9	
	4,5,6,7,15	

8

10

11

12

13

14

SYNTAX

SOUND <register>,<data>

EXAMPLES

Special sound effect.

```
10 SOUND 7,62
20 SOUND 1,0
30 SOUND 0,254
40 SOUND 8,16
50 SOUND 13,9
60 SOUND 12,60
70 SOUND 11,0
```

BUG HUNT

If you cannot hear any sound, you have either got the syntax wrong, or the volume control on your television is off.

ASSOCIATED KEYWORDS AND REFERENCE

PLAY
Section 2 Chapter 19 : ADVANCED SOUND EFFECTS USING PSG.

SPACE$ SPACE string

DESCRIPTION

SPACE$ gives a string of <space>s of a given number. The argument can be real but must be within the range of 0 to 255. Fractional portions of a real number are discarded.

SYNTAX

SPACE$(<numeric>)
range 0 to 255.

EXAMPLES

```
10 A$="MSX"+SPACE$(5)+"COMPUTER"
20 PRINT A$
```

```
RUN
MSX     COMPUTER
```

POINTS TO REMEMBER

There is a similar function SPC which also prints blank spaces but this can only be used in PRINT and LPRINT statements.

BUG HUNT

A Type Mismatch error occurs when equated to a numerical variable.

ASSOCIATED KEYWORDS AND REFERENCE

SPC
Section 1 Chapter 9 : MORE ON PRINT AND THE SCREEN

SPC SPaCe

DESCRIPTION

This prints spaces on screen. It can only be used in PRINT and LPRINT statements. It is used mainly to save memory, when you have to incorporate a lot of blank spaces in a PRINT statement.

SYNTAX

SPC(<numeric>)
range 0 to 255

EXAMPLES

```
10 A$="MSX":B$="COMPUTER"
20 PRINT A$;SPC(10);B$
```

```
RUN
MSX          COMPUTER
```

POINTS TO REMEMBER

This is different from SPACE$ because it cannot be treated as a string variable.

BUG HUNT

An Illegal function call results if the argument exceeds 255.

ASSOCIATED KEYWORDS AND REFERENCE

PRINT
LPRINT
SPACE$
Section 1 Chapter 9 : MORE ON PRINT AND THE SCREEN

SPRITE ON/OFF/STOP

DESCRIPTION

This is the sprite graphics event handling statement. It activates (ON) or deactivates (OFF or STOP) the trapping of sprite collisions in a BASIC program.

A SPRITE ON must be executed to activate sprite collision detection and make the computer jump to the subroutine specified in the ON SPRITE GOSUB statement. After the SPRITE ON statement, the computer checks for any sprite collision every time the BASIC starts a new statement. When a collision is detected, the computer will immediately jump to the subroutine.

If a SPRITE STOP is executed, the computer will stop sprite collision detection, but it will remember if there was a collision while SPRITE STOP was acting, and as soon as SPRITE ON is executed it will make the computer jump to the sprite subroutine.

SPRITE OFF will completely stop sprite trapping. No collision will be remembered.

SYNTAX

SPRITE ON
SPRITE OFF
SPRITE STOP

EXAMPLES

In this example, you will see two square sprites, one yellow and another white, approaching each other from opposite sides of the screen. When they collide, ON SPRITE GOSUB will come in to effect and make the BASIC jump to the sprite interrupt subroutine. The SPRITE OFF in the sprite interrupt routine prevents any further detection of sprite collisions. (Try this routine without SPRITE OFF and see what happens.)

```
10 ON SPRITE GOSUB 110
20 SCREEN 2,0
30 SPRITE$(0)=STRING$(8,CHR$(255))
40 SPRITE$(1)=STRING$(8,CHR$(255))
50 SPRITE ON
60 FOR I=10 TO 240
70 PUT SPRITE 0,(I,100),11,0
80 PUT SPRITE 1,(250-I,100),15,1
90 NEXT I
100 END
105 REM SPRITE COLLISION ROUTINE
110 SPRITE OFF
120 BEEP
130 RETURN
```

LINE 10 SETS TRAP SUBROUTINE TO 110
LINE 20 SETS GRAPHICS SCREEN
LINE 30 SPRITE 0 IS A SQUARE
LINE 40 SO IS SPRITE 1
LINE 50 SPRITE COLLISION DETECTOR ON
LINE 60 LOOP
LINE 70 SPRITE (YELLOW) MOVES FROM LEFT
LINE 80 SPRITE (WHITE) MOVES FROM RIGHT
LINE 90 NEXT LOOP
LINE 100 FINISH
LINE 110 KILLS TRAP (ONCE IS ENOUGH)
LINE 120 NOISE INDICATOR
LINE 130 GO BACK THE LOOP

POINTS TO REMEMBER

SPRITE interrupt is disabled when the program is not running and also during error trapping routines.

BUG HUNT

You must execute an ON SPRITE GOSUB statement before you use the SPRITE ON/OFF/STOP statement. Otherwise the computer will not detect any sprite collisions.

ASSOCIATED KEYWORDS AND REFERENCE

ON SPRITE GOSUB
SPRITE$
PUT SPRITE
Section 2 Chapter 15 : ADVANCED GRAPHICS IV : SPRITE GRAPHICS

SPRITE$

DESCRIPTION

You can define sprites using SPRITE$. You can have up to 256 sprite patterns when you are using sprite size 0 or 1 (unmagnified), or 64 of them in sprite size 2 or 3 (magnified).

The length of a sprite is fixed at 32 bytes but for small sprites you only need to define the first 8 characters. The rest are automatically defined as CHR$(0).

To define a 16 by 16 sprite, you must define all 32 bytes in the sprite pattern.

The details on how to use sprites are given fully in Advanced Programming Guide: Sprites.

SYNTAX

SPRITE$(<integer>)=<string-exp>

EXAMPLES

```
10 SCREEN 2
20 SPRITE$(0)=CHR$(16)+CHR$(48)
+CHR$(112)+CHR$(255)+CHR$(255)
+CHR$(112)+CHR$(48)+CHR$(16)
70 PUT SPRITE 0,(100,100),15,0
80 GOTO 80
```

LINE 10 HI-RES. SCREEN MODE 2
LINE 20 DEFINES SPRITE 0
LINE 70 PUT SPRITE 0 at x=100,y=100, COLOUR WHITE IN SPRITE PLA'N 0

Result: you see an arrow in the middle of the screen.

POINTS TO REMEMBER

You can only use sprites in screen mode 1, 2, and 3.

The sprite size is defined using the SCREEN statement. The SCREEN statement also kills all previously defined sprites.

You cannot mix two sprite sizes.

ASSOCIATED KEYWORDS AND REFERENCE

PUT SPRITE
SCREEN
SPRITE ON/OFF/STOP
Section 2 Chapter 15 : ADVANCED GRAPHICS IV : SPRITE GRAPHICS

SQR SQuare Root

DESCRIPTION

This is the square root function.

The argument must be positive. MSX SQR cannot handle complex numbers.

SYNTAX

SQR(<numeric>)

EXAMPLES

```
10 PRINT SQR(9)
20 PRINT SQR(ATN(1)*4)
```

```
RUN
 3
 1.7724538509055
```

POINTS TO REMEMBER

SQR is calculated to double precision.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 19 : MATHEMATICAL FUNCTIONS

STEP

DESCRIPTION

STEP is a part of FOR-TO-NEXT loop. STEP indicates how much the variable used in the FOR-TO is to be incremented at each iteration.

You can STEP in smaller steps than 1, or even have a negative STEP if you like. STEP 1 is not necessary, as all FOR-TO-NEXT loops without specified STEP are incremented by 1.

SYNTAX

FOR <num-var> = <numeric> TO <numeric> STEP <numeric>

EXAMPLES

```
10 FOR I = 1 TO 3 STEP 0.5
20 PRINT I;
30 NEXT I
40 PRINT
50 FOR J = 3 TO 1 STEP -1
60 PRINT J;
70 NEXT J
```

```
RUN
 1   1.5   2   2.5   3
 3   2   1
```

BUG HUNT

FOR I = 1 TO 10 STEP – 1
This is a common error. This loop will only loop once.

ASSOCIATED KEYWORDS AND REFERENCE

FOR
TO
NEXT
Section 1 Chapter 5 : THE USE OF LOOPS

STICK joySTICK

DESCRIPTION

STICK returns the direction of a joystick or the cursor keys when they are used as joystick.

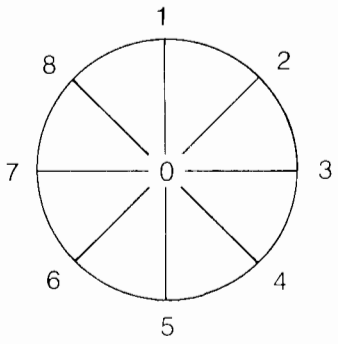
STICK(<n>) ... <n> can be either 0, 1, or 2.

n=0 cursor keys

n=1 joystick 1

n=2 joystick 2

When the joystick is in neutral, 0 is returned. Otherwise it is as follows:



You must press two cursor keys at once to get the diagonal direction, i.e press ↑ and → for direction 2.

SYNTAX

STICK(<n>)

EXAMPLES

This example displays the direction values when you are using cursor keys as a joystick.

```
10 A=STICK(0)
20 LOCATE 20,10: PRINT A
30 GOTO 10
```

POINTS TO REMEMBER

STICK always returns an integer between 0 and 8.

To find the status of the joystick trigger, use the STRIG function.

ASSOCIATED KEYWORDS AND REFERENCE

STRIG
Section 2 Chapter 23 : PERIPHERAL DEVICES.

STOP

DESCRIPTION

STOP terminates BASIC program execution and returns to command level.

The "Break in <line>" message is displayed after STOP termination.

SYNTAX

STOP

EXAMPLES

```
10 PRINT 1909
20 STOP
```

```
RUN
 1909
Break in 20
Ok
```

POINTS TO REMEMBER

The STOP statement does not CLOSE files at the end of the program, whereas END does.

Execution can be resumed from the next line using CONT.

You can use as many STOP statements as you wish.

This is different from STOP ON/OFF/STOP statements.

ASSOCIATED KEYWORDS AND REFERENCE

END

Section 1 Chapter 7 : USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

STOP ON/OFF/STOP

DESCRIPTION

This is quite different from the STOP statement, which terminates execution of a program. STOP ON/OFF/STOP enables/disables <CTRL><STOP> KEY trapping. When STOP ON is executed, the BASIC starts to check if the <CTRL><STOP> has been pressed every time it executes a new statement. If the a <CTRL><STOP> has been detected, then the BASIC is diverted to the subroutine specified by the ON STOP GOSUB statement executed earlier in the program.

If STOP STOP is executed, the computer will not jump to the subroutine specified if <CTRL><STOP> is pressed, but the computer will immediately jump to the subroutine when next STOP ON is executed.

STOP OFF will completely stop <CTRL><STOP> detection.

SYNTAX

STOP ON
STOP STOP
STOP OFF

EXAMPLES

This program shows you how ON STOP GOSUB can prevent a user from breaking into your program. Line 20 STOP ON enables the trapping. If you press <CTRL><STOP>, it will say "control stop disabled" and carries on executing the program. This routine has a special exit routine. Press "s" to disable <CTRL><STOP> trapping. Otherwise it will print out MSX indefinitely.

```
10 ON STOP GOSUB 100
20 STOP ON
30 IF INKEY$="s" THEN STOP OFF:PR
INT"CONTROL STOP Enabled"
40 PRINT "MSX"
50 GOTO 30
100 BEEP
110 PRINT"CONTROL STOP DISABLED"
120 RETURN
```

LINE 10 SPECIFIES THE LOCATION OF <CTRL><STOP> SUBROUTINE
LINE 20 ENABLES THE DETECTOR
LINE 30 DISABLES THE DETECTOR AND GIVES A MESSAGE
LINE 40 PRINTS MSX
LINE 50 LOOPS BACK TO 30
LINE 100 WARNING BEEP
LINE 110 MESSAGE
LINE 120 RETURNS TO WHEREVER THE <CTRL><STOP> WAS DETECTED

POINTS TO REMEMBER

Note that ON STOP trapping does not stop <STOP> key halt. It only prevents you from breaking into a program. That's all. Try pressing the <STOP> key only, in the above program. You will find that it will halt the program and display the cursor. If you press the <STOP> key for the second time, the program will continue.

Remember to execute STOP ON to enable the <CTRL><STOP> trapping.

The <CTRL><STOP> interrupt is disabled when the program is not running and also during error trapping routines.

BUG HUNT

You must have an ON STOP GOSUB statement and a <CTRL><STOP> handling subroutine.

ASSOCIATED KEYWORDS AND REFERENCE

ON STOP GOSUB
Section 2 Chapter 9 : EVENT HANDLING AND INTERRUPT BY BASIC

STR$

DESCRIPTION

STR$ converts the numeric argument into a string.

SYNTAX

STR$(<numeric>)

EXAMPLES

```
10 A$="PI="+STR$(3.14)
20 PRINT A$
```

```
RUN
PI=3.14
```

POINTS TO REMEMBER

The opposite function of STR$ is performed by VAL.

