MSX

Melbourne House

MSX EXPOSED

Joe Pritchard

MSX Exposed

86|08753

MSX EXPOSED

Joe Pritchard

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
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

Cataloguing in Publication

Pritchard, Joe.
MSX exposed.
Includes index.
ISBN 0 86161 182 9.

1. MSX (Computer system). 2. Software compatibility. I. Title.

001.64'4

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

Figures

Preface

In this book I give the MSX programmer enough insight into the way in which the various devices that make up the MSX computer can be programmed to gain the maximum benefit from the machine. Many of the demonstration routines for directly accessing the various system components are written in BASIC, but the principles involved can be transferred directly to machine code programming.

One thing that I have not tried to do is teach the programmer Z80 machine code programming, which would take a book in itself. Instead I have given particular note to how the machine code programmer can gain access to the MSX system components as easily as if the program were being written in BASIC.

I would like to express my gratitude to all those who have been involved, directly or indirectly, with the production of this book: Alfred Milgrom and his marvellous staff, Dr Ian Logan for suggesting the structure of the book, my family — especially my mother, father and uncle — who have put up with an absence of mail and visits during the preparation of the book, and my wife Nicky, who has put up with my disappearing for several hours at a time to write and type the manuscript. This book is dedicated to all these people, as well as to the staff of the South Yorkshire & Humberside Microelectronics Education Programme Regional Centre . . . and the late Mrs H.M.A. Brownlow for reasons that will be obvious to anyone who knows me.

Joe Pritchard
Doncaster, 1984

1 The MSX System

The MSX System is a totally new departure for the home computer; a series of computers that are compatible with each other in terms of BASIC Language and performance! Any computer that claims the MSX standard will be able to run software that has been written for other MSX machines and MSX computers will have access, therefore, to an extremely wide range of software. The MSX standard provides a minimum system which machines must adhere to, thus minimising or totally removing the fears that software written for one machine in the range will not run on others. Once the machine possesses this minimum specification, individual manufacturers will no doubt supply machine specific features, which will include such items as the electronics necessary to handle a printer and joysticks. However, certain elements of the system will always be constant, and it is these parts of the system that this book will describe and explore. Let's begin by taking a look at the system as a whole.

Minimum MSX Implementation

Figure 1.1 shows a block diagram of the 'innards' of an MSX machine. This is not the place to give an indepth description of each of the major system components; this will be done in later chapters. However, we shall look briefly at each component of the system to see where it fits in with regard to the rest of the system.

Z-80 CPU

This is the heart of the MSX System, the Central Processor Unit. It is a microprocessor chip, an electronic device that, with certain other components, controls the action of the rest of the system. It can be controlled by a series of instructions which it performs in a stored sequence. This sequence of instructions is called a PROGRAM. We will look more closely at this device in subsequent chapters; suffice to say for the moment that any computer in the MSX series will have this chip as it's CPU. The CPU is always running the program contained in the MSX ROM, and it is this ROM program, called the BASIC Interpreter, that executes your BASIC program. You can instruct the CPU to run other programs, that you have written in a language called MACHINE CODE, by using a BASIC command called USR. We shall meet this command in greater detail in the chapters concerned with mixing BASIC and machine code in your programs.

ROM

This stands for Read Only Memory, and contains a program that is executed by the CPU when the machine is turned on. The program stored in this area of memory is permanent and unalterable by the user, and provides the instructions needed by the CPU to enable it to read the keyboard, execute your BASIC programs and perform the dozens of other tasks that your MSX computer must do. Any computer in the MSX series will have a ROM that is very similar to the one in your machine, if not identical. It is what is called a 32k ROM, there being space inside it to store over 32000 different numbers, each number having a value between 0 and 255 and representing an instruction to the Z-80 or part of an instruction or an item of data that the CPU may need to perform its tasks. More details about 'k's and numbers will be given in the appendix, which will investigate number systems.

RAM

This is another area of memory in the computer, but is of a different type to the ROM. This memory is called Random Access Memory, and the user can modify its contents with no difficulty. This is where your BASIC program lives when you have typed it in, and it is also the memory that the computer uses as 'scrap paper' when it is running your programs. Any variables that your BASIC program declares while it is running are stored in RAM. Commands like CLEAR and NEW affect RAM; CLEAR sets all numeric variables to zero and all string variables to empty strings by directly affecting the area of RAM that holds the variables. NEW clears away a program from the memory of the computer by again directly affecting the contents of RAM. The BASIC command POKE also enables the user to modify RAM, as we shall see when we come to write machine code programs for the machine. There is one final, and rather drastic way, to modify RAM, and that is to turn the computer off!

The minimum amount of RAM that an MSX system can have is 8k, and you can add memory to your system via the slot system. More details about slots will be given in the chapter on Memory Maps.

Cartridge Slot

This is again a vital feature of the MSX concept, and all computers flying the MSX pennant should be totally compatible in this respect. The slot is a means by which the on board memory can be added to in various ways. There are 4 slots on the minimum MSX system, and it is possible to add more. For the moment, we will simply describe the slots as means of adding more ROM or RAM to the system.

Video RAM

This is a special type of RAM that is totally dedicated to holding the information used to put the display together. Whereas the normal RAM is directly accessible to the CPU, the Video RAM is not. The CPU alters memory locations in Video RAM by using the Video Display Processor. There is 16k of Video RAM, commonly abbreviated to VRAM, and this abbreviation will be used throughout this book.

Video Display Processor

This is another device that any computer in the MSX range must possess. The Video Display Processor, or VDP, is a device dedicated to the control of the video or television display of the computer. In the MSX series, the device used is the TMS9918A or a chip that is very similar. The VDP is directed by the CPU, and provides 4 different types of display. These display types are called display MODES, and will be dealt with in greater detail in a subsequent chapter. The VDP interfaces the computer to the display unit that is to be used with the computer. All the information that it requires to produce the display is held in VRAM, and the way that the VDP interprets this data depends upon the display mode in use. VRAM is modified whenever the user writes anything to the screen, uses the CLS command, the COLOR command, VPOKE or any of the sprite related commands.

Programmable Sound Generator

This device is the third chip that is central to the MSX concept and must in all machines be compatible with the General Instrument AY-3-8910 device. The abbreviation that we shall employ for Programmable Sound Generator is PSG, and it is responsible for the wide range of sounds that can be obtained from the MSX computers. It interfaces directly with the CPU and is controlled by the CPU. It is also responsible for implementing the Input and Output functions of the system that are accessible to the user, usually in the form of Joysticks.

System Input/Output

The CPU needs to be able to interact with other devices apart from the display and the sound generator. These include such things as tape recorders, the keyboard, and printers. The interface between these devices and the CPU is called the Programmable Peripheral Input/Output chip, or PPI for short. The PPI must be compatible with the 8255 device. It is controlled by the CPU.

That sums up the essential components of the MSX computers. Subsequent chapters in this book will describe how each component fits into the system, giving the user an insight into the operation of the computer. However, let's take a brief look at how the components are connected together. Figure **1.1** shows how the components of the system interact with each other; but exactly how does data pass between, say, the CPU and the Read Only Memory?

The BUS System

In computing, the BUS is not the large, wheeled vehicle that we all manage to miss at bus stops. A bus in a computer is a collection of electrical conductors that carry electrical signals representing either a binary '1' or a 'Ø'; signals are conveyed around the computer in a binary form, and such a 1 or Ø is called a binary digit, or BIT. If you are interested in the binary system, then an account of it is given in the appendices at the end of the book. 8 of these bits make up what is called a byte, and this is

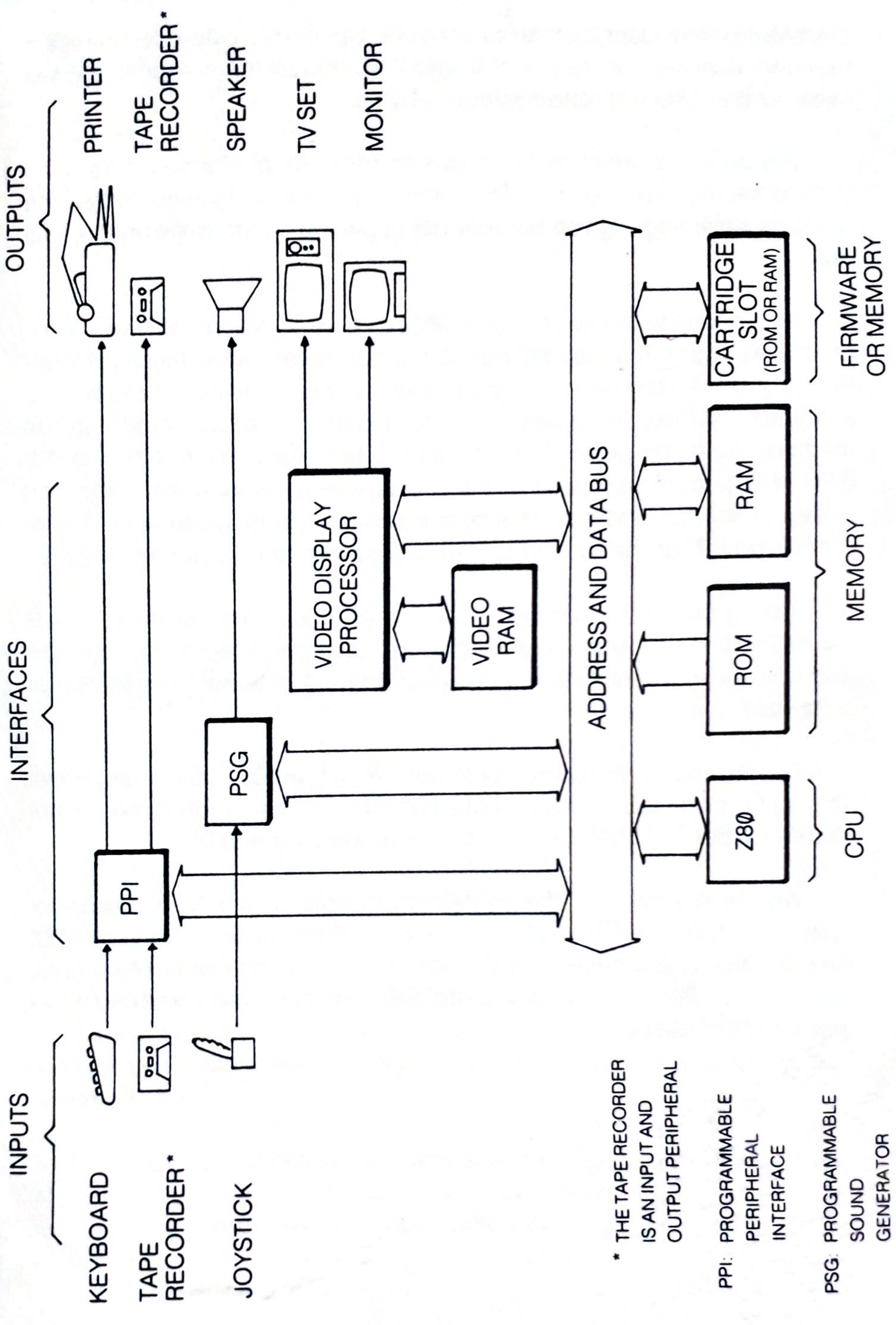


FIGURE 1.1 BLOCK SCHEMATIC OF MSX SYSTEM

the basic unit of data transfer around the CPU. This byte can represent numbers between the values of 0 and 255, and all the numbers that are used by the CPU are represented by bytes.

The data bus, which is the means of transferring information from the CPU to the memory, PSG or VDP, carries bytes around the computer, the bytes representing a given number being placed sequentially on the data bus.

How does the computer know what to do with a number on the data bus? Well, there is a second bus in the computer called the ADDRESS bus, which is 16 bits wide. This can carry numbers between 0 and 65535, and each of these numbers refers to a certain unique location in the memory of the computer. When it puts a byte of data on the data bus, the CPU also puts the address of the memory location to which it is to be written on the address bus. This address could refer to a memory location, part of the PPI or the VDP or sound generator of the computer system.

A third bus in the computer controls all these data transfers. This is called the control bus, and it informs devices connected to the address and data buses whether the CPU wants to read data from the device or write data to it.

The final bus in the computer is that linking the VDP to the Video RAM. This bus system, consisting as it does of a data and an address bus, is not usable by the CPU but can only be accessed by the VDP.

We have now met the main components of the MSX computer system. In the next Chapter, we'll examine the simpler aspects of MSX BASIC. Should you have difficulty with some of the concepts introduced, don't worry. Such ideas as expressions, variables and constants I will explain in Chapter 3.

2 The Core BASIC

In this chapter you will find quick descriptions of all the common BASIC commands. These commands are in no way specific to MSX BASIC, and so I have called the commands and statements listed in this Chapter the Core Statements. None of the commands listed in this Chapter relate to the excellent graphics or sound capabilities of the MSX computers; these features will be described in detail in later chapters. The role of this chapter is to provide a ready reference guide to the main BASIC statements that all programs use.

The first commands we will look at are those that are used in Direct Mode, that is, without any line numbers. They are principally commands to help us write programs, such as commands to list the program we've written so far.

AUTO n,m

This command generates line numbers automatically. It can be invoked by typing in the word, or by using the function key F2. The first line number generated is n, and subsequent ones are separated by the increment m. Thus

```
AUTO 10,10
```

will hence generate line numbers at 10,20,30 . . . etc. The highest line number that is allowable on the MSX machines is 65529, and if AUTO generates a line number higher than this, or if you type one in, the error message "Syntax Error" will be generated. If the line number generated by AUTO is already occupied by BASIC statements, then a "*" is printed after the line number to warn you of this fact. Typing RETURN at this point will preserve whatever is on line. AUTO can be exited by either typing CTRL-C or CTRL-STOP. AUTO n, will generate line numbers starting at line n with the last specified increment.

It is often useful to leave blank lines in programs to separate one part of the program from another. If you simply type in a line number followed by a space on the MSX system, then the line is not inserted. However, the ":" can be put at the beginning of a line and appears to cause no problems. Thus

```
100 :
```

will give a line blank except for the ":" at the line 100.

If you type in a line without spaces, e.g.

```
10REM
```

spaces will be inserted at the appropriate places in the line, giving 10 REM.

CONT

This command is short for Continue, and it resumes execution if the program is stopped by the use of CTRL-STOP. To restart the program, type in the command and press RETURN. If the program has finished execution altogether then CONT will have no effect. CONT cannot be used as a statement in a program line. Also, if the stop was caused by an error condition, CONT is not likely to be effective. Finally, CONT will not work if the program has been edited since CTRL-STOP was pressed.

DELETE n-m

This command allows us to delete blocks of lines from the program. n is the first line of the block to be deleted and m is the last line. DELETE n will just remove line n from the program. DELETE -m will delete all the program lines between line 0 and line m, including line m. Thus

```
DELETE -100
```

will delete all lines in the program from line 0 to line 100 inclusive. DELETE 100-, which we might expect to delete all lines from the program between line 100 and the end of the program, is not allowed. If the m parameter is less than the n parameter, as in

```
DELETE 300-200
```

then an error is generated. Also, if the lines referenced by the command do not exist, an error is generated. Delete can be used as part of a program line, but as soon as the delete operation has been completed, the computer stops executing the program and returns to Direct Mode.

LIST n-m

Allows you to look at the program you are writing. n and m are line numbers, n being the lowest line number of interest and m being the highest. If the command LIST is used without any parameters, then the whole of the program is listed to the screen. If it is apparent that the listing will disappear off the top of the screen before you have read it, then a single press of the STOP key will cause the listing to pause for a while. The listing can be restarted by pressing the STOP key a second time. This can be done as often as is required. If parameters are used, then the command works as follows.

LIST n	Lists the line, n
LIST n-	Lists from line n to the end of the program.
LIST -n	Lists from the start of the program to line n.
LIST n-m	Lists lines n to m of the program including lines n and m.

Listing is exited by CTRL-STOP. List can be a statement in a program, but as soon as the listing is completed the computer enters Direct Mode and program execution ceases.

LLIST

This command is similar to list but sends the listing to a printer if one is connected. If a printer is not connected and the command is issued, the computer will "hang-up" until CTRL-STOP is pressed. Control will then be returned to you with the message "Device I/O Error". The parameters that can be passed with this command are the same as those for the LIST instruction.

NEW

This command wipes a BASIC program from the memory of the computer. The command can be incorporated into a program line, but as it would wipe the program from memory as soon as it was executed, I don't see many applications for it in this role!

RENUM

RENUM renumbers programs, maintaining the sequence in which lines appear in the program but altering the actual line numbers that the statements possess. The syntax for the command is

```
RENUM new, old, increment
```

new

new is the first line number to be used in the new sequence.

old

old is the currently existing line number that is to be used as the starting place for the renumbering operation.

increment

This is the 'gap' to be left between adjacent program lines.

The parameters old and increment are not compulsory; if they are absent then the computer assumes a value of 10 for both parameters. Here below is an example of its use.

```
1 REM 1
2 REM 3
3 REM 4
```

Renumber this now, using RENUM 10. This gives the range

```
10 REM 1
20 REM 3
30 REM 4
```

Renumbering this with the command RENUM 10,10,100 will give

```
 10 REM 1
110 REM 3
210 REM 4
```

As with AUTO, RENUM cannot generate line numbers greater than 65529. All GOTO's and GOSUB's that are found in the program are renumbered to take the new line numbers into account. If a GOTO or GOSUB makes a reference to a line that does not contain any BASIC statements — i.e. a non-existent line — then the renumber goes ahead but an error is generated for each occurrence of a non-existent line number. The message generated is ''Undefined Line n in m'', where n is the non-existent line number and m is the line number of the statement that caused the problem.

RUN

Typing in this command will cause the computer to execute the BASIC program currently in RAM, starting at the lowest line number in the program. If the word RUN is followed by a line number that is in the program, then the computer will execute the BASIC program from that line.

TRON

This command can often prove to be invaluable in getting your programs working properly. It stands for Trace On, and after this command has been issued, a running program will print to the screen each line number as that line is executed. It is disabled by either NEW or TROFF. It can be used as part of a program line or in Direct Mode.

```
10 PRINT "Hello"
20 PRINT "Goodbye"
30 TRON
40 PRINT "MSX"
50 TROFF
```

will produce this when run.

```
Hello
Goodbye
[40] MSX
[50]
```

The next group of BASIC statements that we will consider are all concerned in some way with variables. We have already met the LET, DEF var and DIM statements, and so these will not be discussed here.

CLEAR n,m

Both n and m are optional parameters. CLEAR on its own will perform the following functions:

- i Clears all numeric variables to 0
- ii Sets all strings to be 'empty'.
- iii Closes any files that are still open.

If the n parameter is specified, then the CLEAR n command sets up STRING SPACE. On turning the computer on, the MSX BASIC allocates 200 bytes of memory for use by string variables. Should you require more space than this for string variables, then you must use the CLEAR n command to generate more string space. CLEAR 250 will thus reserve 250 bytes of RAM for string variables. To see what happens when you run out of string space, try the following commands.

```
CLEAR 0
A$ = "fred"
```

The m parameter specifies the highest location in the memory of the computer that is available to BASIC programs and variables. This allows the programmer to put an area of RAM 'out of bounds' to the BASIC system. This means that the area of RAM set apart in this way is a safe resting place for machine code programs.

ERASE array1, array2, . . .

This command allows you to selectively clear arrays from the variable space of the computer without affecting the values of other variables. The command is most useful for redimensioning already existing arrays, without getting the "Redimensioned Array" error. Simply use ERASE, and then redimension the arrays using DIM statements.

```
ERASE fr,ea
```

will erase the arrays fr and ea. The command can be used in program lines or in direct mode.

INPUT

One of the most important commands in BASIC. We have seen how we can assign values to variables in program lines so that when the program is executed the variables are given the value. However, what happens if we want to change the value of the variable while the program is running? Well, the INPUT command causes execution of the program to halt so that the user can type in numbers or strings that will be assigned to certain specified variables. The syntax for the input command is

```
INPUT "string constant"; variable list
```

The string constant is optional, and we will look at it in greater detail shortly. The variable list consists of a number of variable names separated by commas. Thus

```
INPUT A,S,D
```

will allow the user to assign values to the variables A, S and D. Note how we only require the semi-colon when we have used the string constant. Array type variables may also have values assigned to them using the INPUT statement, and string variables can also be part of the variable list. Thus

```
INPUT A$,A(1)
```

is quite legal. When the above command is encountered in a program, the computer prints a prompt to the screen in the form of a '?'. If the string constant is specified, as in

```
INPUT "Type in the first number"; A
```

then the string constant is printed to the screen followed by the '?'. The string constant is called a PROMPT STRING.