BUG HUNT

The argument must be numeric.

ASSOCIATED KEYWORDS AND REFERENCE

VAL
Section 1 Chapter 15 : FUNCTIONS
Section 2 Chapter 3 : TYPE CONVERSION

STRIG TRIGger Status

DESCRIPTION

This function returns the status of the trigger button of a joystick.

SYNTAX

STRIG(<n>)

<n>=0	Space bar used as trigger
<n>=1,3	Joystick 1
<n>=2,4	Joystick 2

STRIG(<n>)=0	not pressed
STRIG(<n>)=−1	pressed

EXAMPLES

If you press the space bar you will get a message:

```
10 IF STRIG(0) THEN PRINT "SPACE
BAR PRESSED"
20 GOTO 10
```

POINTS TO REMEMBER

STRIG is mainly used in arcade games.

ASSOCIATED KEYWORDS AND REFERENCE

STICK
Section 2 Chapter 23 : PERIPHERAL DEVICES

STRIG ON/OFF/STOP

DESCRIPTION

This is quite different from the STRIG statement, which returns the status of trigger buttons. STRIG(<n>) ON/OFF/STOP enables/disables trigger button trapping. When STRIG(<n>) ON is executed, the BASIC starts to check if the trigger button <n> has been pressed, every time it executes a new statement. If the trigger <n> has been detected, then the BASIC is diverted to the subroutine specified by the ON STRIG GOSUB statement executed earlier in the program.

If STRIG(<n>) STOP is executed, the computer will not jump to the subroutine specified when trigger <n> is pressed, but it will remember if the trigger <n> has been pressed and the computer will immediately jump to the subroutine when the next STRIG(<n>) ON is executed.

STRIG(<n>) OFF will completely stop <n> trigger detection.

SYNTAX

STRIG(<n>) ON
STRIG(<n>) STOP
STRIG(<n>) OFF

<n> is the trigger number.

There are five triggers:
0=Space bar
1=trigger of joystick 1
2=trigger of joystick 2
3=trigger of joystick 1
4=trigger of joystick 2

EXAMPLES

Here is a very short program which demonstrates how the trigger trapping works with the space bar as a trigger. When the space bar is pressed, the program jumps to the trigger trap routine. Otherwise it will print out MSX continuously until the <s> key is pressed.

```
10 ON STRIG GOSUB 100
20 STRIG(0) ON
30 IF INKEY$="s" THEN END
40 PRINT "MSX"
50 GOTO 30
90 REM SPACE TRIGGER ROUTINE
100 BEEP
110 PRINT "SPACE BAR PRESSED"
120 RETURN
```

LINE 10 SET TRIGGER ROUTINE TO LINE 100
LINE 20 SWITCH ON
LINE 30 IF - s - IS PRESSED THEN END
LINE 40 PRINT MSX
LINE 50 LOOP BACK TO LINE 30
LINE 100 WARNING BEEP
LINE 110 GIVES A MESSAGE
LINE 120 RETURN TO WHEREVER THE INTERRUPT WAS DETECTED

```
RUN
MSX
MSX
MSX
MSX
SPACE BAR PRESSED                    HIT <space>
MSX
MSX
MSX
ok                                        HIT <s>
```

POINTS TO REMEMBER

STRIG interrupt is disabled when the program is not running and also during error trapping routines.

BUG HUNT

Common mistake:
STRIG (0) ON ..... this is wrong. There should not be a space between STRIG and (0).

You get an Undefined line number error if you put the wrong line number in the ON STRIG GOSUB statement.

ASSOCIATED KEYWORDS AND REFERENCES

ON STRIG GOSUB
STRIG ON/OFF/STOP
Section 2 Chapter 9 : EVENT HANDLING AND INTERRUPT BY BASIC

STRING$

DESCRIPTION

This returns a string with a specified number of specified characters.

SYNTAX

STRING$(<n>,<ASCII number>)

This returns n characters, made up of the character given by the specified ASCII code.

STRING$(<length>, <string>)

This returns a string of n characters, all the characters being the first character of the specified string variable.

<n> can be a variable or a constant.

EXAMPLES

```
10 INPUT A
20 PRINT STRING$(A,"X")
30 GOTO 10
```

```
RUN
20
XXXXXXXXXXXXXXXXXXXX
10
XXXXXXXXXX
```

POINTS TO REMEMBER

The STRING$ is useful for defining a square sprite.
SPRITE$(0)=STRING$(8,CHR$(255))

BUG HUNT

An Out of String space error results if the length of string exceeds the size of string space, which is 200 characters at default.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 20 : THE ASCII CODES

SWAP

DESCRIPTION

This SWAPs the contents of two variables.

SYNTAX

SWAP <variable>,<variable>

EXAMPLES

```
10 A=100 :B=200
20 C$="MSX" :D$="ASCII"
30 SWAP A,B
40 SWAP C$,D$
50 PRINT A,B
60 PRINT C$,D$
```

```
RUN
200           100
ASCII         MSX
```

BUG HUNT

Type mismatch results if you try to SWAP different types of variables.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 6 : THE USE OF CONDITIONS

TAB

DESCRIPTION

TAB is always used in conjunction with PRINT or LPRINT statements and it tabulates the number of character positions specified, counting from the left hand side of screen.

SYNTAX

PRINTTAB(<numeric>)
LPRINTTAB(<numeric>)

EXAMPLES

```
10 B=100
20 PRINTTAB(5)"B="TAB(10)B
```

```
RUN
     B=      100
Ok
```

POINTS TO REMEMBER

<numeric> must be less than WIDTH – 1 or it will skip down to the next line.

If <numeric> is a real number, then the decimal point will be cut off to give an integer.

The LOCATE statement is far more flexible than TAB since you can specify x and y coordinates.

BUG HUNT

An Illegal function call occurs if the numeric argument exceeds 255.

ASSOCIATED KEYWORDS AND REFERENCE

PRINT
LPRINT
LOCATE
Section 1 Chapter 9 : MORE ON PRINT AND THE SCREEN

TAN TANgent

DESCRIPTION

This returns the tangent of the argument in radians. TAN is calculated in double precision.

SYNTAX

TAN(<numeric>)

EXAMPLES

```
PRINT TAN(0.5)
  .54630248984381
```

BUG HUNT

The argument must be a numeric. If it is a string, a Type mismatch error will occur.

ASSOCIATED KEYWORDS AND REFERENCE

ATN
COS
SIN
Section 1 Chapter 19 : MATHEMATICAL FUNCTIONS

THEN

DESCRIPTION

This is used with IF statements, which are often referred to as IF THEN statements. The statement after THEN is executed when the conditions after IF are satisfied.

If there is a line number after THEN, it is taken as an abbreviation for GOTO <line> number.

You may have several statements, separated by colons, after the word THEN if you want to make the computer carry out several actions if the conditions in IF are satisfied.

SYNTAX

IF <conditions> THEN <statements>
IF <conditions> THEN <statements> ELSE <statements>
IF <conditions> THEN <line>

EXAMPLES

```
10 INPUT X
20 IF X=10 THEN PRINT "YES"
30 GOTO 10
```

```
RUN
?10
YES
?
```

POINTS TO REMEMBER

THEN can be replaced by a GOTO statement only when you are jumping to another line.

IF x=1 GOTO 2000

ASSOCIATED KEYWORDS AND REFERENCE

IF
GOTO
ELSE
Section 1 Chapter 6 : THE USE OF CONDITIONS
Section 2 Chapter 7 : BOOLEAN II: THE IF/THEN/ELSE

TIME

DESCRIPTION

The TIME function returns the value of the system's internal timer. The timer is set to 0 when the machine is switched on, and is automatically incremented every time the Video Display Processor generates an interrupt. This happens 50 times per second.

You can assign any integer value to the timer using TIME function. You can also reset the timer from zero if you want.

SYNTAX

TIME

EXAMPLES

Digital watch

```
10 CLS
20 TIME=0
30 MIN%=TIME/3600
40 SEC%=TIME/60-MIN%*60
50 LOCATE 10,11:PRINT "MINUTE";MIN%
60 LOCATE 10,12:PRINT "SECOND";SEC%
70 GOTO 30
```

RESULT: You see a digital clock in the middle of your screen.

```
MINUTE 1
SECOND 34
```

POINTS TO REMEMBER

When the VDP's interrupt is disabled, the clock will stop. This happens when you use an external device such as a cassette for loading in programs.

TIME always returns a positive integer.

TIME is used in RND(-TIME) to truly randomise the random number generator.

ASSOCIATED KEYWORDS AND REFERENCE

RND

Section 1 Chapter 15 : FUNCTIONS

TO

DESCRIPTION

TO is a part of the FOR..TO..STEP..NEXT structure and is always used after FOR. The initial value is given before TO and the final terminating value of the loop is given after TO.

SYNTAX

FOR <num-var>=<numeric> TO <numeric>

FOR <num-var>=<numeric> TO <numeric> STEP <numeric>

EXAMPLES

```
10 FOR I=1 TO 2
20 PRINT I
30 NEXT I
```

```
RUN
 1
 2
OK
```

POINTS TO REMEMBER

You do not really need a space between TO and the numeric values but it is easier to read if you use spaces, e.g.

FOR I=100TO200

ASSOCIATED KEYWORDS AND REFERENCE

FOR
STEP
NEXT
Section 1 Chapter 5 : THE USE OF LOOPS

TROFF TRace OFF

DESCRIPTION

Stops TRON, or stops tracing.

TROFF must be executed to stop tracing. There is no other way of getting out of TRON other than using TROFF.

SYNTAX

TROFF

POINTS TO REMEMBER

To use the tracing facility, use TRON.

ASSOCIATED KEYWORDS AND REFERENCE

TRON

Section 1 Chapter 7 : USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

TRON TRace ON

DESCRIPTION

TRON makes the computer print out the line number of the line it is currently executing in a program.

TRON is only used for debugging. You resort to this only when your program has become a 'spaghetti' program and you cannot follow it in any other way!

SYNTAX

TRON

EXAMPLES

```
10 FOR I=1 TO 2
20 PRINT I
30 NEXT I
```

```
TRON
OK
RUN
[10][20] 1
[30][20] 2
[30]
OK
```

POINTS TO REMEMBER

TRON causes confusion on the screen by printing too many line numbers. Also, it does not work in graphics modes.

To switch TRON off, use TROFF.

BUG HUNT

This is used for bug hunting.

ASSOCIATED KEYWORDS AND REFERENCE

TROFF

Section 1 Chapter 7 : USEFUL COMMANDS AND HINTS FOR WRITING PROGRAMS

USR

DESCRIPTION

USR calls the user's machine code subroutine from BASIC. The machine code subroutine jump position is defined using the DEFUSR statement.

You can pass to/from machine code data using this function.

SYNTAX

USR[<digit>](<argument>)
<digit> =0 to 9
<argument> (any type) to be passed to the machine code routine.

EXAMPLES

```
PRINT USR0("QWERTY")
DMY=USR4(B%)
X=USR9(1000)
```

POINTS TO REMEMBER

It is not possible to explain how to use machine code on this section, so go to the section on how to use the USR function for details.

BUG HUNT

When you are using your own machine code routine, do be careful. Always CSAVE your program before execution, because if the machine crashes due to a fault in the machine code subroutine, there is little or no chance of recovering the lost program.

ASSOCIATED KEYWORDS AND REFERENCE

DEFUSR
Section 2 Chapter 21 : USR FUNCTION AND MACHINE CODE

VAL VALue

DESCRIPTION

VAL returns the value of a string which contains a number. If a string is a string representation of a number, then it can be VALed. However, it cannot work on an expression within a string.

VAL will ignore spaces, tabs, and line feeds in front of a number in a string (see Examples). It cannot, however, cope with other characters in front of a number, e.g. VAL ("ZXY100") will not give 100.

VAL can recognise plus and minus signs.

SYNTAX

VAL(<string>)

EXAMPLES

```
PRINT VAL("      - 100")
- 100
PRINT VAL("GARBAGE1000")
0
A$="+10":PRINT VAL(A$)
10
```

POINTS TO REMEMBER

The opposite function of VAL is performed by STR$.

BUG HUNT

VAL ("- 100+900") will only return - 100, since VAL cannot work out expressions within a string.

VAL(A%) will cause a type mismatch error.

ASSOCIATED KEYWORDS AND REFERENCE

STR$
Section 1 Chapter 15 : FUNCTIONS
Section 2 Chapter 3 : TYPE CONVERSION

VARPTR VARiable PoinTeR

DESCRIPTION

VARPTR (<variable name>) returns the address of the first byte of the data identified with the given variable name. The variable name can be of any type but obviously that variable has to exist before you use VARPTR.

It can also return the address of the start of a file control block.

The address returned will be an integer, in the range – 32768 to 32767. If a negative number is given, add 65536 to get the actual address.

SYNTAX

VARPTR(<variable name>)
VARPTR(#<file name>)

EXAMPLES

```
PRINT VARPTR(A(0))
S$="IIIII":PRINT 65536+VARPTR(S$)
D%=100:PRINT "H";HEX$(65536+VARPTR(D%))
```

POINTS TO REMEMBER

VARPTR is sometimes used to obtain the address of an array, so it may be passed to a machine code subroutine. A function call of the form of VARPTR(A(0)) is usually specified when passing an array, so that the lowest address of the array is returned.

BUG HUNT

If the variable referred to in the argument does not exist, then an Illegal function call error will result.

ASSOCIATED KEYWORDS AND REFERENCE

Section 2 Chapter 20 : MEMORY MAP

VDP Video Display Processor register

DESCRIPTION

VDP is used to access the Video Display Processor.

There are 9 registers.

Ø to 7 are write only, so you can assign values to these.
8 is read only.

The details of VDP registers are given in detail in the "Graphics section: How to access the Video Display Processor".

SYNTAX

For registers Ø to 7, <digit> = Ø to 7
VDP(<digit>)=<numeric>

Register 8
<num-var>=VDP(8)

EXAMPLES

```
PRINT "VDP STATUS REGISTER ";BIN$(VDP(8))
```

The above prints out the binary representation of the VDP status register 8.

POINTS TO REMEMBER

The VDP processor of U.K. version of MSX is TMS 9929A which is PAL T.V. system compatible.

BUG HUNT

If you misuse the VDP function you are likely to mess up the display, unless you know what you are doing. If the screen goes haywire, then switch off and start again.

ASSOCIATED KEYWORDS AND REFERENCE

Section 2 Chapter 16 : ADVANCED GRAPHICS V :
HOW TO ACCESS THE VIDEO DISPLAY PROCESSOR

VPEEK Video RAM PEEK

DESCRIPTION

VPEEK peeks the Video RAM.

SYNTAX

VPEEK(<address>)
<address> = 0 to 16383

EXAMPLES

PRINT VPEEK(100)
V%=VPEEK(999)

POINTS TO REMEMBER

It is advised that this function is used only by advanced users.

Since the Video RAM is quite separate from the main memory RAM, you need this special peek function to access it.

All MSX computers have 16K bytes of Video RAM.

VPEEK is the the complementary function to VPOKE.

BUG HUNT

An Illegal function call results if <address> is out of range.

ASSOCIATED KEYWORDS AND REFERENCE

VPOKE
Section 2 Chapter 17 : ADVANCED GRAPHICS VI : THE VIDEO RAM

VPOKE Video RAM POKE

DESCRIPTION

VPOKE pokes a byte into the video RAM. You must give the video RAM address, and the data byte to be poked in.

The value of the data must be between 0 and 255.

SYNTAX

VPOKE <address>,<byte>

<address> range 0 to 16383
<byte> range 0 to 255

EXAMPLES

VPOKE 998,255

POINTS TO REMEMBER

Do not use this unless you are sure you know what you are doing.

Since the Video RAM is quite separate from the main memory RAM you need this special poke function to access it.

All MSX computers have 16K bytes of Video RAM.

VPOKE is the the complementary statement to VPEEK.

BUG HUNT

If <address> is out of range :-> Overflow error results.
If <byte> is out of range :-> an Illegal function call results.

ASSOCIATED KEYWORDS AND REFERENCE

VPEEK
Section 2 Chapter 7 : ADVANCED GRAPHICS VI:
THE VIDEO RAM

WAIT

DESCRIPTION

WAIT suspends program execution, and waits until a specified input port develops a specific bit pattern.

SYNTAX

WAIT<port number>,<I>[,<J>]
Range:
<port number> = 0 to 255
<I> and <J> = 0 to 255

POINTS TO REMEMBER

The data read in at the port is XORed with J and ANDed with I. If the result is zero, the BASIC loops back and reads the data in at the port again. (If J is ommitted, then J=0)

If the result is non-zero, program execution starts again.

This statement accesses the I/O ports directly without going through BIOS. Therefore, it is very likely that if you use this statement in a program, the program will lose its MSX compatibility and become machine dependent.

MSX only uses ports between 0 and 255: anything above 255 is meaningless.

ASSOCIATED KEYWORDS AND REFERENCE

OUT
INP

WIDTH WIDTH of screen

DESCRIPTION

The WIDTH statement sets the width of the screen. It also clears the screen when executed.

screen mode	default	maximum
screen 0	37	40
screen 1	29	32

SYNTAX

WIDTH <numeric>

EXAMPLES

WIDTH 30
WIDTH 20

POINTS TO REMEMBER

The minimum width is 1.

BUG HUNT

An Illegal function call results when the argument is out of the allowed range.

ASSOCIATED KEYWORDS AND REFERENCE

Section 1 Chapter 21 : THE SCREEN MODES

XOR eXclusive OR

DESCRIPTION

XOR, EXCULSIVE OR is one of the logical operators which perform Boolean algebra.

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

Therefore, 100 XOR 50, can be calculated as follows:

	X	100	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
XOR	Y	50	0	0	0	0	0	0	0	0	1	1	0	0	1	0	
		86	0	0	0	0	0	0	0	0	1	0	1	0	1	1	0

SYNTAX

<num-var>=<numeric> XOR <numeric>

POINTS TO REMEMBER

The following relationship is true:
A = A XOR Y XOR Y

BUG HUNT

An Overflow error occurs if the argument is out of integer range.

ASSOCIATED KEYWORDS AND REFERENCE

AND OR EQV NOT IMP
Section 2 Chapter 6 : BOOLEAN ALGEBRA

SECTION 4

THE OPERATING SYSTEM

This section deals with the BIOS (Basic Input/Output System), explaining how to use the entry points, and giving a listing of the system RAM area. To understand this section of the book, the reader should have a knowledge of Z80 assembly language programming. No attempt is given to teach this.

Sections 4.1 to 4.7 describe the BIOS calls available on any MSX computer. Related calls are grouped together in each of these sections.The format of each BIOS entry point description has been standardised. The first line contains the name of the routine, followed by its entry point address in hexadecimal. The next line gives the Z80 instruction, which must be executed to call the BIOS routine. Following this is an explanation of the function of the call, and information on how to pass and receive parameters from BIOS routines. The next part contains a list of the Z80 registers and memory locations which may be altered as a consequence of calling the routine. The final part gives information regarding the state of interrupts and possible uses of the call.

Section 4.8 gives an explanation of hooks followed by an alphabetical listing of each hook, with its address and where it is called from.

The final section contains an alphabetical listing of the system RAM locations, with their addresses and lengths (in bytes).

Section 4.1

RST INSTRUCTIONS

This section deals with the eight instructions that can be executed using the Z80's RST instruction:

RST0	CHKRAM
RST1	SYNCHR
RST2	CHRGTR
RST3	OUTDO
RST4	DCOMPR
RST5	GETYPR
RST6	CALLF
RST7	KEYINT

CHKRAM 0000

Executed via RST 00

When the computer is first switched on, this is where execution commences. Jumping to this address by any means will cause a complete reset. The name CHKRAM is short for check the RAM, which is misleading because the routine does considerably more than that!

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

These are not applicable as the computer never returns from a call to this address.

REGISTERS AND MEMORY LOCATIONS ALTERED

All registers are altered and anything stored in memory is lost.

POSSIBLE USES

There is only one use for this 'routine' — to reset the computer.

SYNCHR 0008

Executed via RST1

This routine is used by the BASIC interpreter to check for syntax errors. If none are found, processing continues with call CHRGTR (0010). Otherwise a 'syntax error' is generated.

INPUT PARAMETERS

HL points to the next character of BASIC text, and the byte following the RST1 instruction contains the character with which it is to be compared.

OUTPUT PARAMETERS

On return, HL points to the next character and A has that character. The carry flag is set if the character is a number, and the zero flag is set if the end of the statement has been reached.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator, flags and HL register pair are affected by this call.

Notes

This call is of no use to machine code programmers and is best left alone.

CHRGTR 0010

Executed via RST2

This is used by the BASIC interpreter to fetch the next character or token from the BASIC program. It is normally called from routine SYNCHR (RST1).

INPUT PARAMETERS

HL must point to the character to be fetched.

OUTPUT PARAMETERS

On return, HL points to the next character of the program, the carry flag is set if a number is read, and the zero flag is set if the end of the current statement is encountered.

REGISTERS AND MEMORY LOCATIONS ALTERED.

AF and HL are altered by this RST.

Notes

As with SYNCHR (RST1), this routine is best left alone.

OUTDO 0018

Executed via RST3

This instruction writes the contents of the accumulator to the currently selected device.

INPUT PARAMETERS

The accumulator should contain the code to be written. If the code is to be written to a disk file, PTRFIL should contain the pointer to that file. If the data is to be written to the printer, then PTRFIL must contain zero and PRTFLG should be non-zero. If the code is to be sent to the terminal, then both PTRFIL and PTRFLG must be zero.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

No registers or memory locations should be affected by this call.

Notes

After pushing AF on to the stack, this routine calls the hook H.OUTD, before continuing execution.

DCOMPR 0020

Executed via RST4

This call compares the HL and DE register pairs of the Z80.

INPUT PARAMETERS

No input parameters are required, apart from the DE and HL register pairs.

OUTPUT PARAMETERS

On exit, the carry flat will be set if HL is less than DE, otherwise it will be cleared. Also, the zero flag will be set if HL = DE.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags are affected by this instruction.

GETYPR 0028

Executed via RST5

This instruction is used by the BASIC interpreter to find out which type of floating point accumulator it is using.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

Various flags are set according to the type of floating point accumulator currently in use.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the flags are affected by this routine.

CALLF 0030

Executed via RST6

RST6 performs an inter slot call.

INPUT PARAMETERS

The byte following the RST6 instruction contains the slot description byte, and the next two bytes contain the address to be called.

OUTPUT PARAMETERS

These depend on the routine called.

REGISTERS AND MEMORY LOCATIONS ALTERED

These depend on the routine call.

Notes

The slot description byte takes the following form:

Bit no.	7	6	5	4	3	2	1	0
Contents	F	x	x	x	S	S	P	P

Where:

PP (0-3) is the number of the primary slot to be selected,

SS (0-3) is the number of the secondary slot,

F (0-1) is a flag which is set to 1 if secondary slots are in use and 0 if they are not.

Bits 4-6 are not used and may be either set or unset.

KEYINT 0038

Executed via RST7 or an interrupt.

The MSX system operates in interrupt mode 1, so this is the RST instruction executed when a maskable interrupt occurs. Interrupts occur evey 0.02s due to the 50 Hz timer. When this happens, the interval count is decremented, JIFFY is incremented, the music queues are updated, the joy-stick trigger buttons are checked, and a keyboard scan is performed.

INPUT PARAMETERS

By its nature (i.e. hardware interrupt), there can be no input parameters to this routine.

OUTPUT PARAMETERS

Various 'traps' may be flagged (e.g. ON SPRITE), and another key may be placed in the keyboard buffer.

REGISTERS AND MEMORY LOCATIONS ALTERED

All registers remain unchanged, but many system memory locations are affected by this call.

Notes

The hook H.KEYI is called immediately after pushing all the registers on to the stack, enabling other interrupts to be processed.

Section 4.2

ENTRY POINTS ASSOCIATED WITH SLOT HANDLING

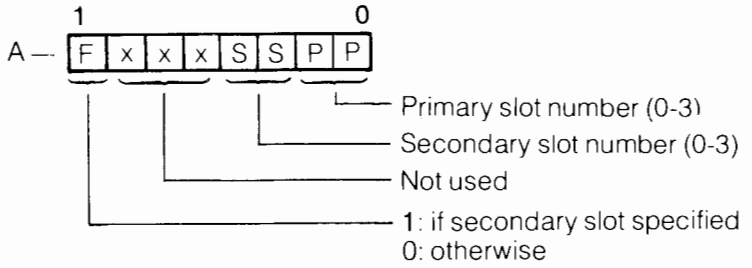
The BIOS entry points described in this section are used to manage the slot system (see section 2, chapter 22, MSX Memory Management and Cartridge Slot Mechanism for details).

RDSLT 000C

This call is used to read a memory location in any slot.

INPUT PARAMETERS

The HL register pair should contain the address of the location to be read and the accumulator should specify which slot to read from. The value of the accumulator has the following meaning: The two lowest bits contain the primary slot number (0-3) to be used, the next two bits contain the secondary slot number (0-3), the next three bits are not used and if a secondary slot has been specified then the top bit must be set otherwise it must be cleared.



OUTPUT PARAMETERS

The accumulator has the contents of the memory location specified.

REGISTERS & MEMORY LOCATIONS ALTERED

Registers AF, BC and DE are affected by this call.

NOTES

This routine disables all interrupts. An attempt to select page 3 of a slot, using this routine, will cause the computer to crash! (See ENASLT for a solution to this problem.)

WRSLT 0014

Executed via CALL &H0014

This call is used to write to any memory location in any slot.

INPUT PARAMETERS

The HL register pair should contain the address of the location to be written to; the accumulator should contain a number specifying which slot it is in, and register E, the number to be written. The contents of the accumulator take the same format as that described for entry point RDSLT (000C).