When the user comes to type in the response to an INPUT statement, then the type of value typed in by the user must correspond to the variable type expected by the INPUT statement. In the statement

```
INPUT A(1),A(2),B
```

the user is expected to type in 3 numbers; there are two ways in which this can be done. If the user were to type in just one number and then press RETURN, a second prompt, '??', would be generated. This type of prompt is generated every time that the computer is expecting another item of data to be typed in. The second method is to type in all the data items at once, separated by commas. Thus suitable data for the above INPUT statement could be typed in as shown below:

```
? 1,2,3 (RETURN)
```

If a string input is requested, the quotation marks are not required. If, however, you type in a string when a numeric value is expected, the error message "Redo from Start" will be issued, and you will have type in ALL the responses to that particular INPUT statement again, even if the previous values typed in were legal. If more data items are typed in than were expected by the INPUT statement, then the message "Extra Ignored" will be displayed. This means exactly what it says; the data items that were typed in that were surplus to requirements are simply disregarded.

An INPUT statement can be terminated by either CTRL-C or CTRL-STOP. The problem with both of these operations is that program execution is halted as well. The program can, however, be restarted by using the CONT command.

LINE INPUT "string constant"; string variable

Try the below program:

```
10 INPUT A$
20 PRINT A$
30 GOTO 10
```

Run it, and type in a few strings to confirm that it works. Now type in a string containing a comma, and see what happens. The "Extra Ignored" message is generated, and the PRINT statement at line 20 only prints up the string that was entered up to the comma. One way around this is to put the string that is typed in quotation marks, but this method has problems if you want to have quote marks as part of the string. LINE INPUT gets us around all these problems. Change Line 10 in the above to

```
10 LINE INPUT A$
```

and re-run the program. The first thing that you will notice is that the '?' prompt is not printed. Each character that you type in is added to A$. If you want a prompt to be put to the screen, then a string constant can be used, as shown below.

```
LINE INPUT "Only string variables"; a$
```

CTRL-C and CTRL-STOP have effects similar to those noticed with the INPUT statement.

READ, DATA and RESTORE

Imagine that we have written a program that requires the names of the months of the year to be held in a string array called year$(12). One way that we might put the months into the array is shown below.

```
10 DIM year$(12)
20 year$(1) ="January"
30 year$(2) ="February"
```

and so on. This works, but takes up quite a lot of program space to do 12 separate assignments to the array. A more efficient way of assigning the months to the array is to use DATA and READ statements to provide a means of reading the months from a list held in the computer program and assigning them to the elements of the array. The FOR-NEXT loop that we have used in this program will be explained shortly. At this time, suffice to say that we use it to read items from the data list and assign the data item read to a particular element of the array, the value of the subscript depending upon the value of the variable I.

```
10 DIM year $(12)
20 FOR I=1 TO 12
30 READ year$(I)
40 NEXT I
50 END
60 DATA January,February,March,April,May,
   June
70 DATA July,August,September,October, November
80 DATA December
```

Lines 60 to 80 provide a list of the months that we wish to be read into the array. Each begins with the word DATA and consists of a series of string constants separated from each other by commas. Of course, in another program, the DATA statement might be a list of numbers or a mixed list of both numbers and strings. However, in each case the values must be constants and must be separated from one another by commas. If a string constant in the list contains a comma as part of the string, then it must be enclosed in quotation marks. Quotation marks are also needed if trailing or leading spaces are to be included as part of the string. Normally, spaces like these, at the beginning or end of a string, are discarded when the computer makes use of the list. These DATA statements are not executed by the program. A series of DATA statements in a program, even if they are separated by lines containing other statements, are considered to be one long list of constants. The start of the list is always at the DATA statement with the lowest line number and the end of the list is at the end of the DATA statement with the highest line number.

To access the constants that are held in DATA statements we use a statement called READ. READ is followed by either a single variable name or a list of variable names that are separated by commas. Reading the DATA statements is done by the variable following the READ statement being assigned the next value in a DATA statement. If no other READ commands have been issued, for example, the first READ statement will assign to its variable the first constant in the first DATA statement in the program. In the example above, for the first time the READ command was executed, the constant ''January'' was read into the array.

To clarify matters, imagine that there is a pointer in the BASIC interpreter that, on running the program, points to the first item in the first DATA statement of the program. The first READ statement encountered will read this value into its variable, and then move the pointer on to point to the next item in the DATA statement. Any subsequent read operations will simply move the pointer towards the end of the list. If the pointer is at the end of a DATA statement, then after the next READ the pointer will point to the first item in the next DATA statement. If this next DATA statement does not exist, then the next READ will cause an ''Out of Data'' error message to be issued. If the pointer still points to data items, but no more READs occur, then the extra items are simply ignored.

What happens if we wish to get a data item from a DATA statement that has already been read? We cannot go backwards along the list using READ, but the command RESTORE enables us to reset the pointer to the first item in a DATA statement on a particular line in the program.

RESTORE when used alone will set the pointer to the first item in the first DATA statement in the program. A command such as

```
RESTORE 3000
```

will set the pointer to point at the first item in the DATA statement at line 3000. It is not possible to restore the pointer to a given data item within a line, only to the first item on the line.

MID$ (string expression 1,n,m) = string expression 2

MID$ gives us the opportunity to replace part of one string, string expression 1, with another string, string expression 2. String expression 1 CANNOT be a string constant. n is the position of the first character in expression 1 that will be replaced by the characters in expression 2. For example;

```
A$ = "QQQQQQQQ"
MID$ (A$, 2) ="ELP"
PRINT A$
```

will return "QELPQQQQ" as the value of A$. m is optional and refers to the number of characters from string 2 that you want to be put in string 1. The result of the replacement will be a string that is never any longer than string 1 was in the first place.

SWAP var1, var2

This command, as its name implies, swaps the values of var1 and var2. String variables or numeric variables can be included, but it is not possible to do a SWAP with one parameter a string variable and the other a numeric variable. Neither of the parameters of the swap command can be a constant.

★ ★ ★

The next group of commands we will look at are the BASIC commands that are not DEVICE SPECIFIC. These commands, which we will discuss in later chapters, are commands that work by sending information to or reading information from the other chips in the MSX computer; strictly speaking, you might say that commands such as PRINT and INPUT are device specific, as they obviously send data to the VDP in order for it to appear on the screen. However, these two commands are available in all versions of the BASIC language, and so I believe that I am justified in saying that they are not Device Specific commands. Examples of the latter are the SCREEN command, which operates in conjunction with the VDP, and the SOUND command, which works with the PSG.

DEF USRn = integer expression

This is used by advanced programmers to inform the computer of the start address of a piece of machine code. n is a digit between Ø and 9, and if it is omitted the value assumed by the computer is Ø. The integer expression should evaluate to give the start address of the piece of machine code, and so should give an integer in the range Ø to 65535. The addresses corresponding to different values of n can be redefined throughout the program as many times as is necessary. The command USRn is used in conjunction with this command and further details will be given there.

END

Nothing spectacular, this command simply causes program execution to halt, and the computer to return to Direct Mode.

ERROR integer expression

As you will no doubt have noticed, your computer lets you know when you have made an error in your program. The various error messages that are printed by the computer all have a number associated with them. Thus the error that causes the message "Out of Data" to be printed has the number 4. The command ERROR allows us to simulate any error that we so choose by using this code number. Thus issuing the command

```
ERROR 4
```

will cause the "Out of Data" message to be printed, just as if the error had been caused by trying to READ past the end of a DATA statement. If this statement were in a running program, then the program would stop running with this message. However, errors generated in this way can, like all errors, be trapped by the ON ERROR command, which we shall look at in Chapter 5. If you do some experiments with the ERROR command, you will find that some values of the integer expression generate the message "Unprintable Error". Such numbers can be used to generate "user defined" errors, which will cause control to pass to the ON ERROR routine if the program has one, or will cause the program to stop with the error message if it hasn't. The numbers that give this error are 23, 26-49 and

60-255. If you want to add your own error messages, then you are strongly advised to do so using the numbers between 60 and 255, as the others are reserved for future expansion of the MSX system.

ERR and ERL

These are system variables — i.e. variables whose value at any time can only be changed by the BASIC of the MSX system or by machine code or other advanced programming methods.

ERR holds the number of the last error that occurred, whether the error was caused by a program error or use of the ERROR statement. More will be said about its use when we consider ON ERROR in Chapter 5.

ERL gives the line number at which the last error occurred, again irrespective of what caused the error. If the error was generated in Direct Mode, then ERL has the value 65535.

FOR var = x TO y STEP z
NEXT var

This is the first CONTROL STRUCTURE that we have encountered in MSX BASIC. A control structure is a command that controls the flow of the program. Normally, a program begins running at the lowest line number and carries on through the program executing each line in turn. However, if we need to execute some lines repeatedly, or miss some lines out, we use what is called a control structure. The FOR-NEXT loop, as it is called, enables us to execute a block of program lines a given number of times. Other control structures are GOTO, IF . . . THEN . . . ELSE and GOSUB . . . RETURN. These will all be examined in this section of the book.

x is the initial value of the variable var, which is called the control variable of the loop. y is the final value, or limit value, that var will attain. z is optional, and x, y and z are all numeric expressions.

The program lines between the FOR and the corresponding NEXT are then executed repeatedly until the value of var exceeds the limit value.

After the statements between the FOR and NEXT have been executed, the variable var is incremented by either 1 or the value z if the STEP feature is present. Once the value of var exceeds the limit value, the statement immediately following the NEXT is executed. The loop

```
10 FOR I = 1 TO 10
20 PRINT I
30 NEXT I
```

will print the numbers between 1 and 10 to the screen. If we had STEP 2 on the FOR statement, then the numbers 1, 3, 5 . . . would be printed. Try experimenting with the commands to get used to them. If you want to go from a high value of x to a lower value of y then we simply have a negative STEP value. The values of x,y and z need not be integer, but in many programming applications they are. The variable name following the NEXT statement is not necessary, but it helps program readability if it is present.

We can have FOR . . . NEXT loops within other FOR . . . NEXT loops. This is called NESTING the loops. It is essential in these cases that the NEXT for the inner loop is encountered by the FOR of the inner loop before the NEXT of the outer loop is executed. Thus

```
FOR I = 1 TO 10
      FOR J = 1 TO 10
      ......
      NEXT J
NEXT I
```

is legal, whereas

```
FOR I = 1 TO 10
      FOR J = 1 TO 10
      ......
      NEXT I
NEXT J
```

will cause problems. The default type of the numeric variables used in FOR . . . NEXT loops as control variables is Double Precision. We very rarely require this type of accuracy in this application and so single precision or Integer can be used instead. This has two effects; space is saved when the variables are stored in memory, and the loops are executed more quickly when single precision or Integer type variables are used. Try the program below, with Double and Single Precision and Integer variable types for I. Details about the TIME function will be given later. Here we will just use it to get a relative time for each variable type.

```
10 DEFDBL I
20 TIME = 0
30 FOR I = 1 TO 200: NEXT I
40 PRINT TIME
```

The results that I obtained on the Sony HB-55 MSX machine were as follows:

Type of I	**TIME**
Double Precision	23
Single Precision	2Ø
Integer	1Ø

Omitting the I from the NEXT statement causes the times to be 19, 16 and 6 respectively. Putting the NEXT on a separate line gives times of 20, 17 and 7. Thus, if speed is required, the control variable of a FOR . . . NEXT loop should be Integer.

GOSUB and RETURN

In programs, we often have sequences of statements that are repeated in many places throughout the program. We can replace each of these sequences by a GOSUB n instruction, where n is the line number of the sequence of instructions that we wish to be executed at that point in the program. We thus only need one copy of the set of instructions to be kept in the program, and this copy is called a SUBROUTINE. A subroutine always ends in the command RETURN, which passes control of the program back to the statement following the GOSUB. The line number that is referenced by the GOSUB statement must be a numeric constant; unlike some other BASIC dialects, the line number must not be a variable or expression.

It is a good idea when writing programs to separate your subroutines from the main part of the program by an END command or a STOP command. Should a RETURN be executed without a corresponding GOSUB then an error will be generated. For this reason, I tend to keep all my subroutine definitions at the end of my programs, and I split them up by using REM statements (see later) to give the title and function of the sub-routine that follows it.

GOTO n

As we have already noted, a program is normally executed in the numeric order of the line numbers. A GOTO n statement will cause control of the program to pass to line n. Use them very carefully — a program that is full of GOTO statements is very difficult to read and understand. As with GOSUB statements, the line number specified must be a constant. If the line number specified does not exist, then an error will be generated. The command can be used for the purpose of running a program, or portion of a program without clearing the variables, by typing in GOTO n, where n is the number at which you wish program execution to start.

IF expression THEN statements ELSE statements
IF expression THEN GOTO nn ELSE statements

This control structure enables the computer to perform certain statements only if certain conditions are met. The expression is any BASIC expression that returns a 'true' or 'false' result. If the result of the expression, when evaluated, is true then the statements immediately after the THEN are executed. Otherwise the statements following the ELSE are executed. Thus, only one group of statements out of the two are executed whenever the line is executed. The IF . . . GOTO construct is a special case of the IF . . . THEN construct where the THEN is not required. In this case, the GOTO nn is executed if the expression evaluates to true. In the IF . . . THEN . . . ELSE statement, the statements after the THEN and ELSE can be replaced by line numbers if desired

```
100 IF I < 6 THEN 200 ELSE 300
```

which is the equivalent of the statement

```
100 IF I < 6 THEN GOTO 200 ELSE GOTO 300
```

IF . . . THEN . . . ELSE statements can be nested, each ELSE matching up with the nearest unmatched THEN. This can, however, get rather confusing, and whereever possible it is advisable to keep these statements on separate lines. If the line number that follows an ELSE, THEN or GOTO statement does not exist, then an error will be generated.

KEY, KEY LIST

One interesting feature of the MSX computers is that they have a series of keys which are called Function Keys. You will probably be aware that several BASIC words can be entered into the program or in Direct Mode simply by pressing the appropriate Key. For example, pressing F2 will activate the AUTO command, and F5 will RUN the BASIC program currently resident in your machine. However, it is also possible to change what these keys do, using the KEY n command, where n is a number in the range 1 to 10. Thus the commands

```
A$ = "PRINT"
KEY 1, A$
```

will cause the word PRINT to be printed every time Key 1 is pressed. This particular example will leave the cursor immediately after the word, so that you can type in more text. If you wish the command that you place in a function key to be executed immediately, then you can cause the computer to think that the RETURN key has been pressed when the function key has been pressed by the method below. CHR$ (13) simulates the pressing of the RETURN key.

```
KEY 1, "PRINT A" + CHR$ (13)
```

The string that we put into the function key must be less than 15 characters long.

The KEY LIST command lists the current contents of all the function keys.

Further insight into how the function keys can be used will be given in Chapter 6, when ON KEY will be discussed.

ON GOTO and ON GOSUB

Although these commands begin with the word ON, they are slightly different in the way in which they function to the ON commands that will be discussed in Chapter 5. These commands give us a way of transferring control to a line number depending upon the value of a BASIC variable or expression. The syntax is shown below.

```
ON expression GOTO line1, line2, line3 . . .
```

If the value of the expression is 3, for example, then the third line number in the list of line numbers after the GOTO would be jumped to. Thus in the program section

```
100 A = 2
110 ON A GOTO 200, 300, 400
```

the destination line number would be 300. If the expression evaluates to a non integer number, then the fractional part is simply disregarded. In the ON . . . GOSUB construct, the line numbers are the first line numbers of subroutines.

If the value returned by the evaluation of the expression is Ø or greater than the number of items in the line number list, but still less than 255, then the execution of the program continues with the next statement that is after the ON . . . GOTO or ON . . . GOSUB statement. If, however, the expression returns a result that is more than 255 or is negative, then an "Illegal Function Call" error is generated.

POKE address, integer expression

This command is of use when we need to directly alter a value held in a certain address in RAM The address in the syntax above is the address of the byte to be altered and the integer expression is the new value that is to be written to the location. The address should be between -32768 and +65535. If the value of the address is negative, then the machine will poke (address + 65535) with the new value. The integer expression must return a value between Ø and 255.

PRINT list of expressions

We've already used the PRINT command in a couple of the demonstration programs. It does as its name suggests — prints the value of an expression to the screen. The command PRINT when issued on its own, either in Direct Mode or as part of a program line, will cause a blank line to be printed to the display. PRINT followed by a numeric or string expression will print the value of the expression to the screen. MSX BASIC divides each line of the screen into PRINT ZONES. (Note. The PRINT command will not work in certain screen modes; these will be discussed when we examine the VDP in detail.) Each print zone is 14 characters long. Where a value is printed in relation to these zones depends upon the character used to separate expressions from one another in the expression list.

','	causes the next expression value to be printed at the beginning of the next print zone.
';'	causes the next expression to be printed immediately after the last one.

If a PRINT statement finishes with either of these characters, often called DELIMITERS, then the next PRINT statement will print its expressions in accordance with the above. If the list of expressions is too long to fit on one line, or if a single expression returns a value that is too long to fit on one line, then the values will be printed to the next line. The character '?' can be used instead of the word PRINT, as in

```
? "Hello"
```

which is the equivalent of PRINT "Hello".

PRINT is one of the most versatile commands that we have met so far. Because of the variety of ways we can print to the print zones on a given line, it is well worth playing around with the command, seeing exactly what you can do.

PRINT USING string expression; expression list

The PRINT USING command enables us to print strings or numbers to the display in accordance with a preset format. For example, it enables us to have numbers printed out to a set number of digits both before and after the decimal point — very useful if we are printing tables of data to the screen. The string expression in the syntax above is called a formatting string, and it contains certain non-alphanumeric characters.

Let's take a look at these formatting characters, first examining the ones that are used to format string expressions.

'!'

This character simply prints the first characters only of each string in the expression list, e.g.

```
PRINT USING "!"; "Hello"; "Goodbye"
```

will return

```
HG
```

as the result. Incorporation of numeric expressions in the expression list generates a "Type Mismatch" error.

& n spaces &

This formatting string consists of 2 ampersands separated by n spaces. 2+n characters from the string in the expression list will be printed to the screen. If you allow n to equal Ø then 2 characters from the string expresion will be printed. If the 2+n is longer than the string expression, then the string expression is printed to the display with trailing spaces.

```
PRINT USING "& &"; "CATTLE"
```

will return "CAT" to the display.

@

This character gives us a method of placing string variables in the middle of a string constant. The method of use is slightly different to the other two formatting characters that we have encountered, as there is no formatting string as such.

```
A$ = "MSX"
PRINT USING "This is an @ computer"; A$
```

Other characters are used to format numeric expressions. Let's now look at these in a similar fashion.

#

This character enables us to specify the numbers of digits we wish to have printed before and after the decimal point in a numeric expression. For example, "##.##" as a formatting string will specify 2 digits to be printed before the decimal point and 2 digits to be printed after the decimal point, e.g.

```
PRINT USING "##.##"; 10.2
```

will return

```
10.20
```

to the display. Replace the 10.2 with 100.7. This is printed as %100.70. The '%' indicates that there is an excess digit in the number, in this case the 1 in the hundreds column. Try other numbers, such as 0.37 or 1.37. In these cases a space is printed to the left of the first digit. A '+' sign at the beginning or end of the format string will print the sign of the number at the beginning or end of the number. For example,

```
PRINT USING "+##.##"; 1.3
```

will print the sign of the number to the left of the number.

A '-' character at the end of a formatting string will print the number to the appropriate number of digits with a trailing minus sign.