OUTPUT PARAMETERS

No parameters are returned from this routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC and D are affected by this call.

Notes

This routine disables all interrupts. An attempt to select page 3 of a slot, using this routine, will cause the computer to crash! (see ENASLT for a solution to this problem).

CALSLT 001C

Executed via CALL &H001C

This routine performs an inter-slot call, selecting a page in another slot and then calling an address there.

INPUT PARAMETERS

The IX register should contain the address which is to be called, and the most significant byte of the index register IY should specify the slot. The required format of the upper byte of IY is identical to the format of the accumulator for entry point RDSLT. Parameters can be passed in registers AF, BC, DE and HL, but not the alternative registers.

OUTPUT PARAMETERS

No parameters are returned unless they are created by the inter-slot call. Parameters may be returned in any of the registers except the alternative accumulator, as this holds the state of the slot select mechanism before the call.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF', BC', DE' and HL' are changed before the inter-slot call is executed.

Notes

Interrupts are disabled by this routine. This call can also be executed via the RST6 instruction (see CALLF, section 2.1). An attempt to select page 3 of a slot, using this routine, will cause the computer to crash! (see ENASLT for a solution to this problem).

ENASLT 0024

Executed via CALL &H0024

This call selects a page of one of the slots.

INPUT PARAMETERS

The most significant 2 bits of the H register are used to select the appropriate page as follows:—

MSB	Page selected
00	0000-3FFF
01	4000-7FFF
10	8000-BFFF
11	C000-FFFF

The accumulator should specify the slot to be selected, as explained under entry point RDSLT.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL are affected by this call.

Notes

This routine disables all interrupts.

The problem of not being able to select page 3 of a slot, in entry points RDSLT, WRSLT, and CALSLT, can be overcome by using this call. E.g. to read address &HD000 in slot 3, use the following code:

```
CALL   &H0138      ; READ THE PRIMARY SLOT SELECT
                     REGISTER
PUSH   AF          ; SAVE THE ABOVE
LD     HL,&HD000   ; THE MEMORY ADDRESS TO BE READ
PUSH   HL          ; SAVE THIS ADDRESS
LD     A,3
DI                 ; INTERRUPTS MUST BE DISABLED AS
                     THEY ALTER PAGE 3
CALL   &H0024      ; ENABLE SLOT 3, PAGE 3
POP    HL          ; RESTORE ADDRESS
LD     H,(HL)      ; GET DESIRED CONTENTS
POP    AF
CALL   &H013B      ; RE-ENABLE SYSTEM PAGE 3
EI
LD     A,H         ; RETURN DESIRED VALUE IN
                     ACCUMULATOR
RET
```

Similar methods can be used to write to and call a subroutine in page 3 of a different slot.

RSLREG 0138

Executed via CALL &H0138

This call reads the contents of the primary slot select register.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

A copy of the primary slot select register is returned in the accumulator.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator is changed by this call.

Notes

This routine is simply an IN instruction (IN &HA8 probably), followed by a RET.

WSLREG 013B

Executed via CALL &H013B

This call writes to the primary slot select register.

INPUT PARAMETERS

The accumulator should contain the value to be written.

OUTPUT PARAMETERS

No parameters are returned by this routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

No registers or memory locations are affected by this call.

Notes

This routine is simply an OUT instruction (OUT &HA8 probably), followed by a RET.

CALBAS 0159

Executed via CALL &H0159

This entry point is used by the BASIC interpreter to perform an inter-slot call into a BASIC interpreter expansion cartridge.

INPUT PARAMETERS

The address to be called is passed in the IX register.

OUTPUT PARAMETERS

The output from this call depends on the inter-slot call.

REGISTERS AND MEMORY LOCATIONS ALTERED

As above, this is not defined.

Section 4.3

BIOS ENTRY POINTS USED TO ACCESS THE CONSOLE

This section describes those calls which are used to control the 'console', i.e. the keyboard, text display and printer.

CHSNS 009C

Executed via CALL &H009C

This call performs two operations. Firstly it checks the status of the shift keys. If a shift key is being pressed and the function key display is active, then function keys 6-10 will be displayed. Secondly this routine checks the status of the keyboard buffer.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

If the keyboard buffer is empty the Z flag will be set, otherwise it will be cleared.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags are affected by this routine.

Notes

Interrupts are enabled by this call.

CHGET 009F

Executed via CALL &H009F

This routine returns a character from the keyboard buffer. If the buffer is empty the routine waits for a key to be pressed, displaying the cursor if enabled.

INPUT PARAMETERS

No input parameters are required by this routine.

OUTPUT PARAMETERS

The character code of the key pressed is contained in the accumulator.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags are affected by this call.

Notes

This routine calls hook H.CHGE immediately after pushing on to the stack registers HL, DE and BC (in that order). This allows other console input devices to be used.

CHPUT 00A2

Executed via CALL &H00A2

This call outputs a character to the console, moving to the next line and scrolling the screen where necessary. Control codes 7-13 and 27-31 inclusive are supported.

INPUT PARAMETERS

The accumulator should contain the character code to be sent to the console. The position of the cursor on the screen is stored in memory locations CSRX, CSRY.

OUTPUT PARAMETERS

On exit, the cursor position (CSRX,CSRY) will have been updated.

REGISTERS AND MEMORY LOCATIONS ALTERED

No registers are changed, but memory locations CSRX, CSRY and TTYPOS may be affected.

Notes

After pushing registers HL, DE, BC and AF on to the stack, this routine calls the hook H.CHPU, enabling other console output devices to be used (e.g. an 80 column screen). This routine does nothing if Graphics mode II or the Multicolour mode is active.

Interrupts are enabled by this call.

LPTOUT 00A5

Executed via CALL &H00A5

This routine sends a character to the printer, provided one is connected. If the printer is not ready to receive the character the routine will wait until it is ready or the stop key is pressed.

INPUT PARAMETERS

The character to be printed should be placed in the accumulator.

OUTPUT PARAMETERS

On return the carry flag will have been cleared if the character was successfully printed, and set if the stop key was used to abort the operation.

REGISTERS AND MEMORY LOCATIONS ALTERED

The flags may be altered by this call.

Notes

The first thing this routine does is to call the hook H.LPTO, enabling additional processing to be carried out. An example of this is to program the hook to ignore characters sent to the printer, via

```
H.LPTO:  INC     SP
         INC     SP
         RET
```

The hook H.LPTO is at &HFFB6, so the above example can be simply programmed using the BASIC statements:

```
POKE &HFFB6,&H33
POKE &HFFB7,&H33
```

(To get the printer working again type POKE &HFFB6,&HC9.)

LPTSTT 00A8

Executed via CALL &H00A8

This call checks to see whether the line printer is ready to receive a character.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

If the printer is ready to receive data the accumulator will contain 255 and the *Z* flag will have been cleared, otherwise the accumulator will contain zero and the *Z* flag set.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags are affected by this call.

Notes

LPTSTT is called by entry point LPTOUT.
The first thing this routine does is to call the hook H.LPTS.

CNVCHR 00AB

Executed via CALL &H00AB

This call converts a character code into the number of the pattern it represents on the video display processor screen. For character codes 32-255 these numbers are the same as the pattern number for that character, however character codes 0-31 are control codes, but pattern

numbers 0-31 represent the characters obtained by holding the graphics key down. These patterns are represented by two character codes — control code 1, followed by a character in the range 64-95.

INPUT PARAMETERS

The character to be converted should be placed in the accumulator.

OUTPUT PARAMETERS

Four different results can be obtained from this call, depending on the initial state of the graphic header byte (GRPHED) and accumulator.

1. If GRPHED contains zero and the accumulator contains character code 1, the accumulator remains unchanged, GRPHED is set to 1 and the carry and zero flags will be cleared.
2. If GRPHED contains zero and the accumulator has any value other than one, then GRPHED and the accumulator will not be changed and C and Z will be set.
3. If GRPHED contains a non-zero number and the accumulator contains a number in the range 64-95, on exit the accumulator will be 64 less than its initial value, GRPHED will contain zero, C will be set and Z cleared.
4. If GRPHED contains a non-zero number and the accumulator contains a number outside the range 64-95, GRPHED will be set to zero, the accumulator will not be changed and C and Z will be set.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator, flags and memory location GRPHED may be changed by this call.

PINLIN 00AE

Executed via CALL &H00AE

This routine is similar to INLIN, but when the computer is in the auto line numbering mode, pressing <CTRL-U> will not delete the line number as well.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

The HL register points to the memory location below the first character in the buffer and the carry flag is set if <CTRL><STOP> was pressed.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

INLIN 00B1

Executed via CALL &H00B1

This routine is used by the BASIC interpreter to accept a line from the keyboard, displaying the characters on the screen as they are entered and storing the line in a buffer, until <RETURN> or <CTRL><STOP> is pressed.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

The HL register points to the memory location below the first character in the buffer and the carry flag is set if <CTRL><STOP> was pressed.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

QUINLIN 00B4

Executed via CALL &H00B4

This routine is similar to INLIN, but it is used by the INPUT statement to print a question mark and then accept a line of input from the keyboard.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

The HL register points to the memory location below the first character in the buffer and the carry flag is set if <CTRL><STOP> was pressed.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

BREAKX 00B7

Executed via CALL &H00B7

This call checks to see if <CTRL><STOP> is being pressed.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

If <CTRL><STOP> is being pressed the carry flag will be set, otherwise it will be cleared.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator and flags are affected by this call.

Notes

This routine need only be used when interrupts are disabled, otherwise <CTRL><STOP> will be spotted by the 50 Hz interrupt.

ISCNTC 00BA

Executed via CALL &H00BA

This call checks whether the <STOP> key or <CTRL><STOP> is being pressed. If the <STOP> key is being pressed, the program loops until it is pressed again or <CTRL><STOP> is pressed. If <CTRL><STOP> is pressed and it has not been trapped, the computer goes through its break sequence, i.e. switching to text mode, enabling the slot containing the BASIC interpreter and jumping to BASIC command mode. If neither <STOP> nor <CTRL><STOP> is being pressed, this routine does nothing.

INPUT PARAMETERS

Input parameters are not required, but INTFLG will contain 3 if <CTRL><STOP> is being pressed and 4 if <STOP> is being pressed, due to the 50 Hz timer interrupt.

OUTPUT PARAMETERS

No parameters will be returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator, flags and memory locations INTFLG and KILBUF may be affected by this call.

Notes

The first thing this routine does is to inspect location BASROM. If this is zero the routine proceeds as above, otherwise it immediately returns. Therefore placing a non-zero number in BASROM will prevent BASIC programs being interrupted (use with care!).

Interrupts must be enabled if this routine is going to be used.

CKCNTC 00BD

Executed via CALL &H00BD

This routine does the same thing as ISCNTC, but is slightly slower. For this reason it is better to use ISCNTC.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator, flags and memory locations INTFLG and KILBUF may be affected by this call.

BEEP 0C00

Executed via CALL &H00C0

This call causes a BEEP as though <CTRL >< G> had been pressed.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

No output parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL and memory locations MUSICF, PLYCNT, VCBA to VCBA+4, VCBB to VCBB+4 and VCBC to VCBC+4 may be changed by this routine.

Notes

At the end of this routine a jump is made to BIOS entry point GICINI, resetting the sound generator.

CLS 00C3

Executed via CALL &H00C3

This call clears the currently selected screen (including the graphics modes).

INPUT PARAMETERS

If the Z flag is set the screen will be cleared, otherwise this call will do nothing.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and memory locations LINTTB, CSRX and CSRY may be altered by this call.

POSIT 00C6

Executed via CALL &H00C6

This routine moves the text cursor to a specified position.

INPUT PARAMETERS

The column of the desired position should be placed in H and the row in L.

OUTPUT PARAMETERS

CSRX and CSRY will be updated by this call.

REGISTERS AND MEMORY LOCATIONS ALTERED

The AF register and memory locations CSRX and CSRY will be changed by this call.

FNKSB 00C9

Executed via CALL &H00C9

This call checks to see whether the function key display is active and prints out the function key definitions if it is.

INPUT PARAMETERS

If CNSDFG contains zero, this routine does nothing, otherwise it falls into DSPFNK.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC and DE may be affected by this call.

Notes

Interrupts are enabled by this call.

ERAFNK 00CC

Executed via CALL &H00CC

This call deletes the function key display, provided the video display processor is in one of the text modes.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

No output parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC and DE may be altered by this call.

DSPFNK 00CF

Executed via CALL &H00CF

This call displays the function key definitions at the bottom of the screen.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

Memory location CNSDFG is set to 255, flagging that the function key display is active.

REGISTERS AND MEMORY LOCATIONS ALTERED

Register AF, BC, DE and memory location CNSDFG may be changed by this call.

Notes

Interrupts are enabled by this call.

TOTEXT . 00D2

Executed via CALL &H00D2

This routine forces the screen in to one of the text modes, provided it is not already in one.

INPUT PARAMETERS

The text mode which will be entered is stored in memory location OLDSCR.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

This call may change registers AF, BC, DE and HL, and memory locations LINLEN, NAMBAS, CGPBAS and SCRMOD.

Notes

Provided the VDP is not already is a text mode, this routine calls the hook H-TOTE with the contents of OLDSCR in the accumulator. After returning from the hook, control is passed to the BIOS entry point CHGMOD.

Interrupts are enabled during this routine.

CHGCAP 0132

Executed via CALL &H0132

This call controls the status of the <CAPS LOCK> lamp.

INPUT PARAMETERS

If the accumulator contains zero the <CAPS LOCK> lamp will be turned off, otherwise it will be turned on.

OUTPUT PARAMETERS

No parameters are returned from this routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags are affected by this call.

SNSMAT 0141

Executed via CALL &H0141

This call scans a row of the keyboard matrix, and returns the status of the keys in that row.

INPUT PARAMETERS

The number of the row to be scanned should be passed to the accumulator.

OUTPUT PARAMETERS

The status of the keys on that row is returned in the accumulator. If one of these keys is being pressed, the corresponding bit in the accumulator will contain zero, otherwise it will be set to 1.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags will be modified by this call.

Notes

Interrupts will be enabled by this call.

ISFLIO 014A

Executed via call &H014A

This call checks to see if any device I/O is being done.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

If no device I/O is being done the accumulator will be zero and the Z flag will be set otherwise the accumulator will be non-zero and the Z flag will be cleared.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and the flags are affected by this call.

OUTDLP 014D

Executed via CALL &H014D

This call is used by the BASIC interpreter to write a character to the printer.

INPUT PARAMETERS

The character to be printed should be passed to the accumulator.

OUTPUT PARAMETERS

No parameters are returned from this call, but if the operation is aborted, control is passed to the BASIC interpreter's error handling section, and a 'device I/O error' is reported.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the flags are affected by this call.

Notes

TAB characters are expanded to spaces by this routine.

KILBUF 0156

Executed via CALL &H0156

This call clears the keyboard buffer.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the HL register is changed by this call.

Section 4.4

BIOS ENTRY POINTS WHICH CONTROL THE JOYSTICK PORTS

The entry points described in the following pages give control over the joystick ports, and the devices which can be connected to them.

GTSTCK 00D5

Executed via CALL &H00D5

This call returns the status of the cursor keys or one of the joysticks.

INPUT PARAMETERS

The accumulator should contain the joystick indentifier. This is zero for the cursor keys, 1 for joystick 1, and 2 for joystick 2.

OUTPUT PARAMETERS

The direction of the selected joystick (or cursor keys) is returned in the accumulator. These positions are:–

0	for a centralised joystick,
1	for up,
2	for up and to the right,
3	for right,
4	for down and to the right,
5	for down,
6	for down and to the left,
7	for left,
8	for up and to the left.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

Notes

Interrupts are enabled by this call.

GTTRIG 00D8

Executed via CALL &H00D8

This call returns the current status of either the space bar, or one of the joystick trigger buttons.

INPUT PARAMETERS

The accumulator specifies which trigger button should be scanned, as shown below:-

0 = → the space bar
1 = → trigger 1A
2 = → trigger 2A
3 = → trigger 1B
4 = → trigger 2A

OUTPUT PARAMETERS

If the relevent trigger button is being pressed, the accumulator returns 255, otherwise it returns zero.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

Notes

Interrupts are enabled by this call.

GTPAD 00DB

Executed via CALL &H00DB

This call returns the status of a touch pad connected to one of the joystick ports.

INPUT PARAMETERS

The accumulator should contain a number in the range 0-7, depending on the required information. For A in the range 0-3, information about the touch pad connected to joystick port 1 is returned, and similar information is returned about a pad connected to port 2 if A is between 4 and 7. To merely find out if a touch pad is being pressed, use 0 or 4. To find the x-coordinate of a point being pressed, use 1 or 5, and use A=2 or 6 for the y-coordinate. To determine whether a touch pad switch is being pressed, pass 3 or 7 in the accumulator.

OUTPUT PARAMETERS

The output parameter is returned in the accumulator and depends on the input parameter in the following way. In the case of 0 or 4, 0 will be returned if the relevant touch pad is being pressed and if not, the accumulator will contain 255. The same is true for the state of a touch pad switch for an input parameter of 3 or 7. The x or y coordinate (in the range 0-255) of a pressed point, will be returned if the input parameter was 1 or 2 respectively (5 or 6 for joystick port 2).

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be affected by this call.

Notes

Interrupts are enabled by this call. See the BASIC keyword PAD for more information.

GTPDL 00DE

Executed via CALL &H00DE

This call returns the state of one of the 12 possible paddles connected to the joystick ports.

INPUT PARAMETERS

The paddle number should be passed in the accumulator. This is an odd number for a paddle connected to joystick port 1, and even for port 2.

OUTPUT PARAMETERS

A number in the range 0-255 is returned in the accumulator, specifying the state of the requested paddle.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

Notes

Interrupts are enabled by this call.

Section 4.5

BIOS CALL ASSOCIATED WITH THE CASSETTE INTERFACE

The calls described in this section are used to control the cassette interface and filing system.

TAPION 00E1

Executed via CALL &H00E1

This routine turns on the cassette motor and reads the header from the tape.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

The carry flag is set if the operation is aborted.

REGISTERS AND MEMORY LOCATIONS ALTERED

This routine may change registers AF, BC, DE and HL.

Notes

Interrupts are disabled while reading from the cassette.

TAPIN 00E4

Executed via CALL &H00E4

This call reads a byte from the cassette recorder.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

The data is returned in the accumulator. The carry flag is set if the operation was aborted.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

Notes

Interrupts are disabled while reading from the cassette.

TAPIOF 00E7

Executed via CALL &H00E7

This call stops the computer reading from tape.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

No registers or memory locations are affected.

TAPOON 00EA

Executed via CALL &H00EA

This call turns on the cassette motor and writes a header block to the cassette recorder.

INPUT PARAMETERS

The accumulator should contain zero if a short header is desired, otherwise a long header will be used.

OUTPUT PARAMETERS

The carry flag is set if the operation is aborted.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

Notes

Interrupts are enabled by this call.

TAPOUT 00ED

Executed via CALL &H00ED

This call writes a byte to the cassette recorder.

INPUT PARAMETERS

The accumulator should contain the byte to be written.

OUTPUT PARAMETERS

If the operation is aborted the carry flag will be set on exit.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL may be changed by this call.

Notes

Interrupts are enabled by this call.

TAPOOF 00F0

Executed via CALL &H00F0

This call stops the computer writing to the cassette recorder.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

No registers or memory locations are affected.

STMOTR 00F3

Executed via CALL &H00F3

This call turns the cassette recorder's motor on, off, or changes it to the opposite state.

INPUT PARAMETERS

If the accumulator contains 1, the cassette motor will be turned on; 0 and it will be turned off, and if the accumulator contains 255, the state of the cassette motor will be 'Flipped', i.e. If on, it will be turned off, and if off, turned on.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags are affected by this call.

Section 4.6

BIOS ENTRY POINTS DEALING WITH SOUND

The entry points in the following pages are used to control the programmable sound generator.

GICINI 0090

Executed via CALL &H0090

This call initialises the Programmable Sound Generator, flushing the sound queues and turning off any sounds being generated in so doing.

INPUT PARAMETERS

No input parameters are required, but interrupts must be disabled before calling this routine.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

All register contents are preserved, but memory locations MUSICF, PLYCNT, VCBA to VCBA+4, VCBB to VCBB+4 and VCBC to VCBC+4 are set to zero.

WRTPSG 0093

Executed via CALL &H0093

This call writes a value to one of the registers in the Programmable Sound Generator.

INPUT PARAMETERS

The accumulator should contain the number of the register to be written to, which must be in the range 0-13. The data to be written should be placed in E.

OUTPUT PARAMETERS

There are no output parameters.

REGISTERS AND MEMORY LOCATIONS ALTERED

No registers or memory locations are affected.

Notes

Interrupts are disabled at the beginning of this call, and enabled at the end.

RDPSG 0096

Executed via CALL &H0096

This call reads the contents of one of the registers in the Programmable Sound Generator.

INPUT PARAMETERS

The accumulator should contain the number of the register to be read, which must be in the range 0-13.

OUTPUT PARAMETERS

The contents of the register is returned in the accumulator.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator is affected by this call.

Notes

Interrupts are disabled at the beginning of this call, and enabled at the end.

STRTMS 0099

Executed via CALL &H0099

This routine starts the background music task if there is work queued for it.

INPUT PARAMETERS

No parameters need to be passed to this routine.

OUTPUT PARAMETERS

On exit the accumulator will contain zero if the sound buffer is empty.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, HL and memory locations PLYCNT and MUSICF may be altered by this call.

Notes

The background music task will not actually start until the next 50Hz interrupt.

Section 4.7

BIOS ENTRY POINTS ASSOCIATED WITH THE VDP

The bios entry points described in this section give complete control over the MSX computer's video display processor.

DISSCR 0041

Executed via CALL &H0041

Standing for disable screen, this call will cause the screen to blank out to the border colour. All output will be sent to the screen but it will not be visible until ENASCR (&H0044) is called. Changing mode will also re-enable the screen.

INPUT PARAMETERS

None.

OUTPUT PARAMETERS

None.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF and BC are modified by this call, and interrupts are enabled.

POSSIBLE USES

This call is of use in BASIC, via the USR function, to blank out the screen. It could be used to set up a picture with the screen disabled, then enabling the screen (ENASCR) would give the impression of instantaneous plotting. Example:

```
10 DEF USR0 = &H41
20 DEF USR1=&H44
30 CLS
40 REM DISABLE SCREEN
50 X=USR0 (0)
60 INPUT PW$
70 REM ENABLE SCREEN
80 X=USR1 (0)
```

ENASCR 0044

Executed via CALL &H0044

This address is called to enable the screen, after it has been disabled by DISSCR.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

No output is produced.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF and BC are modified by this routine, and interrupts are enabled.

POSSIBLE USES

Enabling the screen after DISSCR. See DISSCR for an example.

WRTVDP 0047

Executed via CALL &H0047

This call writes to a register of the video display processor. For details of the effects of writing to the VDP see Advanced Programming Guide, Chapter 16, Video Display Processor Section.

INPUT PARAMETERS

The VDP register number to be accessed should be specified in register C and the data to be written in B.

OUTPUT PARAMETERS

No output parameters are returned, but memory location RG0SAV+C contains the value initially specified in B.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF and BC, and memory location RGxSAV are modified by this call, where x is the register written to. E.g. if C contains 5 and B contains 0 then on exit from this routine memory location RG5SAV will contain 0. Also interrupts are enabled by this call.

POSSIBLE USES

This is a very powerful entry point, allowing control over the VDP. It is recommended that this routine be used for writing to VDP registers as copies of them are automatically kept. When a value is placed in a

register there is no way of telling what that value is, as these registers are 'write only'.

RDVDP 013E

Executed via CALL &H013E

This routine is used to read the video display processor's status register.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

The accumulator contains a copy of the VDP's status register on exit.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator is affected by this call.

Notes

The state of interrupts is not affected by this call.

RDVRM 004A

Executed via CALL &H004A

This call reads a memory location in the video RAM, as controlled by the video display processor.

INPUT PARAMETERS

The HL register pair should contain the address in the video RAM to be accessed.

OUTPUT PARAMETERS

On return, A contains the contents of the video memory pointed to by HL. Also, interrupts are enabled by this routine. Note: the state of the flags on exit from this routine do not reflect the contents of the accumulator.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only A and the flags are changed by this routine.

WRTVRM 004D

Executed via CALL &H004D

This call writes the contents of the accumulator to a location in the video RAM.

INPUT PARAMETERS

The HL register pair should contain the address in the video RAM to be written to, and A the value required.

OUTPUT PARAMETERS

None.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator and flags are changed by this routine and interrupts enabled.

POSSIBLE USES

Accessing the video RAM.

SETRD 0050

Executed via CALL &H0050

This call sets up the video display processor for a read operation.

INPUT PARAMETERS

HL should contain the address to be read.

OUTPUT PARAMETERS

None.

REGISTERS AND MEMORY LOCATIONS ALTERED

The A register and flags are affected, and interrupts enabled by this call.

POSSIBLE USES

Before reading the VDP RAM, the VDP must be initialised for a read operation. This routine is called by RDVRM and LDIRMV, so it is not necessary to use this entry point unless reading from the VDP directly is to be performed.

SETWRT 0053

Executed via CALL &H0053

This call sets up the video display processor for a write operation.

INPUT PARAMETERS

The HL register pair must contain the address in VDP RAM to be written to.

OUTPUT PARAMETERS

None.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator and flags are affected, and interrupts enabled by this routine.

POSSIBLE USES

As with SETRD this routine need never be used unless the user desires to access the VDP directly, since it is called by subroutines WRTVRM, FILVRM and LDIRVM.

FILVRM 0056

Executed via CALL &H0056

This call fills a section of the Video RAM, controlled by the Video Display Processor, with a single value.