"**" added to the front of a formatting string for specifying the number of digits to be printed will print leading '*' characters if required instead of leading spaces, e.g.

```
PRINT USING "**.##";2.2
```

will print *2.20 to the display. Similarily,

```
PRINT USING "**#.##";10.3
```

will print *10.30. Note that these numbers, when formatted in this way, are rounded up or down according to their value so that the number is accurate to the number of digits shown.

¥¥

This is of minimal use to the European user. It is used in conjunction with the '#' characters, and represents 2 digit positions to the left of the decimal point, one of the character positions being occupied by a 'Y' character. Thus

```
PRINT USING "¥¥.##; 1.3
```

will print up ¥1.30 to the display.

','

The comma is quite useful in numeric formatting. Again, it is used in conjunction with the '#' characters. Placed to the left of the decimal point, it causes a comma to be printed to the left of every third digit to the left of the decimal point. For example,

```
PRINT USING "#####,.##";10000.2
```

will print 10,000.20 to the screen.

" ^^^^ "

Placed after the '#' characters in a numeric formatting string, it indicates that the number is to be printed to the display in exponential format.

```
PRINT USING "#.# ^^^^";7.4
```

will print 0.7 E+01.

There is one possible error that can turn up with numeric formatting strings. This is when we specify more than 24 digits to be printed. A formatting string can be a string variable. E.g. A$ = "##.##".

REM

REM is short for REMark — it allows you to place comments in the program that have no effect on the correct execution of the program.

```
100 REM This is a remark
```

The line can be jumped to by GOSUB or GOTO commands. Execution continues with the first statement after the REM. Any statements on the same line as the REM, i.e. after a ':', are ignored. The ' character can be used at the end of a line instead of REM.

STOP

This command simply causes the computer to cease execution of the BASIC program currently running. Control is passed back to Direct Mode. Unlike END, any open files will remain open. CONT will cause execution to continue.

INTRINSIC FUNCTIONS

We have already considered BASIC statements and commands. A function is a series of operations that is performed on BASIC variables and constants. More details will be given in the next Chapter, but here we will discuss the Intrinsic Functions. These functions are present in the MSX computers from the instant that the power is turned on. The programs for working the functions out are stored in the ROM as part of the BASIC interpreter. We will now look at them in alphabetical order.

ABS(n)

This function returns the ABSOLUTE value of the number n. This is the value of n irrespective of the sign of the number. Thus ABS (−10) is 10, and is hence greater than ABS (5).

ASC (n$)

This returns the ASCII code of n$.

ATN(n)

This returns the arc-tangent of the number n in radians. All trigonometric functions, such as SIN, TAN, COS will work in radians and to double precision. The result for this function is always between −PI/2 and +PI/2.

BIN$(n)

Returns a string representing the binary value of n. n is a numeric expression returning a result in the range −32768 to +65535. If n is a negative number, then the string returned represents the two's complement form of the number.

CDBL(n)

Converts n to double precision.

CHR$(n)

Returns the character that has the ASCII code n.

CINT(n)

Truncates n to an integer. n must be in the range − 32768 to + 32767.

COS(n)

Returns cosine of n in radians.

CSNG(n)

Converts n to single precision.

CSRLIN

Has no argument, but returns the current vertical co-ordinate of the screen cursor. Details of screen lay out will be given in the chapter on the VDP.

EXP(n)

Gives e to the power of n. n must be less than 145.1, or else an "Overflow" error is generated.

FIX(n)

Returns the integer form of n, but when used on negative numbers does not return the next lowest negative number, as does CINT.

FRE(Ø)

The argument here is a dummy; PRINT FRE(Ø) will return the number of bytes available to you for your program, etc.

FRE("")

Again a dummy argument, returns number of bytes of string space left.

HEX$(n)

Returns a string representing the Hexadecimal value of n. Same constraints apply to n as in BIN$.

INKEY$

This returns either a one character string representing a key pressed or an empty string if a key was not pressed. Any key pressed is not echoed to the display.

```
10 A$ = INKEY$
20 IF A$ = "" THEN GOTO 10 ' wait for key press
30 PRINT ASC (A$)
40 GOTO 10
```

This program prints out the ASCII code of the keys pressed. Note how if you press a function key with this program running a whole string of ASCII codes are generated.

INPUT$(n)

This function accepts n characters before allowing the computer to carry on executing the program. INPUT$(1) will accept one character, and the characters so accepted are not echoed to the screen. This function is quite useful as it allows us to write subroutines that respond to only certain key presses. For example, the routine below waits for the space bar to be pressed before moving on.

```
1000 REM Space Bar Routine
1010 PRINT "Press Space Bar to go on"
1020 G$ = INPUT$ (1)
1030 IF G$<>" " THEN GOTO 1020
1040 RETURN
```

INSTR(n, x$, z$)

This function searches the string x$ for occurrences of the string z$. If n is present, then x$ is searched from character n onwards. If absent, the whole of x$ is searched. n must be between 0 and 255. The function returns Ø if z$ was not found or if both strings were null. If z$ is found, then the function returns the position within x$ where z$ was found.

INT(n)

Returns the integer portion of n by simply discarding the fractional part.

LEFT$(x$,n)

Returns the leftmost n characters from x$, eg,

```
PRINT LEFT$ (''GOODBYE'', 3)
GOO
```

LEN(a$)

Returns the number of characters in the string a$. This includes non printing characters and spaces.

LOG(n)

Returns the natural logarithm of n. n must be greater than Ø.

LPOS(Ø)

Only used with a printer. It returns the current print head position.

MID$(a$,n,m)

This returns a string of m characters from a$ starting at character n. m can be omitted, and if this is done then all the characters to the right of character n will be returned.

OCT$(n)

Similar to BIN$(n), but string represents the Octal value of n.

PEEK(n)

Returns the value of the byte held at address n. n must be between -32768 and + 65535. See POKE for further details.

POS(Ø)

This returns the current horizontal position of the cursor on the screen. The argument is a dummy, and the leftmost column on the display is said to be column Ø.

RIGHT$(a$,n)

Returns the rightmost n characters of a$.

RND(n)

This generates a random number between Ø and 1. If n = Ø then the number generated is the same as the last one that was generated. If n > Ø, then the random number generator produces truly random numbers. To get random integers between Ø and 1Ø, for example, we can use a simple user-defined function.

```
1Ø DEF FNr(Q) = INT (RND(1)*11)
```

Note how we use 11 as a multiplier; this is because the INT function always rounds down, and so if we had 1Ø as a multiplier the number 1Ø would never be produced.

SGN(n)

Returns a number representing the sign of n. If n = Ø, then function returns zero. If n < Ø then function returns −1 and if n > Ø the function returns + 1.

SIN(n)

Returns sine of n.

SPACES(n)

Returns a string of n spaces. n must be between Ø and 255.

SPC(n)

Similar to SPACES(n), but can only be used with the statements PRINT or LPRINT.

SQR(n)

Returns the square root of n.

STRING$

This function returns a string of characters in a similar way to SPACES(n). STRING$(n,m) will return a string of n characters whose ASCII code is m. STRING$(n,a$) returns a string of n characters, the character being the first character of a$.

TAB(n)

Moves the place at which the next print statement will start printing to a position n on current line. If print position is already past position n then the function is ignored. It is used in conjunction with PRINT or LPRINT.

TAN(n)

Returns tangent of n.

TIME

This system variable gives us access to an internal timer on the MSX computers. As previously mentioned, the variable TIME is incremented 50 times per second on MSX computers equipped with PAL TV displays (i.e. UK models) and 60 times a second on MSX computers with NTSC TV displays. Setting TIME to zero will reset the clock. TIME is not incremented during tape operations, but retains the value it had before the tape operation commenced.

USRn(m)

Used to call a machine code routine, either in the ROM or one of the programmers own devising. n is a digit between Ø and 9, and indicates to the computer the address of the routine wanted. The address will have already been assigned to the USRn call by the DEF USR statement. m is the argument of the function and will be passed over to the machine code routine.

VAL(a$)

This returns the numeric value of a$, e.g.

```
A$ = "12"
PRINT VAL (A$)
12
```

VARPTR

This function is really aimed at advanced programmers, and offers a method of finding out where in memory variables are stored. PRINT VARPTR(n) will return the address of the first byte associated with variable n. If n has not been assigned, then an error is generated. The address returned will be between -32768 and 32767. If the address is negative, simply add 65536

```
PRINT VARPTR (a(Ø))
```

will return the start address in RAM of the first byte of element Ø of array a. Note that the address of the array will move in memory as other variables are assigned. So, whenever this information is required, use VARPTR again.

VARPTR (#file number) returns the address of the first byte of the file control block. This is only of value if you want to directly access the file control block. Be very careful doing this, as it is possible to thoroughly confuse the MSX tape system!

That sums up the simple BASIC statements that are available to the MSX BASIC programmer. In the next Chapter, we'll look in some detail at the raw material on which these statements and functions work — variables, constants and expressions.

3 Data Structures and Variables

All computer programs process information in some form or another; the information acted upon could be the name and address of someone, the number of days in the year, or the position of a Space Invader in a video game or any one of an infinite number of things. However, before the computer can process the data it must be represented in the computer in a suitable form. The ways in which the data is stored in the computer is called a DATA STRUCTURE. We'll now look at these in greater detail. Those of you interested in number systems in general may care to look at the relevant Appendix. The first data structure we'll look at is the character, because within the computer, a character can be represented in a single byte of memory.

Characters

How is non-numeric data dealt with by a computer, which, after all, is a mainly numeric machine? Textual data, such as the letter "A", is stored in the computer as a number between Ø and 255. This will, you will note, fit into a byte. Each character on the computer keyboard has a numeric code associated with it, and the most common method of coding characters is to use the ASCII code. ASCII is an acronymn for American Standard Code for Information Interchange. This code is utilised by the MSX machines, and in it the letter "A" is represented by the number 65. A lower case "a" has the code 97. Other characters have different ASCII codes, and to examine the ASCII codes associated with different characters we can use a BASIC function called ASC( ). Typing in

```
PRINT ASC("B")
```

and pressing RETURN will give the result 66. There is a function that performs the reverse of this operation; given the ASCII code of a character, this function, CHR$( ), will print the character. So,

```
PRINT CHR$(66)
```

will print the letter "B". Some characters, such as the character with code 1, will not print a character to the screen; these are known as NON PRINTING characters. Other characters will do some strange things when printed; try printing character 7 and character 12.

Strings

This book is made up of strings. A string is a collection of characters, and strings are often found to end in character 13.

Constants

A constant in a program is a value that does not change as the program runs. There can be either numeric or string variables, 1.234 and "Hello" being examples. A string constant may be up to 255 characters long. There are 6 ways in which the MSX computers can represent numeric constants, so let's have a look at them. A numeric constant can be either a positive or a negative number.

Integer Constants

An integer constant in MSX BASIC can have a value between −32768 and +32767. Obviously, integer constants don't contain decimal points.

Fixed Point Constants

These are numbers containing a decimal point.

Floating Point Constants

These are positive or negative numbers that are represented in the exponential format, i.e.

1.234 E+n

Here, n is called the exponent and the number to the left of the E is the mantissa. Floating point constants can be in the range 1ØE−64 and 1ØE+63.

Hexadecimal Constants

These are hexadecimal numbers, and they are prefixed by the &H characters.

Octal Constants

Octal constants are prefixed by &O or just &.

Binary Constants

These are prefixed by &B.

Numeric constants can be either single or double precision, single ones being represented within the machine to 6 digit accuracy and double ones to 14 digit accuracy. Unless you specify otherwise, the constant will always be represented to double precision. Any number in exponential form, however, will normally be treated as single precision. If you require a double precision number to be represented in exponential format, then the letter "E" is replaced by the letter "D". Constants can also be put into single precision by following the number with a "!".

Thus we've got many ways or representing constants in MSX BASIC. So far, however, the numbers retain the same value throughout the running of the program. What would be useful would be a means of allowing us to represent a number in some way that enabled us to change its value as the program progressed. This is where the concept of the variable becomes useful. A VARIABLE is best imagined as a series of bytes in memory which the computer can refer to by a name, the VARIABLE NAME. The series of bytes represents a string or a number. The variable name is thus used by the programmer to access the number stored in the variable. When we give a value to the variable, we say that we are ASSIGNING a value to the variable; to do this we can use the LET statement in BASIC, or we can just use the "=" sign;

```
LET A=100
    A=100
```

Both of the above statements assign a value of 100 to the variable A. Each variable type requires a certain amount of space to store the number in. The table below shows this.

Type	**No. of Bytes**
Integer	2
Single Precision	4
Double Precision	8
String	3 + 1 per character

Variable Names

A variable name is a collection of alpha-numeric characters (those characters that are either letters or numbers) that form a unique identifier for a particular variable. Names can occasionally include "!", "#", "$" or "%" as the last character of the name but this last character has a special meaning. They indicate whether the variable is single or double precision, string or integer. The characters are called TYPE DECLARATION characters, and are as follows:

Character	**Type**
%	Integer
$	String
!	Single Precision
#	Double Precision

If we don't give a variable name a type definition character, then the computer will usually take that variable to be a double precision variable.

Making up Names

Christening variables is quite easy; a variable name can be of any length, but only the first two characters of the name are considered by the computer, not counting the type declaration character. Also, a variable name must begin with a letter; should it start with a number, the computer will incorporate the variable name as a line in the program. Thus A1 is legal, but 1A is not. To see the effect of only the first two characters being significant, type this in, following each line with RETURN.

```
NEW
THISTLE=1
THROUGH=20
PRINT THROUGH, THISTLE
```

You will hopefully get 20 and 20 printed to the screen, thus indicating that the computer cannot differentiate between the two variable names. As soon as the computer has matched the first two characters with a variable that it knows exists, then it does not check any more. This can obviously cause problems if we are not careful. A further point to note is that variable names cannot contain any names of BASIC functions, commands or statements, and it cannot differentiate between upper and lower case letters. For example, "sprint" is an illegal variable name because it contains the word "print" which the computer will read as "PRINT". For similar reasons, SINK and DATA are illegal variable names because the first name contains SIN and the second DATA. A variable name must not start with the letters FN, as if it does the computer thinks that you are referring to a user-defined function. Thus FNF is an illegal variable name, as the machine thinks you are referring to a function called F. Don't worry about this sudden introduction of the user defined function; it will be explained soon.

The type declaration characters have been previously mentioned; z# and z1 are both double precision variables, z% is an integer variable and z$ is a string variable. However, we can also declare the type of variable using some statements that I call DEF var statements. There are 4 of these, DEFINT, DEFDBL, DEFSTR and DEFSNG, each of which is followed by either a single letter or two letters separated by a "—".

Thus DEFINT a will define all variables that begin with the letter a to be integer variables. In a similar fashion, DEFDBL would define the variables to be double precision, DEFSNG to be single precision and DEFSTR would define the variables to be string types. One point to note here is that the subsequent use of a type declaration character will overrule the DEF var statements issued. As an example, let us issue a DEFINT a command, and then assign the variable a#=1.234. If we were to print a# out then we would find that it was a double precision variable, as we might expect from the use of the "#" sign. However, a second variable, a1, would still be an integer variable. Try the program overleaf.

```
10 DEFINT A
20 A=1.2346
30 A#=1.234567
40 A!=1.2345
50 PRINT A,A!,A#
```

Run the program, and note how the variables that have type declaration characters are treated as different variables to the one without the declaration character.

The statement DEFINT A-Z will cause all the variables available to be integer unless otherwise said. Any variables that you want to be string, double or single precision must be specified by the use of type declaration characters. In a similar fashion, DEFINT I-K will define all variables beginning with the letters I, J or K to be integers. DEFSTR can give strange results to the unwary; variables will be string variables without the need for a "$" sign. The statement A="apple" would normally generate an error message — "Type Mismatch". However, after a DEFSTR A statement, the statement A="apple" is legal but the more usual A=1.234 is not! Again, use of the type declaration characters will overrule the DEFSTR statement.

Array Variables

An array is a data structure that is available to the MSX programmer. However, whereas the data structures we have previously discussed are collections of bytes, an array is a collection of numbers or strings which can be accessed under the same name. An individual item stored in an array is called an ELEMENT, and all the elements of an array will hold data of the same type. When you turn on your computer, the machine will allow you to use arrays with up to 11 elements in them, numbered 0 to 9, without having to inform the computer that you wish to use arrays. If you wish to have more elements than this, then the array will have to be DIMENSIONED using the BASIC DIM statement. Thus DIM A(20) will dimension an array called A to have 21 elements, numbered 0 to 20.

When we first dimension an array, each element has the value 0 if it is a numeric array and empty string if it is a string array. Assigning values to an element of an array is quite easy.

```
A(1)=1.234
A(2)=3
LET A(4)=23
```

A(Ø,4)	A(1,4)	A(2,4)	A(3,4)	A(4,4)
A(Ø,3)	A(1,3)	A(2,3)	A(3,3)	A(4,3)
A(Ø,2)	A(1,2)	A(2,2)	A(3,2)	A(4,2)
A(Ø,1)	A(1,1)	A(2,1)	A(3,1)	A(4,1)
A(Ø,Ø)	A(1,Ø)	A(2,Ø)	A(3,Ø)	A(4,Ø)

FIGURE 3.1 REPRESENTATION OF ARRAY A: DIM A (5,5)

These statements will all assign the appropriate values to various elements of array A. Arrays are still under the influence of the DEF var statement, and they can also have type declaration characters. An array can be one dimensional, like the example array A, or can have several dimensions, such as B(1Ø,6). The best way to view the multi-dimensional array is to think of it as a collection of boxes arranged on a grid. The different boxes are the various elements of the array, and Figure 3.1 shows a diagrammatic representation of part of such an array. Should you attempt to access an element of an array which does not exist, such as A(99) when we've only dimensioned A to 30 elements, then we get the "Subscript out of Range" error. The number used to access an element of an array is called the SUBSCRIPT. The subscript can be a constant, variable or expression, and the "Subscript out of Range" error often occurs when an expression returns a too high value. The maximum number of dimensions that an array can have is 255, but the number of elements is only really limited by the amount of RAM in your machine.

Changing Types

Try the following:

```
A="fred"
```

You'll get a "Type Mismatch" error — you can't put a string into a numeric variable! However, MSX BASIC will allow you to convert one type of number to another. If you set a variable of one type to a value held in a variable of a different type, then obviously the variable will assume the value and the value will be held in the variable according to the variable type. For example,

```
A%=1.23456
PRINT A%
```

will print 1. The real number has been converted to an integer by this act. The fractional part is discarded, and no attempt is made to round the number up or down. Thus, in double precision work, such a value when assigned to a single precision variable will only be represented to 6 digits. During the evaluation of expressions, the degree of precision applied to the result is the highest degree of precision possessed by a variable or constant in the expression.

EXPRESSIONS IN MSX BASIC

In everyday life, the sum 2+3 is a simple expression. We carry out the addition, a process known as EVALUATION, and produce a result, or VALUE. In this case, the result would be 5. In technical terms, we describe an expression as a collection of numbers, constants, variables and operators that can be evaluated to return a result. It is not compulsory for an expression to contain all of the above.

```
1+2+3+4
```

is an expression, as is

```
A+B+1+2
```

However, most expressions do contain at least one operator. An OPERATOR can alter the value of a variable or constant by performing some arithmetical or logical operation on it.

In the expression

```
1+2
```

the operator is "+". This is an arithmetical operator, and in the expression

```
1+ASC("A")
```

the ASC( ) is said to be a functional operator. In total, there are 4 main families of operators available to the MSC programmer. These are:

i Arithmetic Operators
ii Relational Operators
iii Logical Operators
iv Functional Operators

The most obvious thing to do now is to look at each family in greater detail.

Let's start with the ones that we're already familiar with — the Arithmetic Operators.

Arithmetic Operators

There are 8 different arithmetic operators available in MSX BASIC. These are exponentiation (∧), Negation (—), Multiplication (*), Division (/), Addition and Subtraction, Integer Division (¥) and Modulus Arithmetic. We'll take a look at the operators here that are new to us shortly. The computer will evaluate expressions according to a sequence of rules. For example, the expression

A+B*C

can be evaluated in 2 different ways, as either (A+B)*C or as A+(B*C). The computer will evaluate the expression as written in the second of these examples. The small expression in parentheses will, however, be evaluated according to these rules. These rules are called the rules of PRECEDENCE. The multiplication operator has a higher precedence than the addition operator. The figure below shows the order of operator precedence.