INPUT PARAMETERS

The address of the first byte in the block to be filled should be given in HL, the length of the block in BC and the value to be written in A.

OUTPUT PARAMETERS

No parameters are returned by this routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF and BC are altered and interrupts enabled by this call.

POSSIBLE USES

This is useful for quickly filling part of the screen with a single colour. It is used by the BASIC keyword PAINT.

LDIRMV 0059

Executed via CALL &H0059

This call moves a block of memory from the Video RAM, controlled by the Video Display Processor, into the main memory.

INPUT PARAMETERS

The address in video RAM should be specified in HL, the length of the block in BC, and the destination address in the main memory in DE.

OUTPUT PARAMETERS

A block of memory as specified above by DE and BC, is returned by this routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, BC, DE and the destination block are affected and interrupts enabled.

POSSIBLE USES

This call is useful if a lot of processing is required on a screen, as it is far quicker to load the screen into memory, process it, then place it back into video RAM using LDIRVM. Another use is for saving screen displays by copying the screen into main memory, then saving this data. The display can be restored using LDIRVM.

Notes

Do not confuse this routine with LDIRVM – LDIRMV moves a block 'from' video RAM.

LDIRVM 005C

Executed via CALL &H005C

This routine copies a block of memory into the RAM controlled by the video display processor.

INPUT PARAMETERS

The address of the block should be specified in HL, the destination in video RAM in DE, and the length, in BC.

OUTPUT PARAMETERS

No parameters are returned by this routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, BC and DE are altered, and interrupts enabled by this call.

Notes

Do not confuse this call with LDIRMV – LDIRVM moves a block of memory 'to' video RAM.

CHGMOD 005F

Executed via CALL &H005F

This call sets the VDP to one of its four modes of operation, i.e. Text mode, Graphics mode I (32 column), Graphics mode II or Multicolour mode.

INPUT PARAMETERS

The Accumulator must contain the VDP mode required, as follows:

A=0 → Text mode (40 column)
A=1 → Graphics mode I (32 column)
A=2 → Graphics mode II (Hi-res)
A=3 → Multicolour mode

OUTPUT PARAMETERS

On exit SCRMOD contains the mode number.

REGISTERS AND MEMORY LOCATIONS ALTERED

The registers changed are AF, BC, DE and HL. Memory locations possibly affected by this call include LINLEN, NAMBAS, CGPBAS, SCRMOD, and OLDSCR.

Notes

This subroutine calls one of INITXT, INIT32, INIGRP or INIMULT depending on the value contained in the accumulator. On exit, interrupts are enabled.

CHGCLR 0062

Executed by CALL &H0062

This call changes the colour of the foreground, background and border if the VDP is in a text mode (40 or 32 column); or changes the border colour if a graphics mode is in operation.

INPUT PARAMETERS

The foreground colour is taken from FORCLR
The background colour is taken from BAKCLR
The border colour is taken from BDRCLR

OUTPUT PARAMETERS

There are no output parameters.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, BC and HL are affected by this call.

CLRSPR 0069

Executed via CALL &H0069

This call initialises all the sprites. The patterns are set to transparent (zero), the sprite colours are set to the current foreground colour; the vertical positions are set to 209 (off the screen); the sprite names are set to the sprite plane number (i.e. the name of the sprite on plane 0 is set to ASCII code 0, etc.)

INPUT PARAMETERS

No input parameter are required.

OUTPUT PARAMETERS

No parameters are returned by this call.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE and HL are affected by this call.

Notes

This routine is not called when changing mode.

INITXT 006C

Executed via CALL &H006C

This entry point initialises the memory locations which describe the screen, for Text mode, then calls SETTXT.

INPUT PARAMETERS

Memory locations TXTNAM,. TXTCGP and LINL40 contain the input parameters.

OUTPUT PARAMETERS

On exit, SCRMOD=0, OLDSCR=0, NAMBAS=TXTNAM, CGPBAS=TXTCGP and LINLEN=LINL40.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL, memory locations RG0SAV, RG1SAV, RG2SAV, RG3SAV, RG4SAV, RG5SAV, RG6SAV and the memory locations listed in the output parameters section may be affected by this call.

Notes

Interrupts are enabled by this call.

INIT32 006F

Executed via CALL &H006F

This entry point initialises the memory locations which describe the screen for Graphics mode I (32 column text), then calls SETT32.

INPUT PARAMETERS

Memory locations T32NAM, T32CGP, T32COL, T32ATR, T32PAT and LINL32 contain the input parameters for this routine.

OUTPUT PARAMETERS

On exit, SCRMOD=1, OLDSCR=1, NAMBAS=T32NAM, CGPBAS=CGPBAS, PATBAS=T32PAT, ATRBAS=T32ATR and LINLEN=LINL32.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL, memory locations RG0SAV, RG1SAV, RG2SAV, RG3SAV, RG4SAV, RG5SAV, RG6SAV and the memory locations listed in the output parameters section, may be affected by this call.

Notes

Interrupts are enabled by this call.

INIGRP 0072

Executed via CALL &H0072

This entry point initialises the various memory locations for Graphics mode II (High resolution mode), then calls SETGRP.

INPUT PARAMETERS

Memory locations GRPNAM, GRPCGP, GRPCOL, GRPATR and GRPPAT contain the input parameters used in this call.

OUTPUT PARAMETERS

On exit, SCRMOD=2, PATBAS=GRPPAT and ATRBAS=GRPATR.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL, memory locations RG0SAV, RG1SAV, RG2SAV, RG3SAV, RG4SAV, RG5SAV, RG6SAV and the memory locations listed in the output parameters section may be affected by this call.

Notes

Interrupts are enabled by this call.

INIMLT 0075

Executed via CALL &H0075

This call initialises the various memory locations for the Multicolour mode, then calls SETMLT.

INPUT PARAMETERS

Memory locations MLTNAM, MLTCGP, MLTCOL, MLTATR and MLTPAT contain the input parameters.

OUTPUT PARAMETERS

On exit, SCRMOD=3, PATBAS=MLTPAT and ATRBAS=MLTATR.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL, memory locations RG0SAV, RG1SAV, RG2SAV, RG3SAV, RG4SAV, RG5SAV, RG6SAV and the memory locations listed in the output parameters section may be affected by this call.

Notes

Interrupts are enabled by this call.

SETTXT 0078

Executed via CALL &H0078

This call sets up the video display processor registers for Text mode (40 column).

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

There are no output parameters.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL and memory locations RG0SAV, RG1SAV, RG2SAV, RG3SAV, RG4SAV, RG5SAV, RG6SAV are affected by this call.

Notes

Calling this routine is not sufficient to change mode; it set the VDP registers but does not initialise the video RAM. Interrupts are enabled during execution of this routine.

SETT32 007B

Executed via CALL &H007B

This call sets up the video display processor registers for Graphics mode I (32 column text).

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

There are no output parameters.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL and memory locations RG0SAV, RG1SAV, RG2SAV, RG3SAV, RG4SAV, RG5SAV, RG6SAV are affected by this call.

Notes

Calling this routine is not sufficient to change mode; it sets the VDP registers but does not initialise the video RAM. Interrupts are enabled during execution of this routine.

SETGRP 007E

Executed via CALL &H007E

This call sets up the video display processor registers for Graphics mode II (high resolution mode).

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

There are no output parameters.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL and memory locations RG0SAV, RG1SAV, RG2SAV, RG3SAV, RG4SAV, RG5SAV, RG6SAV are affected by this call.

Notes

Calling this routine is not sufficient to change mode; it sets the VDP registers but does not initialise the video RAM. Interrupts are enabled during execution of this routine.

SETMLT 0081

Executed via CALL &H0081

This call sets up the video display processor registers for the Multicolour mode.

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

There are no output parameters.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, BC, DE, HL and memory locations RG0SAV, RG1SAV, RG2SAV, RG3SAV, RG4SAV, RG5SAV, RG6SAV are affected by this call.

Notes

Calling this routine is not sufficient to change mode; it sets the VDP registers but does not initialise the video RAM. Interrupts are enabled during execution of this routine.

CALPAT 0084

Executed via CALL &H0084

This call returns the address of a sprite pattern in video RAM.

INPUT PARAMETERS

The sprite number (0-31) should be placed in the accumulator.

OUTPUT PARAMETERS

The address of the sprite in video RAM is returned in the HL register pair.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, DE and HL are affected by this routine.

Notes

The state of the interrupt flip-flop is not changed by this call.

CALATR 0087

Executed via CALL &H0087

This routine returns the address in video RAM of the attribute table of a sprite.

INPUT PARAMETERS

The sprite number should be specified in the accumulator.

OUTPUT PARAMETERS

On return, the HL register pair contains the address in video RAM of the desired sprite attribute table.

REGISTERS AND MEMORY LOCATIONS ALTERED

DE, HL and the flags are modified by this routine.

Notes

Interrupts are not affected.

GSPSIZ 008A

Executed via CALL &H008A

This call returns the current sprite size in terms of the number of bytes occupied by each sprite (8 or 32).

INPUT PARAMETERS

No input parameters are required.

OUTPUT PARAMETERS

The number of bytes per sprite is returned in the accumulator.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator and flags are affected by this routine.

Notes

If 16x16 sprites are in use, the carry flag is set on exit, otherwise it is cleared. Interrupts are not affected by this call.

GRPPRT 008D

Executed via CALL &H008D

This call is used to print a character on the Graphics screen (high resolution mode) at the present position of the graphics cursor.

INPUT PARAMETERS

The code of the character to be printed should be placed in the accumulator.

OUTPUT PARAMETERS

No parameters are returned from this routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

No registers or memory locations are affected.

Notes

This routine will only work in Graphics mode II.

SCALXY 010E

Executed via CALL &H010E

This call ensures that a point, passed through registers, lies on the screen. If not, the coordinates are 'clipped' and an error is flagged. Clipping means that if x or y are too big, the maximum permitted values are assigned to them, and if negative they are set to zero. Also, if in Multicolour mode, x and y are divided by 4, as this mode has 64 horizontal cells and 48 vertical ones.

INPUT PARAMETERS

BC should contain the x coordinate, and DE the y coordinate.

OUTPUT PARAMETERS

On exit BC and DE contain the 'clipped' x and y coordinates respectively. If either x or y is found to be out of range the carry flag will be cleared, otherwise it will be set.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, BC and DE may be altered by this routine.

Notes

Be wary of the fact that x and y are scaled if the Multicolour mode is active – do not scale them yourself as well!

The ranges of x and y, in Graphics mode II or Multicolour mode, are 0-255 and 0-191, respectively.

MAPXYC 0111

Executed via CALL &H0111

This very useful routine calculates the address in video RAM of a pixel in either Graphics mode II or Multicolour mode. It also returns the position in that byte of the bits representing the pixel.

INPUT PARAMETERS

The position (x,y) of the point of interest should be placed in BC (x) and DE (y).. For Graphics mode II, x must be in the range 0-255, and y, 0-191. In multicolour mode, x must lie in the range 0-63, and y in the range 0-47. Also, this routine uses SCRMOD to determine the screen mode, GRPCGP to find the start address of the pattern generator table in Graphics mode II, and MLTCGP to find the start address of the pattern generator table in Multicolour mode.

OUTPUT PARAMETERS

The address in video RAM, of the byte containing the pixel, is returned in CLOC and the byte indicating the relevent bits describing that pixel is returned in CMASK. In Graphics mode II, the pixel of interest is represented by one bit (foreground=1, background=0). This bit is in the same position as the set bit in CMASK. E.g. if the pixel is stored in bit 5 of the byte pointed to by CLOC, then CMASK will contain 00100000 in binary. For Multicolour mode, each pixel is represented by four bits – the colour of that pixel, and again the position of those four bits correspond to the position of the set bits in CMASK. E.g. if the pixel is stored in bits 0-3 of the byte pointed to by CLOC, then CMASK will contain 00001111 in binary.

REGISTERS AND MEMORY LOCATIONS ALTERED

Registers AF, D, HL and memory locations CLOC and CMASK are affected by this call.

Notes

A check that the pixel is actually on the screen is not performed by this call. To make this check and to apply scaling to the x and y values for the Multicolour mode, subroutine SCALXY should be used.

FETCHC 0114

Executed via CALL &H0114

This routine simply reads the current pixel address and mask pattern.

INPUT PARAMETERS

CLOC should contain the current pixel address, and CMASK the mask pattern.

OUTPUT PARAMETERS

On exit, HL contains the contents of CLOC, and A contains the value stored in CMASK.

REGISTERS AND MEMORY LOCATIONS ALTERED

HL and AF are the only registers altered by this call.

Notes

This call can be replaced with two Z80 instructions, i.e.

```
LD      A, (CMASK)
LD      HL, (CLOC)
```

STOREC 0117

Executed via CALL &H0117

This routine simply stores the current pixel address and mask pattern in memory.

INPUT PARAMETERS

HL should contain the current pixel address, and A, the mask pattern.

OUTPUT PARAMETERS

On exit, CMASK contains a copy of the accumulator, and CLOC contains the value held in HL.