High Precedence ↑ **Low Precedence**

Operator

Exponentiation
Negation
Multiplication/Floating Point Division
Integer Division
Modulus Arithmetic
Addition/Substraction

Thus in the expression,

1+2∧2

the 2∧2 will be evaluated first, giving 4, and then the 1 will be added, giving a final result of 5. However, what if we want to evaluate the 1+2 before the exponentiation operation? Well, we use brackets, as we've seen before. To get the expression evaluated in the way we want, we'd write

(1+2)∧2

This will return the result 9. The operations within brackets are performed first, but within a set of brackets the rules of precedence still operate. Thus in long expressions in parentheses, we often bracket parts of the expression to determine what order the expression is evaluated in. Such brackets within brackets are called NESTED PARENTHESES. One important point to remember about nesting brackets like this is that each opening bracket (( ) is matched by a closing bracket ( )). If this is not adhered too then an error will be generated by the computer. This error, however, will NOT be the message "Missing Bracket"! The usual message obtained is "Syntax Error". Whilst on the subject of errors, should you place an operator in the expression and not follow it with a constant, variable or other expression, then the message "Missing Operand" will be generated, an OPERAND being anything that an operator works on.

Integer Division

Normal division usually returns a real number. However, integer division simply discards the fractional portion of the result. Thus,

7¥2=3 and not 3.5

The "¥" symbol indicates Integer Division. There are a couple of points to watch here; before the division is performed, the computer converts the operands to integers, which must be in the range −32768 to +32767. The result is also truncated to an integer. Secondly, don't try and get the computer to divide by zero — it's impossible and the computer knows that, even if your program thinks otherwise! Due to the conversion to integers, the expression

30.1 ¥ 01.4

will be evaluated as 30 ¥ 0. The division by zero error will also occur in real division.

Modulus Arithmetic

When I went to primary school, which wasn't too long ago, I was taught modulus arithmetic. We didn't call it that; to us it was remainder division.

10 MOD 4=2

The computer evaluates the expression above as 10 divided by 4 equals two remainder two, and it is the remainder that is returned as the value of the expression. Similarily,

```
10 MOD 5=0
```

Whilst on the subject of arithmetic operators, it is worth examining a couple of errors that can be generated. The first is "Overflow", where the result of your calculations is too great for the computer to handle. The second is "Type Mismatch", where you've attempted to assign a number to a string variable. I usually do this after issuing a DEFSTR statement, as I tend to assume that the type of a variable that has no type declaration character is numeric!

Relational Operators

Nothing to do with Aunts working in telephone exchanges; they are used to compare 2 values, expressions, variables or constants, and either −1 or 0 is returned as a result. In these cases, −1 is called TRUE, and 0 is called FALSE. The result of an operation of this sort may then be used to control the flow of a program, using the BASIC IF statement that we will soon encounter. There are 6 relational operators in MSX BASIC, and these are shown below.

Symbol	**Operator**
=	Equality
<>	Non Equality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Let's see a couple of examples of the use of Relational Operators. A true result yields −1; and a false result 0.

```
PRINT (1=1)
```

will return −1, as it is true that 1=1. The expression

```
PRINT (1=2)
```

will return the value 0, or False. The more complex expression

```
PRINT (A-B)=1
```

will only return a True value if the expression (A–B) returns a value of 1.

You can assign variables using relational operators; obviously, though, the values assigned to these variables will be either Ø or –1.

Thus the expression

```
A=(1=2)
```

will assign the value Ø to the variable A. When we combine arithmetic operators and relational operators, the arithmetic expressions are evaluated first.

Logical Operators

These are slightly more involved than previous operators, and can cause confusion. However, when used correctly they are powerful programming tools. So, here we go. The most obvious way of using logical operators is to link together relational expressions, i.e. expressions containing relational operators. The logical operators available in MSX are AND, OR, NOT, EQV, XOR and IMP. The most commonly used of these for linking relational expressions are AND, OR and NOT, and so we'll look at these three first in this role. When used in this way, the expression containing the logical and relational operators will return a true or false value. The expression

```
PRINT (A=1) AND (B=2)
```

will return a true value only when both A=1 AND B=2. A further example is

```
PRINT (A$="FRED") AND (B$ = "BLOGGS")
```

Here, A$ must contain "FRED" and B$ must contain "BLOGGS". The expression

```
PRINT NOT (P)
```

will return a true value if P=Ø and a false value otherwise. As Ø is the value used to represent "False", we can see that NOT False is True.

These two operators, and OR, are often employed when our program has to make a decision based on several different conditions being met.

Logical expressions of the sort shown above can be included in IF statements to help programs flow correctly.

The second use of these operators is to test a byte for a particular pattern of bits. This is known as BITWISE operation, and it is this aspect of the use of logical operators that can give rise to problems. First of all, let's see how the logical operators function in a bitwise fashion on single bits. The results of the operations on various combinations of bits are shown in the following TRUTH TABLES.

NOT

A	NOT A
Ø	1
1	Ø

AND

A	B	A AND B
Ø	Ø	Ø
Ø	1	Ø
1	Ø	Ø
1	1	1

OR

A	B	A OR B
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	1

XOR

A	B	A XOR B
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	Ø

EQV

A	B	A EQV C
Ø	Ø	1
Ø	1	Ø
1	Ø	Ø
1	1	1

IMP

A	B	A IMP B
Ø	Ø	1
Ø	1	1
1	Ø	Ø
1	1	1

Let's explain a few things before we go on. XOR is a contraction of the phrase "Exclusive OR". This function only returns a 1 if either A OR B is of value 1 but not if they both are of value 1. Treat EQV as a shorthand form of writing Equivalence; a 1 is returned only if A=B. The IMP function I have christened Importance, as in the expression A IMP B, a 1 is returned only if the B value is greater than or equal to the A value. How can we apply this to numbers or variables? Well, remember how numbers are represented as byte sequences within the computer? The integers are stored as 16 bit numbers, thus occupying two bytes. The leftmost bit of the binary representation of an MSX integer represents the sign of the number, whether it is positive or negative. This is why the MSX integers can only have values between −32768 and +32767. Both numbers that are to be operated on by the bitwise logical operators must be integers of this type. Thus any expressions to be used in a bitwise operation must return a value in this range. Let's try a bitwise ANDing of 2 numbers, say 8 and 3.

```
PRINT 8 AND 3
```

will return the value Ø. To see how this situation arises, let's convert 8 and 3 to their binary equivalents and examine the bit patterns.

```
8 ØØØØØØØØØØØØ1ØØØ
3 ØØØØØØØØØØØØØØ11
```

Now apply the truth table to the binary numbers; we find that we've got no 1's in common positions in the numbers, and thus we get the result Ø returned. To make it more obvious, try the operation with binary numbers.

```
PRINT &B0001000 AND &B00000011
```

We can now see the bit patterns directly. In a similar fashion, the bitwise OR command can be employed.

```
PRINT 8 OR 3
```

will return 11 as the result. Work through this using the bitwise OR truth table.

When we use negative numbers in bitwise operations, we have to be careful due to the fact that negative integers in MSX BASIC are represented in two's complement notation. Thus −1 in binary is represented as 1111111111111111. Thus

```
PRINT −1 AND 2
```

returns 2 as a result. One use of the bitwise operators is to modify the values of certain **bits** within bytes without altering the other bits.

Functional Operators

A function is a defined set of operations that can be applied to an operand or expression. There are two types of function supported by MSX BASIC. The first major type of function is the INTRINSIC FUNCTION — functions present in the computer from the moment the machine is turned on. They include such things as SIN(n), ASC( ) and CHR$(n). The value n here can be an operand or expression and is called the argument of the function. For the ASC( ) function, the argument must be a character. The argument is said to be 'passed' to the function, and the function is said to return a value. Should you try and pass a numeric argument to a function when a character or string argument is required by the function, then a "Type Mismatch" error will occur. This error will be generated by the statement

```
A=SIN ("hello")
```

The argument of a function can be another function, provided that the latter supplies an argument that is of the correct type. Thus

```
PRINT CHR$ (ASC.1 ("A"))
```

will print the letter "A" to the screen.

The second class of function supported by MSX BASIC is the User Defined Function. These are functions written by the programmer to perform a specific task. Let's see how we can define our own functions.

Defining Functions

Before a user-defined function can be used, the computer must be provided with a function definition, which is simply a list of statements that perform the function. A User Defined Function can use intrinsic functions in its definition, as we shall soon see.

Defining a function is done with the DEF FN statement. This is an abbreviation of **Def**ine **Fun**ction, and the statement tells the BASIC interpreter that the following line is a function definition. Function definitions must always be executed before the function is called. The following is an example of a function definition.

```
10 DEF FNtest 1 (a,b,c$)=SIN(a) + SIN(b)
```

Here, test is the function name, and the function simply returns as a result the sum of the sines of the two numbers a and b. a,b and c$ are collectively the parameters of the function, and together are called the parameter list of the function. Each item in the list is separated from its neighbour by a comma. These parameters are just ordinary operands; if they are variables, they are named in accordance with the rules governing the naming of variables.The function name is named in the same way as variables are. It is legal to have a function that has the same name as a variable, as the computer treats variable names and function names differently.

The expression to the right of the ' = ' in the function definition can only be one line long;

```
10 DEF FNtest 1 (a,b)=SIN(a)
20 + SIN(b)
```

will generate a "Syntax Error" message at line 20. A function called test will have been defined, but it will only return the SIN of the number a. Variable names that appear in this expression serve only to define the function, and changes in their values are not reflected in changes of the value of variables with the same name outside the definition. If your function has a parameter list, then the variables listed therein may be included as part of the expression, but it is not compulsory.

A User Defined Function is called with the FN statement. Remember that before a function can be called, the corresponding DEF FN statement must have been executed. To call the function that we defined above, we would type in something like this.

```
100 PRINT FNtest 1 (1,2"Hello")
```

It is vital that the number of items passed to the function is the same as the number of parameters in the parameter list that was set up in the DEF FN statement. Also, the types of the values passed to the function when it is called must be the same as the types that were used in the parameter list when the function was defined. If this is not done, then a "Type Mismatch" error will be generated. Parameters passed to a function in this way are passed over on a one-to-one basis, the value of the variables or constants in the call being passed to the corresponding variable in the function parameter list. Thus in the example we have just seen, the variable a would be assigned a value of 1, b would be assigned a value of 2 and c$ would get the string "Hello" assigned to it. If a variable is used in the expression, but is not in the parameter list, then on evaluation of the expression the value used is the current value of the variable in the program. The ability to pass parameters over to the function in this way, and to get results returned, makes the User Defined Function a powerful programming tool. If we were to use subroutines (see the next chapter), then we would need to set the variables used in the expression to the appropriate values before the subroutine was called.

Thus we would have a piece of code like:

```
100 a=1
110 b=2
120 c$="Hello"
130 GOSUB 1000
```

This assumes that the code at line 1000 is the same code as we put in the function definition that we used a little while ago. The equivalent function call is much more obvious.

```
100 result = FNtest 1 (1,2,"Hello")
```

If you attempt to use a function call before you have defined it using the DEF FN statement, then the error message "Undefined User Function" is issued. Functions will occur in various places in this book, so you will have several examples to look at.

One final point; the DEF FN statement MUST be issued from within a program. Attempting it from Direct Mode will generate the "Illegal Direct" error message, which indicates that you have executed a command from Direct Mode that should only be executed from within a program line.

STRING OPERATIONS

Although we've mentioned string variables, we haven't yet discussed what operations we can do on them. As you cannot do much arithmetic on strings or characters, the only arithmetical operator that we use with strings is '+'.

However, the operator does not perform the addition operation, but joins the strings together. This process is called CONCATENATION.

```
a$ = "ABC"
b$ = "DEF"
PRINT a$ + b$
```

The string "ABCDEF" is printed to the screen. We can thus make one long string from 2 or more short ones.

```
C$ = A$ + B$
```

is quite legal. We can, however, use the relational operators on strings.

When the computer compares two strings using the relational operators, it does so on a character by character basis, comparing the ASCII codes of each character in one string with the corresponding characters codes in the other string. If all the codes are the same then the strings are equal. However, should a code be found in one string that is less than the corresponding code in the other string, then the string with the lower code is said to be less than the other string. For example,

```
"AA"<"AB"
```

Any leading or trailing spaces are also examined, and so

```
"AA"<"AA"
```

This comparison is fine until we come to the situation where one string is shorter than the other string; what happens now? Well, the shorter of the two strings is said to be less than the longer one. So,

```
"AB"<"ABC"
```

As in the case of relational operators applied to numbers, true and false values are returned as the result of a relational comparison of two strings. Logical operators such as AND can be employed to perform more complex relational functions such as

```
PRINT (A$="1" AND B$="2")
```

This will only return a true value if A$="1" and B$="2". We can also use string comparisons to control the flow of the computer program in conjunction with the IF . . . THEN . . . ELSE structure.

Obviously, bitwise logical operations are not possible, but a bitwise operation can be performed on the ASCII code of a character.

We are now ready to move on to look at the rest of MSX BASIC. Don't be afraid to experiment with programming your machine. This is the best way to learn about how you can get the best results from the computer. If examples are suggested in the text, then type them in; if you can then see a way of doing a particular job that is different to the one I have used then experiment; your method may well be better than mine!

4

Cassette Tape Storage

The RAM of your MSX computer will lose all record of your program as soon as the power is turned off or you type the command NEW. It is obvious, therefore, that we need a method of making a permanent copy of the program that we can use to store the program indefinitely. We do this by saving a copy of the program on to cassette tape. As well as being able to save BASIC programs to tape, we can also save variables, arrays or blocks of bytes.

When we transfer data to tape from the computer, it is recorded on the tape as a series of tones, each tone representing a bit of a byte. Thus each byte on the tape is represented by 8 tones, the pitch of the tone telling the computer whether the bit had a 1 or Ø value. This coding of data into different audio tones is called Frequency Shift Keying, or FSK for short. If we could "see" the audio tones on the tape, we would see Figure 4.1.

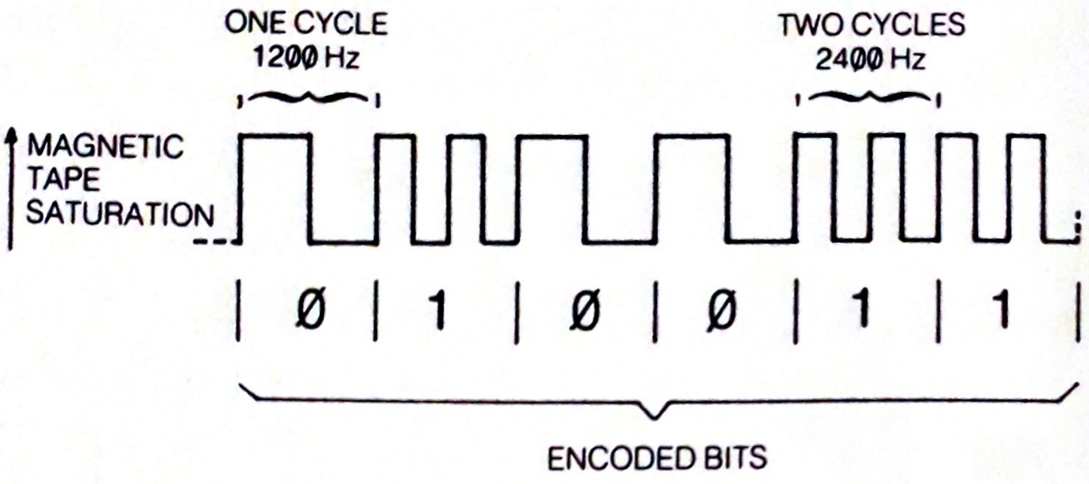


FIGURE 4.1 DATA RECORDED AT 1200 BAUD (1200 BIT/SEC)

The actual tones used to represent 1's or Ø's depend upon the speed at which the computer is saving data to the tape. The rate of data transfer is known as the BAUD RATE, and is a rough measure of the number of bits sent per second. MSX computers can read or write data at either 1200 or 2400 baud.

At 1200 baud, a 'Ø' is represented on the tape by 1 cycle of a 1200 Hz tone. (1 Hz is 1 cycle per second.)

At this baud rate a '1' is represented by 2 cycles of a 2400 Hz tone. 1200 baud is the usual rate of data transfer adopted by the MSX computer, and is reliable with almost all computers.

The 2400 baud rate will enable data transfer at twice the speed of the 1200 baud rate, but is not very reliable with some tape recorders. A 'Ø' is now represented by 1 cycle of a 2400 Hz tone and a '1' is represented by 2 cycles of a 4800 Hz tone.

The baud rate is selectable by the programmer and is normally at 1200 baud. The rate can be changed using the CSAVE command, as we shall soon see, or by the SCREEN command, as we shall see in a later chapter. On reading the data from the tape, the computer can automatically decide which baud rate to use to read the tape.

The computer, as well as sending data to the tape recorder and reading it back in, can control the motor of the tape recorder if the recorder has a remote socket. Thus the computer can turn the cassette motor on, send data to the tape, and then turn the motor off. The BASIC commands

MOTOR ON and MOTOR OFF control the tape recorder motor. MOTOR ON will turn the motor on, and MOTOR OFF will turn it off. Typing in MOTOR on its own will cause the motor to turn off if it is on, and on if it is off. This is called TOGGLING. The motor is controlled by a relay in the computer, and when you issue a MOTOR command, or use commands that use the tape recorder, this relay can be heard ''clicking'' whenever it turns the motor on or off.

Saving Programs

The first thing most people wish to use a tape recorder for with a computer is to save their program; on the MSX computers there are two ways in which BASIC programs can be stored on tape. So, before we go any further, let's examine these two methods. A block of information that is saved on cassette tape from the computer is called a FILE. A program can be stored on tape as a file of ASCII characters, in which case the BASIC command PRINT will be saved as P,R,I,N,T; that is, as individual letters, or as a TOKENIZED file, in which case the BASIC command PRINT is stored as a single number. This number is said to be the TOKEN of the command, and each BASIC command or function has a token. To get a better idea about tokens, let's see how BASIC program lines are stored by the MSX system.

```
10 REM test program
```

is stored as a series of numbers in the RAM of the computer. These numbers are;

```
20, 192, 10, 0, 143, 32, 116 . . .
```

The 20 and 192 form a pointer for the beginning of the next line of the BASIC program. This pointer is stored as a two byte integer, with the low byte stored first. Thus in this particular example, the next program line is stored at address 20+(192*256), or address 49172. At this address we will find another two byte pointer which points to the beginning of the next line. The 10 and the 0 are the line number, again stored low byte first.

Finally, we get to the text of the BASIC line. Note how the REM statement is represented by the token 143 and the text of the remark. Not all the ASCII codes of the remark are shown as they are not terribly important at this time. It is in this form that the program is saved if we use the command CSAVE. The command

```
CSAVE "test"
```

will save the program that is currently residing in the computer as a tokenized file called "test". One obvious reason for saving programs in this fashion is that they take less time to transfer to tape — only one byte is transferred for the statement PRINT instead of the 5 bytes that would have to be transferred if the program were being saved as an ASCII file. The baud rate used for the CSAVE command is either the default rate, 1200 baud, or the current rate selected by the last SCREEN command. However, should you wish to specify a baud rate, then an extra parameter is added to the CSAVE command. Thus

```
CSAVE "test", 1
```

will save the program at 1200 baud and the command

```
CSAVE "test", 2
```

will save the program at 2400 baud. The baud rate set in a CSAVE command is only applicable to that command and has no effect on the baud rate of subsequent data transfer operations. Note that after you press the RETURN key after typing a CSAVE command, data immediately begins to be transmitted to the tape recorder. It is thus necessary to press the RECORD and PLAY keys on your tape recorder before pressing RETURN. The first piece of data to be written to the tape is a single tone called the LEADER. By "listening" to this, the computer can determine the baud rate at which the data was recorded when we come to reload the program. Next comes the HEADER, which contains information about the following file, and then finally the data representing the program is written to the tape.