REGISTERS AND MEMORY LOCATIONS ALTERED

Memory locations CMASK and CLOC are affected by this call.

Notes

This call can be replaced with two Z80 instructions, i.e.

```
LD      (CMASK), A
LD      (CLOC), HL
```

SETATR 011A

Executed via CALL &H011A

This call sets the attribute byte to the contents of the accumulator if the number in the accumulator is less than 16, otherwise the attribute byte is unchanged and an error is flagged.

INPUT PARAMETERS

The accumulator should contain the value to which the attribute byte is to be set.

OUTPUT PARAMETERS

On exit, ATRBYT (the attribute byte) contains the value specified in the accumulator, provided it is less than 16.

REGISTERS AND MEMORY LOCATIONS ALTERED

ATRBYT may be altered by this call.

Notes

The attribute byte represents the colour of a pixel. It is used by routine SETC to set a pixel to this colour.

READC 011D

Executed via CALL &H011D

This call reads the attribute (colour) of the current pixel.

INPUT PARAMETERS

The current pixel address and mask pattern should be stored in CLOC and CMASK respectively.

OUTPUT PARAMETERS

On exit, the colour of the current pixel is stored in the accumulator.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags are affected by this call.

SETC 0120

Executed via CALL &H0120

This call sets the current pixel to the specified colour. If in Multicolour mode, this is the only affect on the display. However, in Graphics mode II, if the specified colour is not the same as either the foreground or background colour, then the foreground colour of all the pixels in the byte containing the current pixel are changed to the specified colour.

INPUT PARAMETERS

The current pixel address and mask pattern should be contained in CLOC and CMASK, respectively, and the colour to which this pixel is to be set should be specified in ATRBYT.

OUTPUT PARAMETERS

No parameters are returned by this routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only the accumulator and flags are affected by this call.

Notes

If all the pixels in a particular byte are set to the same colour, that colour becomes the new background colour: i.e. the byte in video RAM pointed at by CLOC will be zero.

RIGHTC 00FC

Executed via CALL &H00FC

This routine moves the graphics cursor one pixel to the right.

INPUT PARAMETERS

The address of the byte containing the present pixel is stored in CLOC and the pixel mask pattern in CMASK.

OUTPUT PARAMETERS

The contents of CLOC and CMASK will have been updated to the new graphics cursor position.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, and the memory locations CLOC and CMASK, may be altered by this call.

Notes

Moving the graphics cursor off the right-hand side of the screen will cause it to appear on the left side, one pixel down. Also, no check is made to ensure that the graphics cursor does not move off the screen completely, via the bottom right hand corner.

In Multicolour mode, the graphics cursor is moved by one block of 4x4 pixels.

LEFTC 00FF

Executed via CALL &H00FF

This routine moves the graphics cursor one pixel to the left.

INPUT PARAMETERS

The address of the byte containing the present pixel is stored in CLOC and the pixel mask pattern in CMASK.

OUTPUT PARAMETERS

The contents of CLOC and CMASK will have been updated to the new graphics cursor position.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, and the memory locations CLOC and CMASK, may be altered by this call.

Notes

Moving the graphics cursor off the left-hand side of the screen will cause it to appear on the right-hand side, one pixel up. Also, no check is made to ensure that the graphics cursor does not move off the screen completely, via the top left hand corner.

In Multicolour mode, the graphics cursor is moved by one block of 4x4 pixels.

UPC 0102

Executed via CALL &H0102

This routine moves the graphics cursor one pixel up if it is not at the top of the screen, otherwise its position remains unchanged.

INPUT PARAMETERS

The address of the byte containing the present pixel is stored in CLOC and the pixel mask pattern in CMASK.

OUTPUT PARAMETERS

The contents of CLOC and CMASK will have been updated to the new graphics cursor position.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, and the memory locations CLOC and CMASK, may be altered by this call.

Notes

In Multicolour mode, the graphics cursor is moved by one block of 4x4 pixels.

TUPC 0105

Executed via CALL &H0105

This routine moves the graphics cursor one pixel up if it is not at the top of the screen, otherwise its position remains unchanged and the carry flag is set.

INPUT PARAMETERS

The address of the byte containing the present pixel is stored in CLOC and the pixel mask pattern in CMASK.

OUTPUT PARAMETERS

The contents of CLOC and CMASK will have been updated to the new graphics cursor position.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, and the memory locations CLOC and CMASK, may be altered by this call.

Notes

This call is virtually identical to UPC: the only difference is that the carry flag is set if the graphics cursor is already at the top of the screen, and it is cleared otherwise.

In Multicolour mode, the graphics cursor is moved by one block of 4x4 pixels.

DOWNC 0108

Executed via CALL &H0108

This routine moves the graphics cursor one pixel down if it is not at the bottom of the screen, otherwise its position remains unchanged.

INPUT PARAMETERS

The address of the byte containing the present pixel is stored in CLOC and the pixel mask pattern in CMASK.

OUTPUT PARAMETERS

The contents of CLOC and CMASK will have been updated to the new graphics cursor position.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, and the memory locations CLOC and CMASK, may be altered by this call.

Notes

In Multicolour mode, the graphics cursor is moved by one block of 4x4 pixels.

TDOWNC 010B

Executed via CALL &H010B

This routine moves the graphics cursor one pixel down if it is not at the bottom of the screen, otherwise its position remains unchanged and the carry flag is set.

INPUT PARAMETERS

The address of the byte containing the present pixel is stored in CLOC and the pixel mask pattern in CMASK.

OUTPUT PARAMETERS

The contents of CLOC and CMASK will have been updated to the new graphics cursor position.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, and the memory locations CLOC and CMASK, may be altered by this call.

Notes

This call is virtually identical to DOWNC: the only difference is that the carry flag is set if the graphics cursor is already at the bottom of the screen, and it is cleared otherwise.

In Mulicolour mode, the graphics cursor is moved by one block of 4x4 pixels.

NSETCX 0123

Executed via CALL &H0123

This call sets a specified number of pixels, to the right of the current graphics cursor, to a given colour.

INPUT PARAMETERS

HL should contain the number of pixels to be set, ATRBYT the colour of the pixels, and the graphics cursor address and mask pattern should be stored in CLOC and CMASK, respectively.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, BC, DE, HL and memory locations CLOC and CMASK may be affected by this call.

Notes

If in Multicolour mode, the position of the graphics cursor will be one pixel to the left of the last one set, but in Graphics mode II this position remains unchanged.

Interrupts are enabled by this routine.

GTASPC 0126

Executed via CALL &H0126

This routine is used by the BASIC interpreter CIRCLE command, to fetch the current aspect ratio parameters.

INPUT PARAMETERS

ASPCT1 should contain 256/(aspect ratio) and ASPCT2 should contain 256*(aspect ratio).

OUTPUT PARAMETERS

On exit, HL should contain the contents of ASPCT2 and DE, ASPCT1.

REGISTERS AND MEMORY LOCATIONS ALTERED

Only HL and DE are affected by this call.

PNTINI 0129

Executed via CALL &H0129

This is an initialisation routine for the BASIC PAINT instruction. The border colour of the shape to be filled is checked, so that it is less than 16, and then stored.

INPUT PARAMETERS

The accumulator should contain the colour of the border which is to be filled.

OUTPUT PARAMETERS

If the Multicolour mode is currently active, then BRDATR contains a copy of the input parameter, otherwise it contains a copy of the value stored in ATRBYT. Also, if the input parameter is greater than 15, when in Multicolour mode, the carry flag will be set on return from the routine.

REGISTERS AND MEMORY LOCATIONS ALTERED

The accumulator, flags and memory location BRDATR, may be affected by this routine.

Notes

The border colour is only used in the Multicolour mode PAINT routine. In Graphics mode II, the PAINT routine searches for the foreground colour.

SCANR 012C

Executed via CALL &H012C

This call scans a specified number of pixels to the right of the graphics cursor, setting them to a given colour, until a certain 'border colour' pixel is found, or the right hand side of the screen has been reached, or the maximum number of pixels has been scanned.

INPUT PARAMETERS

The maximum number of pixels to be scanned should be stored in DE (up to 256), the border colour should be stored in BRDATR, and the colour to which the pixels are to be set, in ATRBYT.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, BC, DE, HL and memory locations CLOC, CMASK and FILNAM to FILNAM+3 may be affected by this call.

Notes

This routine is used by the PAINT instruction; FILNAM to FILNAM+3 being used as workspace.

Interrupts are enabled by this call.

SCANL 012F

Executed via CALL &H012F

This call scans a specified number of pixels to the left of the graphics cursor, setting them to a given colour, until a certain 'border colour' pixel is found, or the left hand side of the screen has been reached, or the maximum number of pixels has been scanned.

INPUT PARAMETERS

The maximum number of pixels to be scanned should be stored in DE (up to 256), the border colour should be stored in BRDATR, and the colour to which the pixels are to be set, in ATRBYT.

OUTPUT PARAMETERS

No parameters are returned.

REGISTERS AND MEMORY LOCATIONS ALTERED

AF, BC, DE, HL and memory locations CLOC, CMASK and FILNAM to FILNAM+3 may be affected by this call.

Notes

This routine is used by the PAINT instruction, FILNAM to FILNAM+3 being used as workspace.

Interrupts are enabled by this call.

Section 4.8

THE USE OF HOOKS

Hooks provide a way to intercept the BASIC interpreter or operating system at certain points, enabling additional, or alternative, processing to be carried out. To expand on the last sentence it is easiest to use an example. Consider for example the BIOS call CHPUT (see section 2.3). This routine is used to print text on the screen. Here is a disassembly of the first few instructions of CHPUT:–

```
PUSH    HL          ; SAVE ALL THE REGISTERS USED
PUSH    DE
PUSH    BC
PUSH    AF
CALL    &HFDA4      ; CALL THE HOOK H.CHPU
```

First of all registers HL, BC, DE and AF are pushed on to the stack. Then memory address &HFDA4 is called. Normally, the contents of &HFDA4 and the following 4 bytes, is a Z80 RET instruction, immediately passing control to the instruction following the CALL. However, &HFDA4 is in RAM, so its contents can be altered. Let the 3 bytes, starting at &HFDA4 be changed, so that control is passed to another subroutine at &HD000. i.e.

```
FDA4    C3  00  D0      JP          &HD0000
```

Also let the code at &HD000 be as shown below:–

```
POP     HL          ; GET RID OF THE FIRST ITEM ON THE
                      STACK (i.e. RETURN TO HOOK ADDRESS)
POP     AF          ; RESTORE THE REGISTERS
POP     BC
POP     DE
POP     HL
RET                 ; EXIT BIOS CALL CHPUT.
```

Now the item on the top of the stack is 3 more than the address from which &HFDA4 was called, i.e. the return address to CHPUT. Therefore, if this is removed from the stack, this will restore the stack to the state it was in immediately before calling &HFDA4. Also, if the next four items are popped as shown above, the stack will be as it was on entering CHPUT, so the RET instruction will exit the subroutine CHPUT. Consequently the character which was to be printed will not be!

This example demonstrates how the operating system can be intercepted. The five bytes, starting at address &HFDA4, are known as a hook. Similar 'hooks' exist at many other points in the operating system and BASIC interpreter, and a list of them, in alphabetical order, is contained in the following pages.