After pressing PLAY and RECORD on the tape recorder, the tape motor will only start going if the remote plug on the tape lead supplied with your computer is not plugged in. If it is, then the motor will not start going until the RETURN key is pressed after the CSAVE command is entered. After the program has been saved to tape, the cursor will return to the screen and the tape motor, if controlled by the remote facility, will be turned off.

Now let's move on to saving files of ASCII characters to represent BASIC programs. As each character is saved to tape, programs saved as ASCII files take longer to save and hence take up more space on tape. However, saving programs as ASCII files does have its advantages. Programs saved in this way can be used as data for other programs to work on. We shall look at data files in more detail shortly. But the main advantage in saving a program as an ASCII file is that a program so saved can be added on to a program that is already in memory.

This process, known as MERGING the two programs, is quite useful. It enables us to store commonly used sections of programs on tape and then incorporate them into other programs with the minimum of effort. To save a program in ASCII format in MSX BASIC, we use the SAVE command.

```
SAVE "test"
```

will save the program in ASCII format to tape in a file named "test". The command

```
SAVE "CAS:test"
```

will also save the program to cassette in ASCII format. However, the CAS: part of the filename is called a DEVICE DESCRIPTOR and it enables programs to be transferred to other devices in ASCII format. For example, the command

```
SAVE "CRT:test"
```

will perform an operation that is similar to the LIST operation; the program will be displayed on the TV screen. If we have a device descriptor "LPT:" in the file name then the file will be listed on a printer, if connected, just as if the LLIST command had been issued. The baud rates used in these data transfers is that baud rate that was set by the last SCREEN command or 1200 baud. There is no way of selecting a baud rate with the SAVE command as there is with the CSAVE command.

The file name used can be a string variable; thus the commands

```
A$="CRT:test"
SAVE A$
```

will list the currently held program to the screen. If we were to repeat the above operation with A$ equal to "CAS:test" then the program would be written to tape as an ASCII file called "test". The file name used in CSAVE commands can also be a string variable; the commands

```
A$="test"
CSAVE A$
```

will save the program to tape in tokenised form with the file name "test".

There is one final point to note about the SAVE command. When we write the file, the last ASCII code that is sent to the tape is a character 26. This is sent so that the computer will recognise the end of the ASCII file when it encounters it on reloading. For this reason the CHR$ 26 is called an End Of File marker.

Loading Programs

To load a program, we use one of two different commands, depending upon the method used to save the program. If the program was saved with the CSAVE command then we reload it using the CLOAD command. The command

```
CLOAD
```

by itself will load the next tokenised BASIC program that is found on the tape. However, we may have saved several programs on the same tape and so we usually use CLOAD with a file name, as below.

```
CLOAD "test"
```

will load the program "test" when it finds it on the tape. "Test" should, of course, be a tokenised file saved by the use of CSAVE. Files encountered while the computer is searching for "test" will be ignored. In both cases, as soon as the correct file is found the message

```
Found: test
```

is printed to the screen. Loading will now proceed. As soon as loading starts, the program that was previously in the computer is lost. As soon as the program has been loaded, the computer returns to direct mode. For this reason, CLOAD is not much use as a statement in a program, as on completion program execution would be halted. CSAVE can be used in a program line, and after the save has been completed execution of the program continues.

If the program has been saved in ASCII format with the save command, then we load it back in using the LOAD command. This is the equivalent of the CLOAD command, but works on files that have been saved as ASCII files. Thus

```
LOAD "CAS:test"
```

will search for an ASCII format file called "test". Should you simply want the next ASCII format file encountered to be loaded, then the command used is

```
LOAD "CAS:"
```

The statement

```
LOAD "CAS:test", R
```

will load the ASCII file called "test" and execute a RUN command, thus starting the program running. The "R" is the only parameter that you can pass to a LOAD command.

Verify

It would be rather nice to think of our computers as infallible in all things; unfortunately, this is not so. The act of saving a program to tape is fraught with problems, as the tape recorder could need cleaning, the tape recorder can be noisy, or running at the incorrect speed, or any one of a dozen problems might beset the act of saving. Before we NEW the machine, therefore it would be useful to know if the computer has recorded a faithful copy of the program onto the tape which will be loadable on a future occasion. The command that is used to do this is called CLOAD?, and it operates by comparing the bytes read off the tape with the bytes in the computer memory. This process is called VERIFICATION; indeed, on some computers the CLOAD? command is called VERIFY. The syntax of the command is;

```
CLOAD? "test"
```

This command will look at the file "test" on tape and compare it with the program currently in memory.

Merge

The final command that we use to manipulate tape files representing programs is called MERGE. As you progress with programming, you will soon gather together a collection of subroutines that are useful in most programs. It is therefore useful to have these saved on tape in such a way that they can be incorporate into any program. We do this by saving the subroutines to tape using the SAVE command, and when we wish to incorporated them into a program that is already partially written, we use the MERGE command. The command

```
MERGE "CAS:test"
```

will search the tape until a file called "test" is encountered. This file should be a program or program section saved in ASCII format. Once the file has been read in, the program lines that were represented in the file will have been added on to the program. Any lines that were in the program resident in the computer at the time of merging, that have line numbers identical to lines in the file, will be replaced by the lines read in from the file. Thus, take care over the line numbers in ASCII files! Any program sections that are to be saved in this way should be renumbered to give them fairly high line numbers. This will reduce the chances of a conflict of line numbers. As soon as the merge operation is completed, the computer returns to Direct Mode and awaits more commands.

If the file name is left off, as in

```
MERGE "CAS:"
```

then the next file that is encountered will be merged into the program in memory. Remember that for a file to be available for merging, it must have been saved with the SAVE command.

As well as storing programs on tape, the MSX BASIC allows us to store variables and blocks of bytes. We will now study each of these in turn, starting off with the storing of variables on tape.

Data Files

As we have already seen, MSX BASIC provides us with a wide variety of data structures in which we can store numbers or strings. When we turn off the power, however, or execute a CLEAR or RUN instruction, the values of the variables are lost. But by using the tape system of the MSX micros to write this data to tape we can have a permanent record of the variables that were in use at a particular point in the program. The values of the variables are written to tape, along with a file name, and can be reloaded at any time. A file that holds numeric or string variables in this way is called a data file. Let us look at a specific application in which being able to save the data used in the program is essential. Imagine we are writing a program that stores names, addresses and telephone numbers. We could have 3 string arrays, name$(50), add$(50) and num$(50), which hold the name, address and telephone number for a particular person respectively.

Each name and its corresponding address and phone number are collectively known as a RECORD. The start of such a file when written to tape could look like Figure 4.2.

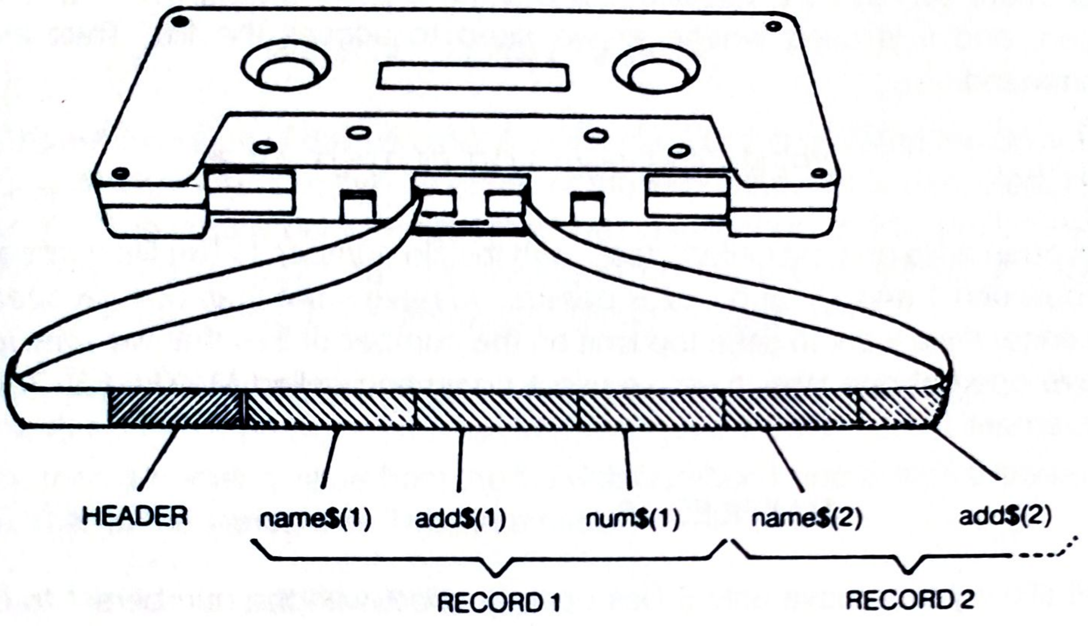


FIGURE 4.2 FORM OF TAPE FILES (SEQUENTIAL FILES)

So, how can we get data held in a variable onto the tape? Let's examine the BASIC statements that we can use to write data to tape.

Opening A File

Before we can write data to tape we must OPEN a file. This operation instructs the computer to prepare an area of memory to act as a BUFFER to which data is written before being sent to tape. A FILE CONTROL BLOCK is also created. This is an area of memory that the computer uses to ensure that the file operation goes ahead as smoothly as possible. Data in the buffer is only written to the tape when the buffer becomes full, or when we signal to the computer that we no longer require the file. It is also necessary to open a file before we can read data from it.

The statement that we use to open a file is of the form

OPEN "CAS:**filename**" FOR **operation** AS **#number**

The file name is the name under which the file will be saved to tape. Note how we again have the chance to use device descriptors to specify where the data is written to. Once we have opened a file, we can either write data to it or read data from the file. Thus the operation in the above statement can be INPUT or OUTPUT. If we open a file for output, then data is transferred from the computer to the tape. If a file is opened for input then data is transferred from tape to the computer. The number in the above statement serves as a specific identifier for the file during the time it is open, and it is used whenever we need to access the file. Thus the command

```
OPEN "CAS:test" FOR OUTPUT AS #1
```

will open a file on tape called "test" with the file number 1. The file number is between 1 and 15, and so it is possible to have more than one file open at once. If we want to set a top limit on the number of files that we want to have open at one time, then we use a command called MAXFILES. The statement

```
MAXFILES=6
```

will allow you to have only 6 files open at once, with the numbers 1 to 6 used as file numbers.

```
MAXFILES=Ø
```

does not allow you to have ANY files open at all. After such a statement has been executed, the only tape operations that are available are the program save and load operations.

In the OPEN statement, "CAS:" can be replaced by either "CRT:" or "LPT:". Again, for "LPT:" to function correctly a printer must be connected to the computer. The statement

```
OPEN "CRT:test" FOR OUTPUT AS #1
```

will cause everything that we write to file number 1 to be written to the screen.

Writing Data to Files

The statements employed here are similar to those we use to print data to the screen. Instead of PRINT, we have PRINT #, and instead of PRINT

USING we have PRINT #, USING. In each case, the "#" is followed by a file number. Thus the statement

```
PRINT #1,35
```

will print the value 35 to the file with number 1. The statement

```
PRINT #1, USING "#.##";A
```

will print the value of the variable A to the file with 2 digits after the decimal point. Remember that before either of these commands is executed, the file to which data is being written must be open for output. In a similar way, files opened for input can be accessed by the use of INPUT #, LINE INPUT # and INPUTS # commands. Thus the statement

```
INPUT #1,A
```

will read a numeric value from the file with number 1 and assign the value read in to the variable A. The statement

```
A$=INPUT$ (3,#1)
```

will cause the execution of the program to wait until 3 characters have been read in from file number 1. For more details, see the notes on INPUT$.

INPUT #filenumber,variable will work in a similar way to the straight forward INPUT statement, but no prompt will be generated. If you are carrying out an input to assign a string variable, then the same considerations apply as in the normal INPUT statement. The LINE INPUT # statement will read in all the characters it encounters in the file until a character 13 is encountered. This final character is then read in and the resultant string is assigned to the variable in the LINE INPUT # statement. Some applications will be given shortly so that you can see how these statements are used.

Closing Files

When we have finished using a file, whether the file was open for input or output, we have to CLOSE it. This is vital for files that are being used for output, as if there is a data item in the buffer, closing the file will ensure that this item is written to the tape. The act of closing a file also releases the RAM that the computer allocated for that file for the buffer and control block for other purposes. Once a file has been closed, you have to re-open it before you can access it again. The statement

```
CLOSE #1
```

will close the file that has number 1 associated with it. If the "#1" is omitted then all files that are currently open are closed.

Examples of the Use of Data Files

In this section, we'll look at how the statements and commands we've met concerning data files can be used in BASIC programs. The first example program, below, is a trivial example but it illustrates the main points.

```
10 REM Writing a file to tape
20 OPEN "CAS:test" FOR OUTPUT AS #1
30 FOR I=1 TO 10
40 PRINT #1,I
50 NEXT I
60 CLOSE #1
```

Line 20 opens the file for output. Line 30 and line 50 make up a FOR-NEXT loop, and the data is written to tape in line 40. Line 60 closes the file to finish the operation. We saw earlier on how we could replace the device descriptor and the file name with a string variable for the save program commands; well, we can do the same here. If we were to insert a line 15, containing A$ = "CAS:test", then we could change line 20 to OPEN A$ FOR OUTPUT AS #1, and this would be perfectly legal.

This is a good place to look at how data is represented in the tape file. If we change the Device Descriptor to "CRT:", we will see the data written to the screen. As you will see, the numbers are written to the screen in the format used by the standard print statement. Thus in the example just seen, the digits are written to tape separated by carriage returns. Strings would also be written to the tape as they would be written to the screen.

The file that we have just written out to tape can now be read back into the computer. The program below demonstrates this in action.

```
10 REM reading a file
20 OPEN "CAS:test" FOR INPUT AS #1
30 FOR I=1 TO 10
40 INPUT #1,J
50 PRINT J
60 NEXT I
70 CLOSE #1
```

As soon as the file has been located on the tape, the FOR-NEXT loop will read in 10 numbers from the file using the INPUT #1, J statement at line 40. The value thus read in is then printed to the screen. An interesting feature of the way in which MSX BASIC reads data from tape files is that line 40 can be rewritten as

```
40 INPUT #1,J$
```

or

```
40 LINE INPUT #1,J$
```

If this is done, and line 50 altered to read PRINT J$, then it will be noticed that J$ contains a string representation of the number that the computer has just read from tape. There is a simple explanation for this. The data items in the file appear just as they would if you were typing in a response to a normal input statement. With numeric values, therefore, the first character that the INPUT statement encounters in a file that is not a space, carriage return or line feed, is treated as the start of a number. Then, each subsequent character is treated as a part of the number until a carriage return, line feed, comma or space is encountered. In this case, with the computer expecting a string input, the first character that is not a space, carriage return or line feed, is treated as the start of the string. If this first character read is a '"', then the string item read in will consist of all characters up to the next '"'. If the first character that is read in by the string input statement is not a '"', then the string item read in will consist of all characters up to the next carriage return, line feed, comma or after 255 characters have been read in from the file. If we wish to read in items from the data file that are strings containing commas, then we use the LINE INPUT # statement. With this statement, the string read in consists of all the characters read in up to and including the next carriage return and line feed. This command is thus of great use where data has been written to tape containing commas, and can also be used to read in to a string variable a line from a BASIC program that has been saved as an ASCII file.

If we construct a file full of string variables, using the program listed below, then we can see what happens when we attempt to read a string item from a file using an input statement that is expecting a numeric value.

```
10 OPEN "CAS:test" FOR OUTPUT AS #1
20 FOR I=1 TO 10
30 PRINT #1, "Hello"
40 NEXT I
50 CLOSE #1
```

The file of strings produced by the program can be read back into the machine using INPUT #1, A$, for example.

```
10 OPEN "CAS:test" FOR INPUT AS #1
20 FOR I=1 TO 10
30 INPUT #1, A$
40 PRINT A$
50 NEXT I
60 CLOSE #1
```

However, replacing line 30 with INPUT #1,A and replacing line 40 with PRINT A will give totally different results. No error is generated, but the value 0 is assigned to the variable. Even if the string is something like "1.234", the numeric value will still be 0.

It is thus quite plain that data must be read from the file in the order in which it was written to the file. Data files can, of course, contain a mixture of both numeric and string data types, as the program below demonstrates.

```
10 INPUT"How many strings"; N
20 OPEN "CAS:string" FOR OUTPUT AS #1
30 PRINT #1,N
40 FOR I=1 TO N
50 INPUT A$
60 PRINT #1,A$
70 NEXT I
80 CLOSE #1
```

Line 10 asks for the number of strings that we wish to write to the tape. The file is then opened in line 20 and the first thing that we write to it is the number of strings that follows. A FOR-NEXT loop then accepts an input from the keyboard and then prints the string typed in to the tape file. Finally, the file is closed. Thus we have a file containing both numeric and string data. The file that is produced by this routine can be read by the program below.

```
10 OPEN "CAS:string" FOR INPUT AS #1
20 INPUT #1,N
30 FOR I=1 TO N
40 INPUT #1,B$
50 PRINT B$
60 NEXT I
70 CLOSE #1
```

Note how we read in the string into B$; the variable into which a value is read from tape does not have to be the same variable name that was written to tape when the file was created.

Use of Variables in Open Statements

The filename and device descriptor can be replaced by a string variable if desired. We have seen how to do this earlier in the chapter. The file number can also be replaced by a numeric variable. The statement

```
OPEN "CAS:test" FOR OUTPUT AS A
```

is legal provided that the current value of A is between 1 and the current value of MAXFILES. When a file has been opened in this way, the INPUT # or PRINT # statements can also contain a file number that is a variable. Here, the variable name that was used in the OPEN statement for the file concerned simply replaces the digit. Thus

```
PRINT #A, "Hello"
```

will print the string "Hello" to the file. If we used variable in this way, however, we must be careful that the value of the variable being used as a file number does not alter while the file is in use!

As a final example of the use of data files on tape, I list below a subroutine that would write the data from our imaginary name, address and telephone number to tape.

```
1000 REM Save file subroutine
1010 OPEN "CAS:data" FOR OUTPUT AS #1
1020 FOR I=1 TO 50
1030 PRINT #1,name$(I)
1040 PRINT #1,add$(I)
1050 PRINT #1,num$(I)
1060 NEXT I
1070 CLOSE #1
1080 RETURN
```

Here we simply use a FOR-NEXT loop to write the elements of the arrays out to tape. When we read the arrays back in, it is necessary that the arrays into which the data is being read in have been dimensioned so as to accommodate all the data that was in the file. In the above example, if we read in the data into arrays that had only been dimensioned to 20 elements, as soon as we tried to read in element 21 from the tape we would get a 'Subscript out of Range' error.

You now have enough information to enable you to write simple file handling programs. One thing that you will have noticed is that you cannot read, say, record 21 from a file without first having read record 20, and the other records prior to that. Files written in this way are called sequential files, as the records are accessed in a particular sequence.

Saving Bytes

Once you start to write machine code programs you will soon want to save the machine code you have written. Everything that we have written to tape files so far has been structured in some way — the BASIC programs we saved and the data files that we have constructed were all written to tape without us needing to know exactly where in the computer's memory the data that was saved was stored. When we come to save machine code, or any other blocks of bytes to tape, however, we need to know 2 things about the bytes that we want to save.

i The position in memory of the first byte that is to be saved — the START ADDRESS.
ii The position in memory of the last byte that is to be saved — the END ADDRESS.

The commands that we use to write bytes to tape and to read them back from tape are BSAVE and BLOAD, respectively. Think of the commands as Byte SAVE and Byte LOAD. The BSAVE command has either 2 or 3 parameters. The essential parameters are the start and end addresses of the code to be written to tape. The third parameter is the EXECUTION ADDRESS, and this is only specified if the file represents a machine code program. The execution address is the address from which the program is run, and provides information for the computer when the file is reloaded. The command

```
BSAVE "test",200,499
```

saves 299 bytes of memory to tape, starting at address 200 and ending at address 499. It is possible to have a device descriptor with the filename, such as "CAS:test", but at the time of writing the only device supported was the cassette recorder. The file name, as we might now begin to expect, can be replaced by a string variable. The start and end addresses can also be replaced by numeric variables, as the command below demonstrates.

```
start=200
ed=499
BSAVE "test",start,ed
```

will save the bytes between start and ed. We couldn't use 'end' as a variable name here because it is a BASIC statement. If it becomes necessary to specify an execution address then it is the third parameter of the command, as can be seen below.

```
BSAVE "game",49800,50000,49820
```

This command will save the bytes between 49800 and 50000, with an execution address of 49820. If the execution address is not specified, then it is assumed to be the start address.

To reload the bytes saved with BSAVE, the BLOAD command is used. In its simplest form, BLOAD is used in the manner below.

```
BLOAD, "CAS:"
```

When used with just the device descriptor in this way, the command will load the next suitable file encountered on the tape. The command

```
BLOAD "test"
```

loads in the file "test" from the tape. In both these examples, the file is loaded to the address from which it was originally saved. If the file name is followed by the letter 'R', as below,

```
BLOAD "game",R
```

then the file is loaded into the computer and is then run, execution starting at the execution address specified when the file was saved. There is one more parameter that we can use with the BLOAD command. This is called 'OFFSET' and enables us to load the file in to your computer at a different address to that from which it was saved.

```
BLOAD "game",R,200
```

will load the file to (start+200). Any execution address that was specified when the file was written will also have 200 added to it to take the new load address into account.

That just about sums up the tape handling facilitities in MSX BASIC.

Before we leave this chapter, a few words on a couple of points that are useful on certain occasions. The first is concerned with the function EOF(n), where n is a file number. This function returns the value -1 if the End of File n has been reached, and returns a Ø if the end has not been reached. If you design your file handling routines so that you are always aware of how many more data items there are in the file to be read in, then you will probably never use this function. If you try to INPUT a data item from a file that doesn't contain any more data, then an error is generated. Checking the file with the EOF(n) function before you actually perform an input will prevent this happening. This function is particularly useful if the file being read in is off an unknown length.

The final points concern errors that are sometimes generated during tape operations.

Tape System Errors

Bad File Name Error Number 56

This error is caused by an incorrect filename being used with a LOAD, SAVE, etc. statement.

Input Past End Error Number 55

Caused by attempting to read data from a file that is either empty of all data or a file that has already had all its data read in.

File Already Open Error Number 54

An OPEN statement has been issued for a file that is already open.

File Not Open Error Number 59

A PRINT #, INPUT # statement has been issued for a file that is not open. Files must be open before these commands are used.

Direct Statement in File Error Number 57

This error occasionally occurs if a direct command is encountered during the LOADing of an ASCII format file. The load is terminated immediately.

Bad File Number Error Number 52

This occurs when a statement or command accesses a file with a file number that is outside the range set by MAXFILES, or an attempt is made to access a file that is not open. It is also generated if you try to execute an INPUT from a file that has been opened for output or vice versa. It is thus a general 'catch all' error message.

Device I/O Error Error Number 19

This error occurs when something rather drastic happens; typical examples are the user typing CTRL-STOP during a tape operation, a poorly recorded piece of data on the tape, electrical noise reaching the tape input or someone pulling the earphone lead on the tape recorder during a loading operation. This can happen to the printer if it is connected, or to the screen if something goes wrong with the computer's innards!

5 The ON Commands

While writing this book, I was often bothered by various little problems that needed my attention. Whenever one of these happenings occurred, I left the typewriter, having first made notes from which I could pick up where I left off on my return, dealt with the problem, and then returned to carry on with the book as if nothing had happened. This is not a dissertation on the rigours of the writer's life, but is an example of an INTERRUPT . . . In the computing world, an Interrupt is a signal that is applied to the Central Processing Unit of the computer to inform the CPU that a device somewhere in the computer needs immediate attention. The CPU finishes what it is doing, stores away the present contents of its internal registers, and then goes away and performs the required job. This is called SERVICING the interrupt, and once this job is completed the CPU resumes its earlier task.

The example of an interrupt that you will probably be most familiar with on the MSX computers is the action of the STOP key. Whatever is happening, pressing this key, especially in conjunction with the CTRL key, will cause the program execution to halt. Other interrupts on the MSX machines read the keyboard and help keep the TV or monitor display going. The fundamental thing about interrupts is that no matter what the computer is doing, the occurrence of the interrupt event causes it to go away and do something else, then return and carry on as if nothing had happened.

This is very similar to the behaviour of the MSX computers when we use BASIC programs that utilise the ON commands. Certain events, such as an error, the depression of a function key or the collision of two sprites, will cause the control of such a program to be passed from the current line being executed to a special routine, which 'services' the BASIC 'interrupt' that the event causes. These ON commands are incredibly powerful; that is why they occupy a chapter on their own. Let's start to examine some of these commands, beginning with the ON ERROR command, a command which gives us the ability to make our programs much more easy for other people, and ourselves, to use.

ON ERROR command

We are all aware of the normal behaviour of the computer when an error occurs; program execution stops, an error message, or REPORT is printed to the screen, and the system variables ERR and ERL contain details about what the error was and where in the program the error occurred. We have already seen in Chapter 2 how errors all have numeric codes associated with them; we will now go on to look at how we can make use of these codes to cause the computer to resume execution of the program with the minimum fuss and bother when an error occurs. This is very important if someone who is not an MSX expert is using your program. The sudden 'beep' and the message "Input past end" will reduce many people into a state of mind that is convinced that computers are a menace! The ON ERROR command causes control to be passed to a certain line number after an error has occurred. The full syntax is shown below.

```
ON ERROR GOTO n
```

where n is an existing line number. Thus, the command

```
ON ERROR GOTO 3000
```

at the start of a BASIC program will cause control of the program to be passed to line 3000 in the event of an error happening. The program statements at line 3000 and on subsequent lines that deal with errors are called the ERROR TRAP. When we use the ON ERROR command, we are said to be enabling error trapping.

Let's see an example of error trapping at work. Type in the command NEW, and then enter the below program. We'll look at screen modes in greater detail in the chapter on the VDP.

```
  10 SCREEN 1
  20 ON ERROR GOTO 3000
  30 FOR I=1 TO 30
  40 PRINT I
  50 NEXT I
  60 GOTO 30
3000 SCREEN 0: PRINT "Ooops, Error number  "
     ;ERR;"  has occurred at line  ";ERL
3010 END
```

Before we see the program operating as an error trap, run it and make sure that it works. If the message that is in the PRINT statement in line 3000 is printed to the display, then you've made a programming error! Assuming that all is in order, the program should begin to print out the numbers using the FOR . . . NEXT loop. Type CTRL-STOP to halt the program. This 'error' is not trapped by the ON ERROR command; it is trapped by a command that we shall meet later in this chapter. However, while the machine is in Direct Mode, type in some rubbish; type in anything that would normally generate a syntax error, or any other error. You will, with any luck, enter the error trap from Direct Mode by this method. The "Ooops . . ." message will be printed, together with the Error number and the number 65535 instead of an error line. The number printed depends upon the error that you caused. You can see from the code that the error number is passed to the error trap routine in the system variable ERR and that the error line is passed over in the variable ERL. To see the error trap operate from within the program, replace line 40 with the code

```
LPRINT I
```

I am assuming here, by the way, that you have no printer connected. Run the program. As we might expect, the machine cannot execute the program due to the printer not being connected. If we now press CTRL-STOP, the error trap is entered with ERR = 19, due to the I/O Error being generated. It is also possible to enter the error trap by use of the ERROR command, from either direct mode or from within a program.

What has this simple routine demonstrated? Well, it has shown that the system variables can be treated just like normal BASIC variables; they can be printed as part of a PRINT statement expression list, and they can also be used in IF . . . THEN . . . ELSE statements, as we shall soon see. The various ways in which control of the program can be passed to an error trap have also been seen; the generation of an error in Direct Mode, the generation of an error in the running program or the use of the ERROR command. However, this simple demonstration doesn't help us regain control of the program. For that, we need to look at another command, called RESUME.

Replace the END at line 3010 with RESUME and repeat the above experiments. You will see some rather interesting, but not terribly useful effects. Whenever an error trap routine finds a RESUME statement, the computer passes control back to the statement that caused the error in the first place. Thus, when the error trap was entered due to a syntax error, instead of the machine simply reporting the fact and stopping, it tried to interpret the offending statement again. This will eventually lead the computer around in circles, and the only way out of the loop will be to press CTRL-STOP. A similar situation will occur with other errors. Thus the RESUME command by itself is not very useful. However, it has other, more useful forms, which we shall now see.

```
3010 RESUME NEXT
```

is the next command to try at this line. Now, after executing the routines in the error trap, the RESUME NEXT command passes control of the program to the next executable statement after the error statement. Thus syntax errors will not cause the loop condition to arise.

The final form of the RESUME command is the most useful. We usually require error handling routines to either restart the program or rerun a small part of the program. For example, if an I/O Device error is generated, we would probably want to re-execute the part of the code that caused the error, after having first given the nature of the error to the user of the program. For example, the user might be asked to check the tape recorder or printer before trying the routine again. The command that enables us to return to a certain line number after an error trap has been executed is

```
RESUME n
```

where n is an existing line number in the program. Look at the example below, which is from a program that uses tape files. Two main errors were expected; either an I/O Device error or a bad file name error. These have the error numbers 19 and 56 respectively.

```
3000 REM Error trap routine
3010 IF ERR=19 THEN RESUME 10
3020 IF ERR=56 THEN RESUME 20
3030 ON ERROR GOTO 0
```

Line 10 of the program called a menu, so that the user could retry that option that had caused the I/O error. Line 20 prompted the user for a file name; thus on the occurrence of the bad file name error, the user was prompted again. The ON ERROR GOTO 0 at line 3030 is so that any programming errors would not cause a jump back into the program but would allow a report of their nature and whereabouts to be made. Thus in this case, any errors generated except numbers 19 and 56 will cause normal error handling to occur. Once the error trap has been entered, then the ON ERROR trapping is turned off until a RESUME instruction has been executed. Once this occurs, then the ON ERROR trap is reactivated.

RESUME must always be followed by a line number in the last method of use that we discussed. It cannot be followed by a variable or expression that evaluates to a legal line number but must be a constant. Error trap routines can include calls to subroutines; errors generated in these routines will give their usual error message, as the error trapping will be deactivated until the RESUME is encountered.

Error

Error traps will work with the BASIC ERROR n command, thus enabling the user to set up his or her own errors that can be trapped like any other BASIC error. Again, ERL and ERR will carry appropriate values into the error trap.

Multiple ON ERRORS

A BASIC program may have more than one ON ERROR statement in it. Each ON ERROR statement takes over from the last one that was executed, and so errors can be directed to different error trap routines according to where the error occurred in the program. For example;

```
10 ON ERROR GOTO 2000
20 PRINT "Hello"
30 hdfgf: REM deliberate error
40 ON ERROR GOTO 3000
```

```
  50 dfhsdg: REM deliberate error
  60 PRINT "Hello"
  70 END
2000 REM first error trap
2010 PRINT "2000"
2020 RESUME NEXT
3000 REM second error trap
3010 PRINT "3000"
3020 RESUME NEXT
```

The first deliberate error in the program will be passed to the error trap at line 2000. The ON ERROR GOTO at line 40, however, redirects the subsequent error to line 3000.

POSSIBLE PROBLEMS

As with subroutines, it is vitally important that the program does not accidentally run into an error trap without entering it in the correct manner. GOSUB is the appropriate method of entry to a subroutine and the only method by which a program should be able to enter the error trap is through an error being generated. If the trap is entered without the error being generated, then the execution of the RESUME instruction in any of its forms will cause an error message to be generated. Thus it should be impossible for a program to accidentally run into an error trap routine. They should be separated from the rest of the program by END or STOP instructions.

One final command that I will mention here is the LIST. command, a special case of LIST. It has already been stated that we can't follow the list command with a variable for a line number, even if the variable is a system variable such as ERL. LIST., when placed in an error trap, will stop the computer by listing the line of the program that caused the error to occur. LIST. can thus be treated as a LIST ERL command. It is thus useful for debugging programs. It can also be used in Direct Mode.

ON KEY command

This command, whose full syntax is

```
ON KEY GOSUB 100,200,300 . . .
```

enables us to define a series of line numbers to which control of the program will be passed in the event of one of the function keys being pressed. Control will be passed in the form of a subroutine call, and so the sequence of lines after the one specified in the ON KEY statement will have to finish with a RETURN. In the above example, control of the running program will be passed to the subroutine at line 200 if key F1 is pressed while the program is running. This EVENT, as it is called, will only be trapped in this way if the function key of interest has been 'turned on' by a KEY (n) ON command, where n is the number of the function key.

Let's look at an example of trapping the function key F1 to jump to a subroutine whenever it is pressed. Don't worry about the SCREEN commands and the writing to the graphics screen yet; we'll look at these two features in Chapter 6.

```
  10 KEY (1) ON
  20 ON KEY GOSUB 1000
  30 SCREEN 1
  40 FOR I=1 TO 10
  50 FOR J=1 TO 100: NEXT
  60 PRINT I
  70 NEXT
  80 GOTO 30
  90 END
1000 SCREEN 3
1010 OPEN "GRP:" FOR OUTPUT AS #1
1020 PRESET (0,20),2
1030 PRINT #1, "That was F1"
1040 CLOSE #1
1050 TIME=0
1060 IF TIME<200 THEN GOTO 1060
1070 RETURN
```

Line 10 activates the key that is to be used when we wish to enter the trap routine, which in this case is F1, and line 20 sets up the line to which control is passed when the key is pressed, in this example line 1000. The key number in line 10 need not be a constant; it could be an expression or variable that returns an appropriate value. Thus putting the lines below in the above program would be perfectly legal.

```
 5 n=1
10 KEY (n) ON
```

The KEY ON command can come before or after the ON KEY GOSUB statement. In the above example, we have only activated key 1 to jump to a subroutine. Suppose that we wanted to use key F5 to cause a jump but no others; in this sort of situation a comma is placed on the ON KEY

GOSUB statement for each key that is inactive. Thus, to make key 5 operate instead of key 1 in the last example program, replace lines 10 and 20 with the two lines below.

```
10 KEY (5) ON
20 ON KEY GOSUB ...., 1000
```

In a similar fashion, the lines

```
10 KEY (6) ON
20 ON KEY GOSUB ,,,,, 1000
```

will cause key 6 to cause the jump to the subroutine at line 1000. To set more than one key up to do a jump to a subroutine, it is necessary to execute a KEY ON command for each key that is of interest, and also to define a line number for that key on the ON KEY GOSUB statement. Thus the lines

```
10 ON KEY GOSUB 2000,,3000,4000
20 KEY (1) ON
30 KEY (3) ON
40 KEY (4) ON
```

will activate keys 1, 3 and 4, so that when the program is running pressing key 1 will cause control to pass to line 2000 of the program, key 3 will pass control to line 3000 and key 4 will pass control to line 4000. If a key is activated by a KEY ON statement, but is not given a line number in the ON KEY GOSUB statement, then any presses of the key are simply ignored.

When we have several keys to turn on with the KEY ON statement, it is quite clear that using several KEY On statements is probably NOT the most efficient way of activating the keys. My favourite method where I need to turn on several keys at once is to use FOR . . . NEXT loops or DATA statements. As an example, look at the activation of keys 1 to 6.

```
10 FOR I=1 TO 6
20 KEY (I) ON
30 NEXT
```

If there are 6 keys to be activated, but they are not in the sequence that these are in, then we use a DATA statement and a FOR . . . NEXT loop to read from the DATA statement, as shown overleaf.

```
  10 FOR I=1 TO 6
  20 READ n
  30 KEY (n) ON
  40 NEXT
1000 DATA 1,2,4,6,7,8
```

This latter program turns on keys 1,2,4,6,7 and 8, something that would be a little difficult to do efficiently by just using FOR . . . NEXT loops.

If we wish to disable a function key that has been active in a program then we use the KEY (n) OFF statement. This is useful when we want the function key to be active for part of the program but inactive in other parts of the program. Thus the command

```
400 KEY (2) OFF
```

will ensure that pressing F2 after this command has been executed has no effect. Of course, if we then want to turn the function key back on we can use a KEY ON command to do so.

KEY (n) STOP has similiar effects to the KEY OFF command; however, while all key presses of the function key are ignored after a KEY OFF command, they are stored up during the time in which the KEY STOP is active. As soon as a KEY ON command is issued for that particular key, the computer will remember if the relevant function key has been pressed during the KEY STOP period, and, if there has been a key press, will pass control to the appropriate line in the program. If the function key concerned was not pressed during the STOP period, then nothing more happens.

When a key trap routine is entered, a KEY STOP command is issued by the computer for that particular function key. A KEY ON instruction is issued when the RETURN is executed. Thus pressing the function key during the execution of its subroutine will cause the routine to be executed a second time as soon as the RETURN is executed.

While using the ON KEY GOSUB command, if you should need reminding about what each key does, don't forget that you can program a string into the function keys. This will be displayed in the normal fashion on line 24 of the display. The nature of the text held in a function key makes no difference to the way in which the computer responds to the key being pressed in an ON KEY GOSUB operation. Remember that if this line 24 display irritates you, the command

```
KEY OFF
```

with no key number involved will remove this display. KEY ON will restore the line 24 display.

The next ON command that we will look at also involves the trapping of an event that is caused by a key press; in this case the pressing of the STOP key.

ON STOP GOSUB n

The main method of stopping a program on the MSX computers is to hold down the CTRL key and the STOP key simultaneously, thus issuing what is known as a Control-Stop command, usually abbreviated to CTRL-STOP. The program will immediately stop executing. This, however, is not always a desirable proposition, especially if the program is being used by someone other than yourself whom you don't wish to be able to see the program. ON STOP GOSUB n enables you to prevent the CTRL-STOP function from stopping your programs. It does not affect the use of the STOP key on its own, which will normally produce a pause in program execution, and neither does it prevent the use of the STOP command within a BASIC program. The CTRL-STOP trap that is set up using this command is only entered by pressing CTRL-STOP when the BASIC program is running. When the trap is active, pressing these keys will cause the program lines starting at the line specified in the ON STOP GOSUB instruction to be executed. Control is passed back to the program by means of a RETURN statement. The CTRL-STOP is thus another form of subroutine, as are all the routines that we have seen in this chapter with the exception of the ON ERROR trap routines, which ended with a RESUME command instead of a RETURN command. In common with the other traps that we have seen in this chapter, the ON STOP GOSUB command has to be 'turned on' by the use of a

```
STOP ON
```

command. This can be issued before or after the ON STOP GOSUB command. Thus the two lines

```
10 ON STOP GOSUB 5000
20 STOP ON
```

will cause control of the program to be passed to line 5000 whenever the CTRL-STOP event occurs. As an example of what can be done with the ON STOP . . . command, the code below will completely inactivate the

CTRL-STOP operation, and this combination of keys will no longer stop the program when it is running.

```
1 ON STOP GOSUB 20000
2 STOP ON
.......
.......
20000 RETURN
```

If you disable the STOP key in this way in a program, then TAKE CARE! It is possible to get into an infinite loop if the program has bugs in it, and should this occur the only way to get out of the loop may be to reset the machine, with the resultant loss of program.

The STOP OFF command turns off the CTRL-STOP trap. The STOP STOP command turns off the trap, but if CTRL-STOP is pressed the event is remembered and acted upon as soon as a STOP ON command is issued.

There is one occasion in a BASIC program protected in this way when the program can be broken into by use of the CTRL-STOP function. This is when an error trap has been entered; until the RESUME statement, all trapping by the use of ON commands is disabled.

When the computer has executed a CTRL-STOP trap routine, the RETURN statement also executes a STOP ON statement. This is because during the trap handling routine, the computer acts as if a STOP STOP command has been issued. If you put a STOP OFF command within the trap routine, then trapping is NOT activated by the RETURN statement, and any subsequent CTRL-STOP's will have the usual effect.

ON SPRITE GOSUB n

This command can only be properly discussed when we have gained a knowledge of the sprites available in MSX BASIC. This will be fully covered in the next chapter, where we will also discuss this command.