NAME	ADDRESS	DETAILS
H-ATTR	FE1C	MSXSTS, at the beginning of ATTR$ (attribute) routine
H-BAKU	FEAD	SPCDSK, at the BAKUPT (back up) routine
H-BINL	FE76	SPCDSK, at the BINLOD (binary load) routine
H-BINS	FE71	SPCDSK, at the BINSAV (binary save) routine
H-BUFL	FF8E	BINTRP, at the BUFLIN (buffer line) routine
H-CHGE	FDC2	MSXIO, at the beginning of CHGET (character get) routine
H-CHPU	FDA4	MSXIO, at the beginning of CHPUT (character put) routine
H-CHRG	FF48	BINTRP
H-CLEA	FED0	BIMISC, at the CLEARC (CLEAR clear) routine
H-CMD	FE0D	MSXSTS, at the beginning of CMD (command) routine
H-COMP	FF57	BINTRP
H-COPY	FE08	MSXSTS, at the beginning of COPY (copy files) routine
H-CRDO	FEE9	BIO, at the CRDO (CRlf DO) routine
H-CRUN	FF20	BINTRP
H-CRUS	FF25	BINTRP
H-CVD	FE49	MSXSTS, at the beginning of CVD (convert dbl) routine
H-CVI	FE3F	MSXSTS, at the beginning of CVI (convert int) routine
H-CVS	FE44	MSXSTS, at the beginning of CVS (convert sng) routine
H-DEVN	FEC1	SPCDEV, at the DEVNAM (device name) routine
H-DGET	FE80	SPCDSK, at the DGET (disk get) routine
H-DIRD	FF11	BINTRP, at the DIRDO (direct statement DO) entry
H-DOGR	FEF3	GENGRP, at the DOGRPH (do graph) routine
H-DSKC	FEEE	BIO, at the DSKCHI (disc character input) routine
H-DSKF	FE12	MSXSTS, at the beginning of DSKF (disk free) routine
H-DSKI	FE17	MSXSTS, at the beginning of DSKI (disk input) routine
H-DSKO	FDEF	MSXSTS, at the beginning of DSKO$ (disk output) routine
H-DSPC	FDA9	MSXIO, at the beginning of DSPCSR (display cursor) routine
H-DSPF	FDB3	MSXIO, at the beginning of DSPFNK (display function key) routine
H-EOF	FEA3	SPCDSK, at the EOF (end of file) function
H-ERAC	FDAE	MSXIO, at the beginning of ERACSR (erase cursor) routine
H-ERAF	FDB8	MSXIO, at the beginning of ERAFNK (erase function key) routine
H-ERRF	FF02	BINTRP
H-ERRO	FFB1	BINTRP, at the ERROR routine
H-ERRP	FEFD	BINTRP, at the ERRPRT (error print) routine
H-EVAL	FF70	BINTRP
H-FIEL	FE2B	MSXSTS, at the beginning of FIELD (field) routine
H-FILE	FE7B	SPCDSK, at the FILES command
H-FILO	FE85	SPCDSK, at the FILOU1 (file out 1) routine
H-FINE	FF1B	BINTRP
H-FING	FF7A	BINTRP
H-FINI	FF16	BINTRP
H-FINP	FF5C	BINTRP
H-FORM	FFAC	MSXIO, at the FORMAT (disk formatter) routine
H-FPOS	FEA8	SPCDSK, at the FPOS (file position) function
H-FRET	FF9D	BISTRG, at the FRETM (free up to temporaries) routine
H-FRME	FF66	BINTRP
H-FRQI	FF93	BINTRP, at the FRQINT routine
H-GEND	FEC6	SPCDEV, at the GENDSP (general device dispatcher)
H-GETP	FE4E	SPCDSK, at the GETPTR (get file pointer) routine
H-GONE	FF43	BINTRP
H-INDS	FE8A	SPCDSK, at the INDSKC (input disk character) routine
H-INIP	FDC7	MSXIO, at the beginning of INIPAT (initialise pattern) routine
H-INLI	FDE5	MSXINL, at the beginning of LININ (line input) routine

Section 4.9

THE SYSTEM RAM

This section contains a listing of every system variable, along with its address and length in bytes. The address is given in hexadecimal and the length in decimal. The system RAM extends from &HF380 to &HFFFF. Locations &HF380 to &HF399 inclusive actually contain machine code subroutines which are used by the slot management BIOS calls.

NAME	ADDRESS	LENGTH	NAME	ADDRESS	LENGTH
ARG	F847	16	CRCSUM	F93D	2
ARYTA2	F7B5	2	CRTCNT	F3B1	1
ARYTAB	F6C4	2	CS120	F3FC	10
ASPCT1	F40B	2	CSAVEA	F942	2
ASPCT2	F40D	2	CSAVEM	F944	1
ASPECT	F931	2	CSCLXY	F941	1
ATRBAS	F928	2	CSRSW	FCA9	1
ATRBYT	F3F2	1	CSRX	F3DD	1
AUTFLG	F6AA	1	CSRY	F3DC	1
AUTINC	F6AD	2	CSTCNT	F93F	2
AUTLIN	F6AB	2	CSTYLE	FCAA	1
BAKCLR	F3EA	1	CURLIN	F41C	2
BASROM	FBB1	1	CXOFF	F945	2
BDRCLR	F3EB	1	CYOFF	F947	2
BOTTOM	FC48	2	DAC	F7F6	16
BRDATR	FCB2	1	DATLIN	F6A3	2
BUF	F55E	258	DATPTR	F6C8	2
BUFEND	FC18	0	DECCNT	F7F4	1
BUFMIN	F55D	1	DECTM2	F7F2	2
CAPST	FCAB	1	DECTMP	F7F0	2
CASPRV	FCB1	1	DEFTBL	F6CA	26
CENCNT	F933	2	DEVICE	FD99	1
CGPBAS	F924	2	DIMFLG	F662	1
CGPNT	F91F	3	DONUM	F665	1
CLIKFL	FBD9	1	DORES	F664	1
CLIKSW	F3DB	1	DOT	F6B5	2
CLINEF	F935	1	DRWANG	FCBD	1
CLMLST	F3B2	1	DRWFLG	FCBB	1
CLOC	F92A	2	DRWSCL	FCBC	1
CLPRIM	F38C	14	DSCTMP	F698	3
CMASK	F92C	1	ENDBUF	F660	1
CNPNTS	F936	2	ENDFOR	F6A1	2
CNSDFG	F3DE	1	ENDPRG	F40F	5
CODSAV	FBCC	1	ENSTOP	FBB0	1
CONLO	F66A	8	ERRFLG	F414	1
CONSAV	F668	1	ERRLIN	F6B3	2
CONTXT	F666	2	ERRTXT	F6B7	2
CONTYP	F669	1	ESCCNT	FCA7	1
CPCNT	F939	2	EXPTBL	FCC1	4
CPCNT8	F93B	2	FBUFFR	F7C5	43
CPLOTF	F938	1	FILNAM	F866	11

NAME	ADDRESS	LENGTH	NAME	ADDRESS	LENGTH
FILNM2	F871	11	MAXUPD	F3EC	3
FILTAB	F860	2	MCLFLG	F958	1
FLBMEM	FCAE	1	MCLLEN	FB3B	1
FLGINP	F6A6	1	MCLPTR	FB3C	2
FNKFLG	FBCE	10	MCLTAB	F956	2
FNKSTR	F87F	160	MEMSIZ	F672	2
FNKSWI	FBCD	1	MINDEL	F92D	2
FORCLR	F3E9	1	MINUPD	F3EF	3
FRCNEW	F3F5	1	MLTATR	F3D7	2
FRETOP	F69B	2	MLTCGP	F3D5	2
FSTPOS	FBCA	2	MLTCOL	F3D3	2
FUNACT	F7BA	2	MLTNAM	F3D1	2
GETPNT	F3FA	2	MLTPAT	F3D9	2
GRPACX	FCB7	2	MOVCNT	F951	2
GRPACY	FCB9	2	MUSICF	FB3F	1
GRPATR	F3CD	2	NAMBAS	F922	2
GRPCGP	F3CB	2	NEWKEY	FBE5	11
GRPCOL	F3C9	2	NLONLY	F87C	1
GRPHED	FCA6	1	NOFUNS	F7B7	1
GRPNAM	F3C7	2	NTMSXP	F417	1
GRPPAT	F3CF	2	NULBUF	F862	2
GXPOS	FCB3	2	OLDKEY	FBDA	11
GYPOS	FCB5	2	OLDLIN	F6BE	2
HEADER	F40A	1	OLDSCR	FCBO	1
HIGH	F408	2	OLDTXT	F6CO	2
HIMEM	FC4A	2	ONEFLG	F6BB	1
HOLD	F83E	8	ONELIN	F6B9	2
HOLD2	F836	8	ONGSBF	FBD8	1
HOLD8	F806	48	OPRTYP	F664	0
INSFLG	FCA8	1	PADX	FC9D	1
INTCNT	FCA2	2	PADY	FC9C	1
INTFLG	FC9B	1	PARM1	F6E8	100
INTVAL	FCAO	2	PARM2	F750	100
JIFFY	FC9E	2	PATBAS	F926	2
KANAMD	FCAD	1	PATWRK	FC40	8
KANAST	FCAC	1	PDIREC	F953	1
KBUF	F4IF	318	PLYCNT	FB40	1
KEYBUF	FBFO	40	PRMFLG	F7B4	1
LFPROG	F954	1	PRMLEN	F6E6	2
LINL32	F3AF	1	PRMLN2	F74E	2
LINL40	F3AE	1	PRMPRV	F74C	2
LINLEN	F3B0	1	PRMSTK	F6E4	2
LINTTB	FBB2	24	PROCNM	FD89	16
LINWRK	FC18	40	PRSCNT	FB35	1
LOHADR	F94B	2	PRTFLG	F416	1
LOHCNT	F94D	2	PTRFIL	F864	2
LOHDIR	F94A	1	PTRFLG	F6A9	1
LOHMSK	F949	1	PUTPNT	F3F8	2
LOW	F406	2	QUEBAK	F971	4
LOWLIM	FCA4	1	QUETAB	F959	24
LPTPOS	F415	1	QUEUEN	FB3E	1
MAXDEL	F92F	2	QUEUES	F3F3	2
MAXFIL	F85F	1	RAWPRT	F418	1

NAME	ADDRESS	LENGTH	NAME	ADDRESS	LENGTH
RDPRIM	F380	5	T32COL	F3BF	2
REPCNT	F3F7	1	T32NAM	F3BD	2
RGOSAV	F3DF	1	T32PAT	F3C5	2
RG1SAV	F3EO	1	TEMP	F6A7	2
RG2SAV	F3E1	1	TEMP2	F6BC	2
RG3SAV	F3E2	1	TEMP3	F69D	2
RG4SAV	F3E3	1	TEMP8	F69F	2
RG5SAV	F3E4	1	TEMP9	F7B8	2
RG6SAV	F3E5	1	TEMPPT	F678	2
RG7SAV	F3E6	1	TEMPST	F67A	30
RNDX	F857	8	TRCFLG	F7C4	1
RS2IQ	FAF5	64	TRGFLG	F3E8	1
RTPROG	F955	1	TRPTBL	FC4C	78
RTYCNT	FC9A	1	TTYPOS	F661	1
RUNBNF	FCBE	1	TXTATR	F3B9	2
RUNFLG	F866	0	TXTCGP	F3B7	2
SAVEND	F87D	2	TXTCOL	F3B5	2
SAVENT	FCBF	2	TXTNAM	F3B3	2
SAVSP	FB36	2	TXTPAT	F3BB	2
SAVSTK	F6B1	2	TXTTAB	F676	2
SAVTXT	F6AF	2	USFLG	F6A6	0
SAVVOL	FB39	2	USRTAB	F39A	20
SCNCNT	F3F6	1	VALTYP	F663	1
SCRMOD	FCAF	1	VARTAB	F6C2	2
SKPCNT	F94F	2	VCBA	FB41	37
SLTATR	FCC9	64	VCBB	FB66	37
SLTTBL	FCC5	4	VCBC	FB8B	37
SLTWRK	FD09	128	VLZADR	F419	2
STATFL	F3E7	1	VLZDAT	F41B	1
STKTOP	F674	2	VOICAQ	F975	128
STREND	F6C6	2	VOICBQ	F9F5	128
SUBFLG	F6A5	1	VOICCQ	FA75	128
SWPTMP	F7BC	8	VOICEN	FB38	1
T32ATR	F3C3	2	WINWID	FCA5	1
T32CGP	F3C1	2	WRPRIM	F385	7

INDEX

Bold type indicates a reference to Section 3 — Basic Keywords.

Write to Us

Melbourne House is always interested in receiving letters from its readers.

Publishing Ideas

If you have written a book or program that you think would be of interest to other computer users, we want to hear from you.

We are always interested in discussing new ideas for books with authors. If you think you have a good book idea, please send a detailed outline first. We prefer to work with authors as early as possible in the writing process.

BASIC programs are wanted for inclusion in our books, and machine language programs are wanted for our list of adventure and game software. Always send a tape or disk and, if possible, a code printout with your submission letter.

Fees and royalties are negotiated according to the quality and ingenuity of the submission, and are more than competitive with those of other publishing houses.

Send your book or program to the Melbourne House office closest to you — see the back of the title page for the address. Mark your letter to the attention of the Editorial Department to ensure an early review of your idea and a prompt reply.

Bugs and Problems

Every effort is made to ensure that our books are error-free. Occasionally, however, you may have difficulties — in such instances do not hesitate to write to Melbourne House. Send your letter to the Melbourne House office closest to you — see the back of the title page for the address.

So that we can process your query as quickly as possible, mark your letter to the attention of Customer Support. Quote the title of this book in your letter, together with the printing and edition numbers, and the year of publication. This information is on the back of the title page at the foot.

Describe your problem precisely, quoting the program title and the offending line numbers.

MSX

Melbourne House

THE COMPLETE MSX PROGRAMMERS GUIDE is the definitive handbook for the **MSX standard** computer; it is invaluable for all MSX users from novice to advanced.

To enable users of all levels to gain maximum benefit from the book, THE COMPLETE MSX PROGRAMMERS GUIDE is set out in four parts.

The first section comprises a comprehensive, self-paced guide which takes the beginner from square one. It de-mystifies the process of learning to program on MSX computers.

Advanced programming techniques which will serve to sharpen the experienced programmer's existing skills are then looked at. Information on sound and graphics facilities is also included.

The third section contains detailed explanations of both BASIC and machine language programming.

The final section shows you how your MSX computer works by giving a complete guide to the operating system.

Tom Sato is a well respected freelance journalist in the UK and Japan, whose vast experience and familiarity with the **MSX standard** computer makes THE COMPLETE MSX PROGRAMMERS GUIDE the exemplary book for the MSX user.

Melbourne House Publishers

ISBN 0-86161-173-X

9 780861 611737