ON INTERVAL = time GOSUB n

This command enables us to program the computer to stop whatever it is doing after a given time interval, execute a subroutine starting at line n, and then return from that subroutine and carry on until the interval has elapsed again, when the process will be repeated. To implement this command, the computer makes use of an interrupt that is generated by the VDP. This tells the CPU to execute some ROM routines that read the keyboard and update the value of the TIME variable. The number of times that this happens per second depends upon the type of MSX system that you've got. If it is a computer that can drive a TV set in the UK, then this interrupt occurs 50 times per second. If the machine is one that was originally intended for the Japanese market, and cannot drive a UK television set, then this interrupt occurs 60 times a second. This interrupt is counted by the computer and made use of when we use the ON INTERVAL command.

The delay, time, is given by the delay required in seconds multiplied by the number of times per second that the interrupt is generated. Thus in the UK, the value of the parameter time is given by

time = required interval * 50

Thus for a delay of 5 seconds between executions of the sub-routine specified in the ON INTERVAL command, the command used would be

```
10 ON INTERVAL=250 GOSUB 1000
```

To activate trapping of the ON INTERVAL command, an INTERVAL ON command has to be issued in a similar way to the KEY (n) ON commands. Again, the sub-routine that is called must end in RETURN.

The program listed below makes use of the ON INTERVAL command to give an 'alarm clock' style program. Once the INTERVAL ON command has been executed the computer enters the subroutine at line 1000 every 10 seconds. The frequency of entry into the subroutine can obviously be varied by altering the time parameter in line 10.

```
   5 SCREEN 0
  10 ON INTERVAL=500 GOSUB 1000
  20 INTERVAL ON
  30 FOR I=1 TO 1000000
  40 PRINT I
  50 NEXT I
  60 END
1000 SCREEN 3
1010 OPEN "GRP:" FOR OUTPUT AS #1
1020 PRESET (0,40)
1030 PRINT #, "Wake up!"
1040 BEEP:BEEP
1050 CLOSE #1
```

```
1060 FOR T=1 TO 2000: NEXT T
1070 SCREEN 0
1080 RETURN
```

As soon as the interval on command is issued in line 20, the routine at line 1000 is entered each time the requested time delay finishes — in this case, for example, the routine will be entered after 10, 20, 30 . . . etc. seconds. This time includes time spent executing the trap routine. Thus, if a trap routine takes 3 seconds to execute, and the time parameter in the ON INTERVAL command is still set at 500, then the trap will be re-entered 7 seconds after it was last exited. If you only wanted the interval to elapse once, giving one entry into the routine and no more, then an INTERVAL OFF instruction in the subroutine will ensure that this happens. On entry to the subroutine, an INTERVAL STOP command is executed. This has similar effects to the STOP STOP command. If another time delay expires while the trap is being executed, then as soon as the RETURN at line 1080 is executed the trap is re-entered. You can see this in action by reducing the time parameter in line 10 of the demonstration program to, say, 10. This is a silly example, but if you have a low value of a time parameter, the execution time of the trap routine has to be similarly short if the program is to do anything other than repeatedly run the trap routine. You have been warned. INTERVAL OFF and INTERVAL STOP instructions can, of course, be used in other parts of the program to prevent the ON INTERVAL calls occurring when they are not needed. Within the time constraints mentioned above, the trap routine can execute any commands you wish. As with all the other ON commands, the program will continue executing as if nothing had happened when the RETURN command is executed.

ON STRIG

This command is used with the joystick; therefore we will deal with it in Chapter 7.

★ ★ ★

That finishes the ON commands. The ability of ON INTERVAL to perform a regular subroutine call will be used in the next chapter when we go on to look at sprites and graphics as we look at the Video Display Processor.

6 The Video Display Processor

The Video Display Processor, or VDP for short, is the device in the computer that gives the MSX microcomputers their excellent graphics abilities. It is one of the chips that is part of the MSX standard, and so in this chapter we shall look at in detail, beginning with some general information.

The device listed in the MSX specification is the TMS 9918 or equivalent. It controls the display that the MSX can provide to either television or monitor, and is in total control of 16384 bytes of Video memory, known as VRAM. 16 colours are available, as we shall later see. The VDP is in communication with the Central Processor Unit via the bus system of the computer, and the following types of information transfer are possible between the CPU and the Video Processor.

- i The CPU sends bytes to the VDP REGISTERS
- ii The CPU reads bytes from the VDP registers
- iii The CPU sends bytes to the Video RAM
- iv The CPU reads bytes from the Video RAM

A couple of definitions at this point; a register is best seen as a byte of RAM within the Video Processor chip, that controls the operation of the VDP. It is not part of the VRAM. Within the VDP there are 8 registers called WRITE ONLY registers; the Z-8Ø CPU can write bytes to these registers but cannot read data from them. There is also one READ ONLY register, called the STATUS REGISTER. We'll look at these in more detail later in the chapter. The Video RAM holds data pertaining to the sprites, the screen display, the colours in use and various other things. By directly accessing the Video RAM or the VDP registers, we can greatly increase the programming power at our disposal. The second part of the chapter will deal with this method of using the VDP from BASIC. However, in the first part of the chapter we'll look at the BASIC commands that are available to us without having to gain a knowledge of the arrangements of the Video RAM or the Video Processor registers.

Display Modes

A Display Mode is a particular way of arranging the display screen. The MSX computers have 4 modes, all using the Video RAM in different ways and each mode being best suited for a particular application. Display Modes are selected using the SCREEN command.

Mode Ø	Text Mode
Mode 1	Text Mode
Mode 2	High Resolution Graphics mode
Mode 3	Low Resolution Graphics mode

Let's look at these modes in some detail now, and see what we can do with each of them.

Mode Ø

This is a text mode which offers you the characters displayed on the keyboard of the computer — that is, letters, numbers and various non-alphanumeric characters. The mode is specified as offering 24 lines of 4Ø characters per line; however, the MSX microcomputers that will be available in the UK will actually display 24 lines of 37 characters. The actual characters that are displayed in this mode are stored in part of the Video RAM. This means that by directly altering the VRAM we can alter the shape of the letters printed to the screen. We'll look at how we can do this later in the chapter. Two colours out of the 16 available can be used in this mode, one for the letters displayed and one colour for the background.

Mode 1

This gives us 24 lines of 32 characters, but again the UK machines will display 24 lines of 29 characters. In the MSX specification, two colours out of the available 16 are usable, one for the foreground colour — that is, the characters displayed — and one for the background. However, by manipulating the VRAM directly, we can modify this slightly. Again, it is possible to modify the character set in VRAM.

The graphics commands of MSX BASIC, such as PSET, LINE or DRAW will generate errors if you try to use them in a text mode. However, in mode 1 sprites are available, and we shall discuss these later in the chapter. One important thing to note is that a text mode will be returned to whenever you execute an INPUT statement from within a graphics only mode or when the computer has finished execution of a program. As this results in the loss of whatever was on the screen at that point, remember to finish the program in an infinite loop if you wish to examine the results of graphics commands.

Mode 2

Mode 2 gives us access to high resolution graphics capability. We can have 16 colours on the screen at one time, and we can use sprites. The resolution of the screen is 256 by 192 pixels. Treat a pixel as a dot on the screen. However, in each group of 8 pixels in the horizontal direction you can only have two separate colours — one foreground colour and one background colour. Thus it is not possible in the horizontal direction to have a row of 8 pixels all having different colours. In the vertical direction, however, there are no such limitations. For 8 vertical pixels in this mode, 8 different colours can be used if necessary.

Mode 3

This is the low resolution graphics screen, again with 16 colours and sprites. The resolution of the screen in this mode is 64 horizontal pixels by 48 vertical pixels. Any colour can be held by any pixel in any screen position. There are no problems with horizontal colour resolution as there are in Mode 2.

If you have experimented at all with the graphics modes of the MSX computers then you will probably be aware of the fact that the normal print statements do not print text to the graphics screens. There is a way around this problem, and we shall look at this shortly.

Printing Text

We'll firstly look at the commands available to us in the text modes, with the exception of sprite control, which has a separate section in this chapter. To see the characters that are available to you in text mode, enter the program below.

```
10 SCREEN Ø
20 FOR I=32 TO 255
30 PRINT CHR$ (I);
40 NEXT I
50 END
```

Line 1Ø selects screen mode Ø. If we wanted any other mode here, we simply put in the appropriate number instead of Ø. We then simply print out all the characters between ASCII code 32 and 255. Note how as the characters are printed we use the full width of the screen. We can, in fact, vary the number of characters that we write to a line by the use of the WIDTH command.

```
WIDTH 2Ø
```

will set the display line to 2Ø characters. Legal values of the parameter in the WIDTH command are between 1 and 4Ø in screen mode Ø, and between 1 and 32 in screen mode 1. Try including the line below in the above program:

```
15 WIDTH 15
```

Note how the use of this command also affects the function key display on line 24 of the display. The width of the display line set with this command stays fixed at that length, even if you change display mode. The only way to restore it to its original value is to reset it with a WIDTH command.

When you have filled a screen with text, the CLS command will clear the screen to blank. The CLS command also works in the graphics modes.

```
LOCATE X,Y
```

The LOCATE command enables us to position text on the text screen wherever we want it. The full syntax of the command is:

```
LOCATE X,Y,cursor
```

The cursor parameter is optional, and if omitted is assumed to be Ø. The command ensures that the next print statement issued by the system prints the text at location X,Y on the screen, X being the horizontal position from the left of the screen and Y being the vertical position from the top of the screen. In the MSX system, the top left hand character of the text screen is at location 0,0. X increases from left to right and Y increases from screen top to bottom. The cursor parameter determines whether or not the cursor block will be displayed after the next print statement has been executed. Look at the example below:

```
10 LOCATE 10,10,0:PRINT "#"
20 GOTO 20
```

will print the "#" but nothing else. However, if we alter line 10 to

```
10 LOCATE 10,10,1:PRINT "#"
```

will cause the "#" to be printed, followed by the cursor block. With the cursor parameter equal to 1, the cursor is said to be enabled, and is disabled if the parameter is equal to 0.

Colour

So far, we've not strayed away from the white letters on a blue background that the MSX computers all start off with when we turn them on. Changing text colour is easy, if we can remember to use the Japanese spelling of colour! Seriously speaking, though, the COLOR command enables us to change the text colour, the background colour and the border colour of the text display. The full syntax is

```
COLOR foreground, background, border
```

The foreground colour is that colour which the text is printed in, the background is the colour of the blank screen and the border colour is the colour of the edges of the screen which cannot be written to. Thus the command

```
COLOR 15,1,1
```

will give us white lettering in a black background with a black border. The colours available and the numbers used to represent them in COLOR statements are as follows.

0	Transparent	**4**	Dark Blue
1	Black	**5**	Light Blue
2	Medium Green	**6**	Dark Red
3	Light Green	**7**	Cyan

8	Medium Red	**12**	Dark Green
9	Light Red	**13**	Magenta
10	Dark Yellow	**14**	Grey
11	Light Yellow	**15**	White

Just one word of explanation is needed here, and that is about the colour transparent. This simply allows whatever is under it — whatever is the background — to show through. Due to the fact that we are only supposed to have two colours available to us in the text modes, changing the foreground colour will change the colour of text already written to the screen to the new colour.

Text in Graphics Modes

Try the program below:

```
10 SCREEN 2
20 PRINT "Hello"
30 GOTO 20
```

Nothing is printed to the screen. This is because some rather special techniques are employed to write text to the graphics screens. However, once mastered, they give a great degree of control over the positioning of text on the display screen and allow us to mix both text and high resolution and low resolution graphics with text.

We gain access to the graphics screens by the use of an OPEN command and a Device Descriptor. The new descriptor is called GRP:. Just as we were able to write text to the text screens using OPEN"CRT:", we use OPEN"GRP:" to write to the graphics screen. The full syntax of the command needed to prepare the way for writing text to the graphics screen is

```
100 OPEN"GRP:" FOR OUTPUT AS #number
```

or

```
200 OPEN"GRP:" AS #number
```

The command can only be sensibly used from within a program after a graphics mode has been selected. The parameter number is the equivalent of the file number that we used when we were writing text to tape files. In a similar way, we use the PRINT # and PRINT # USING commands to write text to the graphics screen. RESET your machine and try the demonstration program below.

```
10 SCREEN 2
20 OPEN "GRP:" AS #1
30 PRINT #1,"Hello"
40 CLOSE #1
50 GOTO 50: REM prevents return to Direct
   Mode
```

You'll see the word "Hello" printed to the top left-hand corner of the display. If you stop the program and then re-run it, you will find that the printed text has moved down the screen by 1 line. This is due to the computer "remembering" that it printed a carriage return at the end of the word "Hello" on its previous run. Thus the new text is printed accordingly. We are therefore in need of a method of positioning the text we wish to print to the graphics screen. LOCATE will not work, and the WIDTH command has no effect until we return to a text mode, when the width we specified in the WIDTH command that was executed in the graphics mode will come into effect.

The command we use to position the text is called PRESET. The syntax of the command when used for text positioning in graphics modes is

```
PRESET (X,Y)
```

where X is the horizontal position of the character and Y is the vertical position of the character. PRESET works on a 256 by 192 grid in mode 2 and on a 64 by 48 grid in mode 3. The program below shows the PRESET command in use, to position text randomly on the screen.

```
10 SCREEN 2
20 OPEN"GRP:" AS #1
30 PRESET (RND(1)*256,RND(1)*192)
40 PRINT #1,"Hello"
50 GOTO 30
60 CLOSE #1
```

The CLOSE statement is rather superfluous here as it is never executed due to the GOTO at line 50. The coordinates passed over with the PRESET command refer to the top left hand corner of the first character of the string that is to be printed.

The coordinate system used again has position 0,0 in the top left hand corner of the screen. Due to the difference in pixel size, text printed in Screen mode 3 is larger than the text printed in any other mode. It is thus excellent for title pages in programs and other such applications where large text is needed. To see this big text in action, simply change the SCREEN 2 in the above program to SCREEN 3. Changing the colour of text printed to the graphics screen is very simple — we just use the COLOR command. We now, of course, have the 16 colours of the graphics modes available to us. This, and the degree of variation we can have in positioning the text makes the use of text in graphics modes very useful. The program below shows how we can change the text colour with the COLOR command. The colour of the words printed to the screen will be repeatedly changed as the program runs.

```
10 SCREEN 3
20 OPEN"GRP:" FOR OUTPUT AS#1
30 PRESET (0,0)
40 COLOR RND(1)*15
50 PRINT #1,"Hello"
60 GOTO 40
70 CLOSE #1
```

The background colour and border colour parameters can be used in such commands, but while in the graphics mode only the foreground and border parameters are acted upon. The background colour used when printing text to the graphics screen is the same one that was in use in the last text mode used. If you want to change the background colour of the screen using the COLOR command whilst in a graphics mode, then use the command combination below.

```
1010 COLOR foreground, background, border
1020 CLS
```

This will, of course, clear the screen of anything there already.

If, while indulging in all these colour changes, you accidentally come back to the text mode with a completely unreadable colour combination, then a press of F6 will set matters to rights.

When all the text has been written to the graphics screen that you want to write, the CLOSE #1 instruction must be executed. If you've used some other file number than 1 here then simply replace the 1 with the number you used.

Let's now go on to look at the simple graphics commands that are available to us in modes 2 and 3. The simplest thing to do in graphics terms is to make a pixel at a particular screen position a certain colour. This is often called plotting a point. The coordinate system used by the graphics commands again starts at the top left hand corner of the screen. When BASIC encounters coordinates in commands it will allow you to have values that are beyond the edge of the screen, as long as the values in the coordinate pair are within the range -32768 to +32767. The values that would be outside the screen are replaced by the closest value that is just on the screen. Thus, 0 would replace any negative values that are specified as a coordinate, if the coordinate is an absolute coordinate. We say that we are plotting a point in an absolute fashion if the coordinate refers to the position of the pixel to be modified with respect to 0,0. It is possible to specify the position of a pixel with respect to another point apart from 0,0. This is said to be plotting a point relative to some point on the screen..

The commands that we can use to plot a pixel in a given colour are called PSET and PRESET. These commands are virtually identical in action. The syntax for the PSET command is

```
PSET (X,Y), colour
```

X and Y specify the coordinates of the point, and the colour parameter specifies the colour that you want the pixel to be. The colour parameter can be omitted from the PRESET command, which has the same syntax as the above. If the colour is left out then the current background colour is chosen. This was the way we used PRESET to position text on the graphics screen. If we wanted to, we could use PSET to do the same job, using a transparent pixel colour:

```
PSET (X,Y),0
```

The size of the pixel obviously depends upon the graphics mode in use. To see this in action, type in the program below, and run it in both modes 2 and 3.

```
10 SCREEN 2: REM or SCREEN 3
20 PSET (RND(1)*50,RND(1)*50),RND(1)*15
30 GOTO 20
```

Although the size of the area of the screen affected by the program is the same in both cases, the pixels that are plotted in Mode 2 are only a quarter of the size of those plotted in Mode 3. As a further demonstration, the below program draws a graph, showing the Sine and Cosine for various angles in two different colours. The sines and cosines are evaluated before we begin drawing, to speed up the drawing process. The angle in degrees, represented by I%, has to be converted first into radians before the sine or cosine functions can be applied. Thus A is the angle in radians. This program is also a good example of the slow speed of trigonometric function evaluation on MSX computers!

```
 10 SCREEN 0: LOCATE 10,10:PRINT "Please
    Wait"
 20 DIM S(360), C(360)
 30 FOR I%=0 TO 360 STEP 2: A=I%/57.33
 40 S(I%)=SIN(A): C(I%)=COS(A)
 50 NEXT I%
 60 SCREEN 2:REM or SCREEN 3
 70 FOR I%=0 TO 360 STEP 2
 80 PRESET (I%/2,S(I%)*30+50),7
 90 PRESET (I%/2,C(i%)*30+50), 1
100 NEXT I%
110 GOTO 110
```

The graphs can be plotted in both mode 2 and mode 3, simply by changing line 60 accordingly. The next demonstration is what is called in computing circles a "Random Walk". The next pixel to be plotted is specified to a certain degree by random factors. In this particular case, the X and Y coordinates are either incremented or decremented, depending upon the values of 2 random numbers.

```
10 SCREEN 2
20 X%=100:Y%=100
30 PSET (X%,Y%),15
40 N%=RND(1)*2:M%=RND(1)*2
50 IF N%>0 THEN X%=X%+1 ELSE X%=X%-1
60 IF M%>0 THEN Y%=Y%+1 ELSE Y%=Y%-1
70 GOTO 30
```

Try altering the amounts by which X% and Y% are altered in lines 50 and 60.

Now that we can use the PSET and PRESET commands to plot individual pixels, it would be nice to be able to draw lines between points on the screen. The MSX BASIC command that we use to do this is called LINE. The full syntax of the LINE command is

```
LINE (X,Y)-(X1,Y1)
```

X and Y are the coordinates of the position at which line drawing is to start and X1,Y1 is the position at which the line is to be finished. Thus the line drawn by the command

```
LINE (10,10)-(100,100)
```

will start at position 10,10 and finish at position 100,100. We can specify the colour of such a line by either a COLOR command before we draw the line or by a colour parameter at the end of the LINE command, as below.

```
LINE (10,10)-(30,100),1
```

This command would draw a black line between the specified coordinates. The numbers used in the colour parameter are those that we've already seen. There is one final parameter that we can add to a line command, which is quite useful. To draw a rectangle, or 'box', on the screen would normally take 4 LINE statements to draw. By adding the parameter B to the LINE statement. The command

```
LINE (10,10)-(100,100),1,B
```

will draw a black box to the screen with it's top left hand corner at position 10,10 and it's bottom right hand corner at 100,100. We can replace the letter B with BF. This will draw the box and then colour it in in the colour specified in the LINE command. Obviously, the shapes drawn with this extended line command are all rectangles with parallel sides. Should we want to draw irregular shapes or triangles, we will still have to separate LINE commands.

Before leaving the LINE command, let's look at how we can draw lines or plot points relative to the current graphics position, rather than to and from absolute coordinates. The X,Y pair of coordinates in any graphics command so far encountered can be replaced by an expression of the form

```
STEP (X,Y)
```

The STEP signifies that relative addressing of the coordinates is to be used. Thus the pair of commands

```
200 PRESET (100,100)
210 LINE STEP (10,10)-(30,60)
```

will result in the drawing of a line from point 110,110 to point 30,60. The

10,10 in the STEP expression refers to point 10,10 relative to the last graphics point used, which in this case was the point 100,100 specified in the PRESET command.

The final form of the LINE command is to omit the start coordinates. The command is

```
LINE-(X1,Y1)
```

and the line is drawn from the last graphics point that was accessed to the point X1, Y1.

We've seen how we can draw boxes to the screen; what about something a little more difficult, such as a circle? Most home computers require that the user write a subroutine to draw circles; not so the MSX system. The command CIRCLE enables us to draw circles or ellipses in either of the graphics modes. The full syntax of the command is

```
CIRCLE (X,Y), radius, colour, start angle, finish angle,
aspect ratio
```

STEP (X,Y) can be used instead of the (X,Y) expression in the above syntax. Of the parameters, colour, start angle, finish angle and aspect ratio are all optional. Let's begin by drawing a few circles.

```
10 SCREEN 2
20 X=RND(1)*256:Y=RND(1)*192:REM random
   circle position
30 R=RND(1)*30:REM Random circle radius
40 CIRCLE (X,Y),R,RND(1)*15:REM draw circle with
50 REM random colour
60 GOTO 20
```

This program will run in mode 3 as well, but the circles will be drawn with much thicker lines. If you were to use the STEP (X,Y) method of positioning the circle, then the first circle drawn in the above program would be dependent upon the position reached by the last PSET, LINE etc. command. The CIRCLE command can only be used in the graphics modes.

Let's now look at the final three parameters that we can use with the CIRCLE command — the start and finish angles, and the aspect ratio. These are used when drawing sections of circles or ellipses. Let's start by looking at how we can draw sections of circles to the screen.

We do this by specifying the start and finish angles in the CIRCLE command. These parameters are in radians, but we can use the function that we designed a short while ago to convert degrees to radians to solve this problem. I will thus address this problem in degrees, as I believe that more people are familiar with this measure of angle. The angles used in these two parameters are between Ø and 36Ø degrees. The MSX computers draw circles as illustrated in Figure 6.1.

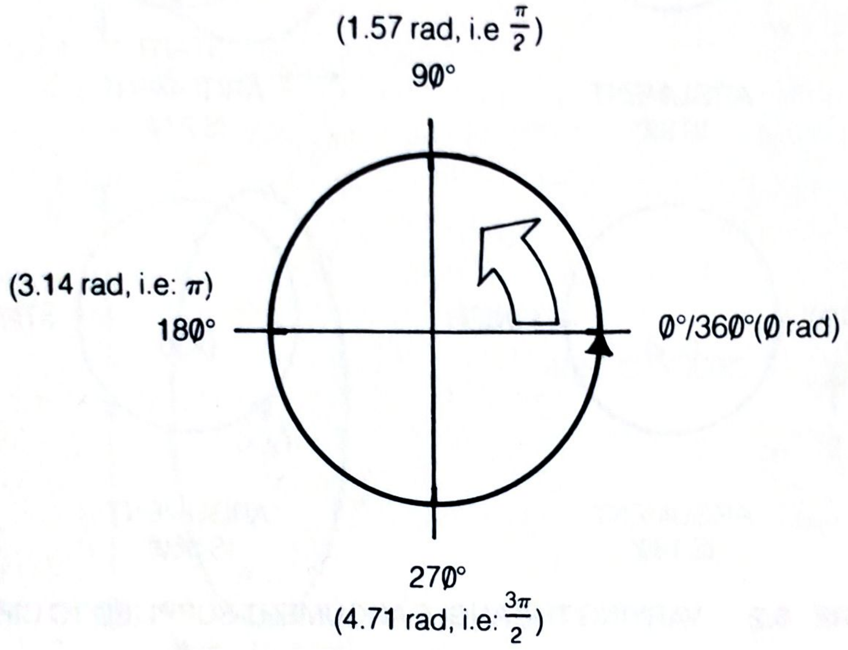

(1.57 rad, i.e $\frac{\pi}{2}$)
9Ø°
(3.14 rad, i.e: π)
18Ø°
Ø°/36Ø°(Ø rad)
27Ø°
(4.71 rad, i.e: $\frac{3\pi}{2}$)

FIGURE 6.1 ANGLE CONVENTION, CIRCLE STATEMENT

The circle is thus drawn in an anti-clockwise direction, starting at the start angle and ending at the finish angle as we might expect. Thus when we draw a circle command, the start angle is taken by the system to be Ø degrees and the finish angle is assumed to be 36Ø degrees. The program below allows you to see the effects of changing the start angle of the circle drawing procedure. Line 2Ø of the program defines the function for our degrees to radians conversion. In line 30 we call this function, putting the required start angle as the argument of the function. You can thus change the start angle by altering the parameter passed to the function.

```
1Ø SCREEN 2
2Ø DEF FNA(angle)=angle/57.33
3Ø st=FNA(18Ø):REM to change start angle, change
4Ø REM this argument
5Ø CIRCLE (1ØØ,1ØØ), 4Ø,15,st
6Ø GOTO 6Ø
```

Figure 6.2 shows the effects of varying the argument.

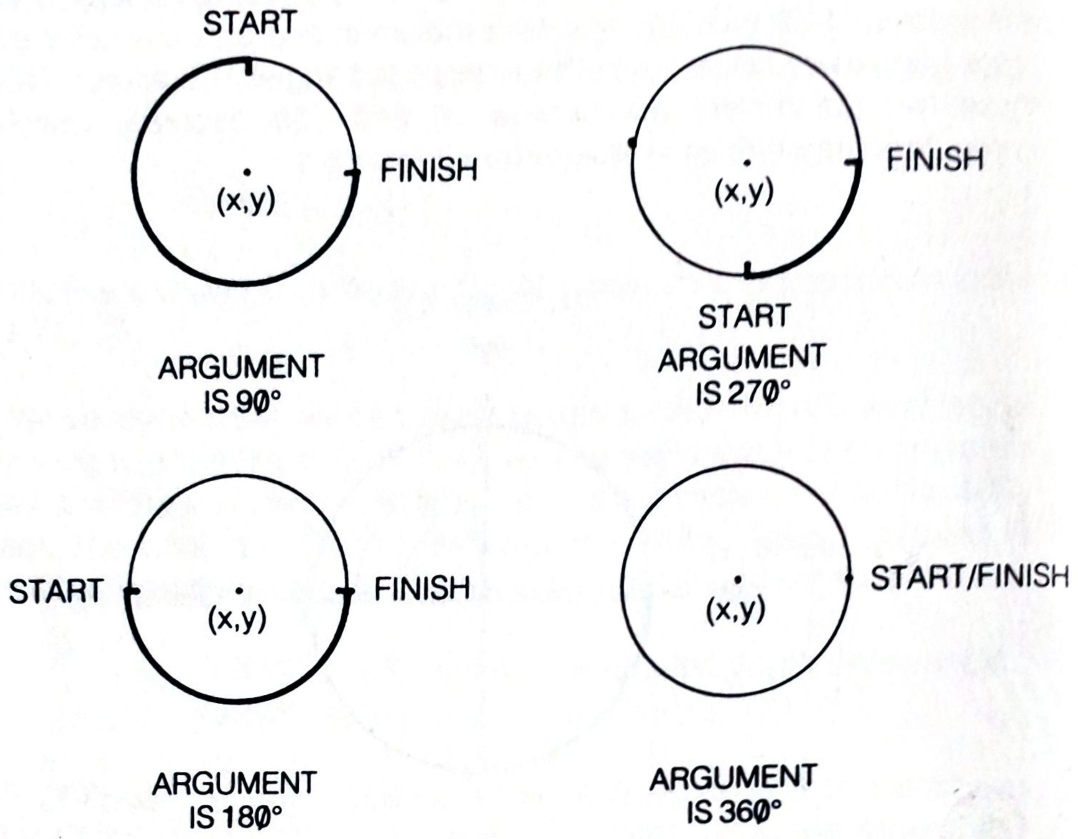
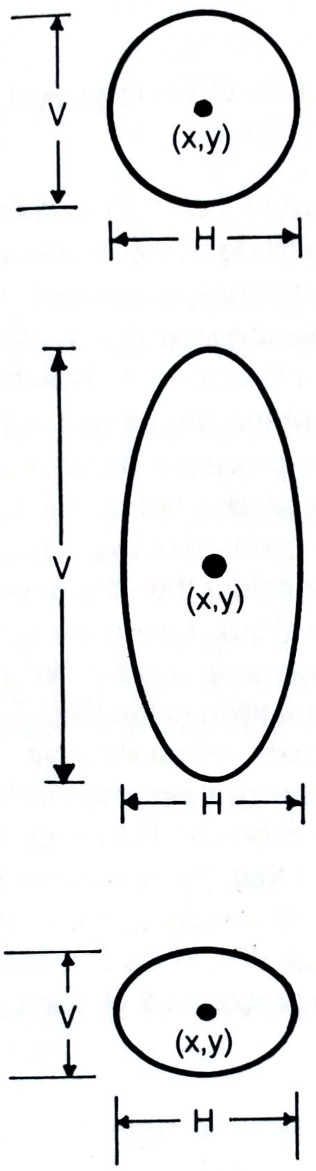


FIGURE 6.2 VARYING THE ANGLE ARGUMENT SUPPLIED TO CIRCLE

It is possible to draw segments of circles by specifying the finish angle parameter in the CIRCLE command as well has the start angle parameter. Thus drawing a circle with the start angle equal to 0 degrees and the finish angle equal to 90 degrees would give us a line describing a quarter circle. The length of the arc thus produced still depends upon the radius of the circle, and the position of the arc is set by the X,Y coordinates specified in the CIRCLE command. Again, experiment with varying values in the finish angle parameter, remembering that the CIRCLE command must be given its angles in radians.

The final parameter of the CIRCLE command, the aspect ratio, is used when we want to draw ellipses. As you may know, an ellipse is a 'flattened' circle. The aspect ratio of the ellipse is a measure of how flattened the ellipse is with respect to a circle. It is the measure of the vertical radius of the figure to the horizontal radius of the figure. In a circle, these two radii are the same, and a circle has an aspect ratio of 1. Figure 6.3 shows two ellipses with their respective aspect ratios. Very large aspect ratios will lead to a straight line, as will very small aspect ratios. The centre of the ellipse is again specified by the X,Y coordinates specified in the CIRCLE command.

One point of interest to all users of the CIRCLE command is that if the angles specified are negative, the angles will be treated as positive ones and the perimeter of the circle or ellipse will be connected to the point X,Y.



ASPECT RATIO $= \frac{V}{H} = \frac{1}{1} = 1$

ASPECT RATIO $= \frac{2}{1} = 2$

ASPECT RATIO $= \frac{Ø.5}{1} = Ø.5$

FIGURE 6.3 VARYING THE ASPECT-RATIO ARGUMENT SUPPLIED TO CIRCLE

Paint

We know how to draw circles or lines in colour by specifying the colour parameter in the command. We can plot pixels in colour in a similar way. However, what happens if we want to fill a circle or some irregular shape with colour? One possibility would be to plot every pixel in the shape in the colour required. This would work, but in BASIC would take a long time to achieve. MSX BASIC provides us with a routine in ROM that works on this principle but in machine code, which means that it is many times faster than BASIC. This routine is called PAINT, and the command has the syntax shown below.

```
PAINT (X,Y), colour, colour to be regarded as border.
```

The STEP (X,Y) form can be used in place of the (X,Y) expression. Both colour, which is the colour with which we wish to fill the shape, and border colour, which will be explained soon, are optional parameters. So, how do we use this command? The X,Y coordinate specifies the position at which the computer is to begin filling the shape with colour. It does this on a line by line basis, and recognises the edges of the area that it is to fill in the following fashion. As the PAINT operation proceeds along a line, it tests the colour of any pixel it encounters. If the pixel is not of the colour that is recognised as a border colour, then the pixel colour is changed to the paint colour. If it is a pixel that is the border colour then the operation goes no further along that line in that direction. Thus if we draw a circle, and position the X,Y coordinates of the PAINT command in the circle, the circle is filled with colour. If the coordinate pair specified in the PAINT command is outside the circle, then the rest of the screen will be filled with colour but the inside of the circle will not be. These examples assume that the circle is drawn in a colour that will be treated as a border colour by the PAINT routine. An important point to note here is that the parameter specifying the border colour is disregarded in screen mode 2. Here, the border colour is the same as the paint colour in use. In mode 3 a border colour can be specified. Let's now look at a few programs that demonstrate the potential of the PAINT command.

```
10 SCREEN 3
20 CIRCLE (100,100),40,15
30 PAINT (100,100),5,15:REM set different border
40 REM and paint colours
50 GOTO 50
```

Note how we set the border colour to the colour in which we drew the circle. Run this program, then add the line below which will fill in the background in a different colour.

```
45 PAINT (100,190),1,15
```

Again we've set the border colour to the colour in which the circle was drawn. Due to the fact that this command has directly specified the border colour, it will not work properly in screen mode 2. Below is a demonstration of the use of paint in mode 2.

```
10 SCREEN 2
20 CIRCLE (100,100),40,5
30 CIRCLE (100,100),50,15
40 CIRCLE (100,100),60,15
50 CIRCLE (100,100),70,1
60 PAINT (100,100),5
70 PAINT (100,152),15
80 PAINT (100,190), 1
90 GOTO 90
```

It is therefore clear that more care has to be taken when using the PAINT command in mode 2; the colour which we draw the outline of the shape to be filled in in mode 2 must be the same as the colour in which we intend to fill the shape in with.

The final simple graphics command that we shall look at is called POINT. This command allows us to find the colour of a pixel at a given screen position. The syntax is

```
POINT (X,Y)
```

and the statement containing the POINT command is usually one that assigns a value obtained from the point command to a variable. The POINT command returns a value between 0 and 15. X and Y give the position of the point of interest. A typical example of the use of POINT is in the statement

```
200 IF POINT(x,y)=1 THEN GOSUB 2000 ELSE
    GOSUB 3000
```

GRAPHICS MACRO LANGUAGE

This rather grand title is given to a series of graphics commands based around the DRAW command. The DRAW command and its associated string of functions do indeed form a language within BASIC. The syntax of the DRAW command is

```
DRAW string of graphics commands
```

The DRAW command only works in the two graphics modes, and any command passed to the DRAW command in the string of graphics commands will start drawing from the last point referenced — this is simply the last point drawn to or plotted by a graphics command of any type.

The string of graphics commands consists of certain letters and numbers, along with certain non-alphanumeric characters. The simplest commands in the graphics macro language are those for drawing straight lines. In each of the commands listed below, n is the number of pixels that you want to draw over.

Command	**Function**
U n	Draw upwards on screen
D n	Draw downwards on screen
L n	Draw to left on screen
R n	Draw to right on screen
E n	Draw diagonally, up and right
F n	Draw diagonally, down and right
G n	Draw diagonally, down and left
H n	Draw diagonally, up and left

The demonstration program below draws a square. Note how we use the PRESET command to position the square and define the colour to be used.

```
10 SCREEN 2
20 PRESET (100,100),15
30 DRAW "U50L50D50R50"
40 GOTO 40
```

Should we wish to draw lines that are at a particular angle, then we can use the M X, Y command, which is similar in many ways to the LINE command that we've already met. If we wish to use the M command to draw to a point relative to the last point visited, rather than 0, as we do when we use the STEP command in LINE, then we prefix the X or Y coordinate with a + or —. For example,

```
1000 DRAW "M +10,100"
```

would move to a point 10 pixels to the right of the current graphics position and 100 pixels down the screen from the current position. The negative sign enables us to reference a point to the left of the current X position and 'above' the current Y position. Always remember that the M command draw a line as it moves in the current foreground colour. If you wish to move to a point on the screen without drawing a line, then any of the commands that we've seen so far can be prefixed with the letter 'B'.

The 'N' command allows you to draw a line with any of the commands that we've previously examined, and then return the current graphics position to where it was before the command prefixed with the N command was executed. Thus the command

```
DRAW "M100,100NU50"
```

will draw a line first to position 100,100, and then draw a line to position 100,50. However, the next line to be drawn will take as its start position 100,100, which is the position of the start of the N prefixed command. To make this clearer, type in the program below, and run it.

```
10 SCREEN 2
20 PRESET (100,100),1
30 DRAW "NU100NL100NL100NR100ND100"
40 GOTO 40
```

The N before every command in the string has the effect of every command in the string drawing from the same start position.

Colour

Colour changes in the graphics macro language are easy to perform. The character 'C' is inserted into the command string, followed by a number between 0 and 15 which represents the colour wanted. The colour code numbers are those which we've previously discussed for use in the text and graphics modes. Thus the line of instructions below will cause a black square to be drawn.

```
DRAW "C1U50L50D50R50"
```

The colour commands can be placed anywhere in the DRAW command string.

Another command that is of use is the Angle command, 'A'. One drawback with this command is that there is not a terribly large number of different angles available to you; 4 in fact! The syntax within a DRAW command string is An, and n has a value according to the table below. As in the CIRCLE command, angles are measured in an anti-clockwise fashion.

Value of n	Angle
0	0 degrees
1	90 degrees
2	180 degrees
3	270 degrees

The demonstration below will draw a square using the angle commands instead of the D, L and R commands.

```
10 SCREEN 3
20 PRESET (100,100),1
30 DRAW "A1U90A2U90A3U90A0U90"
40 GOTO 40
```

Scale Factor

In normal circumstances, the n parameter that we pass over to commands such as U or D, corresponds directly to the number of pixels that you want drawn over. A command of the form Sn, where n is a value between 0 and 255, enables us to vary the number of pixels represented by a certain value of n in these commands. The value of n over 4 is the scaling factor, and so if we have a value of n = 1, a figure will be drawn that is 1/4 the size of the figure drawn with no scaling commands involved. Similarily, if we set n = 8 by issuing the command S8, then the figure drawn will be twice the size of the one we would get without the scaling factor.

Subroutines

It would be nice if we could have, within the graphics macro language, the equivalent of the BASIC subroutine; that is, a means of keeping only one copy of a set of commands but being able to call it into use whenever we wanted to. We can, in fact, do this by using the X command and string variables. For example, we might set sq$ to contain the commands that draw a square. We could then call this into use in another string as part of a DRAW command using the command

```
Xsq$;
```

The full commands needed to draw a square in such a way would be as follows:

```
A$="U30R30D30L30"
DRAW "XA$;"
```

The program below shows this in greater detail, and shows how a string used in an X command can itself call other strings with X commands.

```
10 SCREEN 2
20 a$="U30"
30 b$="Xa$;L30D30R30"
40 PRESET (100,100),1
50 DRAW "Xb$;"
60 GOTO 60
```

Thus by using the X command, it is possible to assemble strings of commands for the DRAW command that are, in effect, more than 255 characters long. Any string variable name that we use in this way must be followed by a semi-colon.

Variables in the graphics macro language

In all the commands that we've discussed above, the numerical parameters have all been constants. This need not always be the case, as we can use a variable that has been assigned a variable in the normal way in the BASIC program in the DRAW command string. The variable name used is prefixed by the "=" sign, and is followed by a ";". Thus we could have:

```
100 up=100
110 DRAW "U=up;D30R30"
```

Expressions are not allowed in a string that is passed over to a DRAW command They must be evaluated and the values obtained assigned to a variable and the variable passed to the string.

In all these commands, the ";" can be used as an optional separater of commands. Any spaces are ignored by the DRAW command when the string is interpreted. As must by now be fairly obvious, the graphics macro language can only be used in graphics modes.

SPRITES

We've seen how we can write text to the graphics screen, and how we can draw lines and circles using the graphics commands. However, once we start writing games software, for example, we begin to want characters that look like space invaders, giant cucumbers, or whatever our game is about! We can create these characters for use in graphics modes quite easily. They are called SPRITES and are extremely useful in graphics programming. A sprite can be defined in simple terms as a character whose shape we can define and that we can move at will on the screen. We can also tell if two sprites collide with each other as they travel around the screen. They can be bigger than normal characters, and a sprite can be made up of more than one character. No matter how many characters a sprite is composed of, the computer treats it as a single entity for the purposes of BASIC programming.

Before we go and examine how we can define and use sprites, it will be useful to look at how the Video Display Processor looks at sprites.

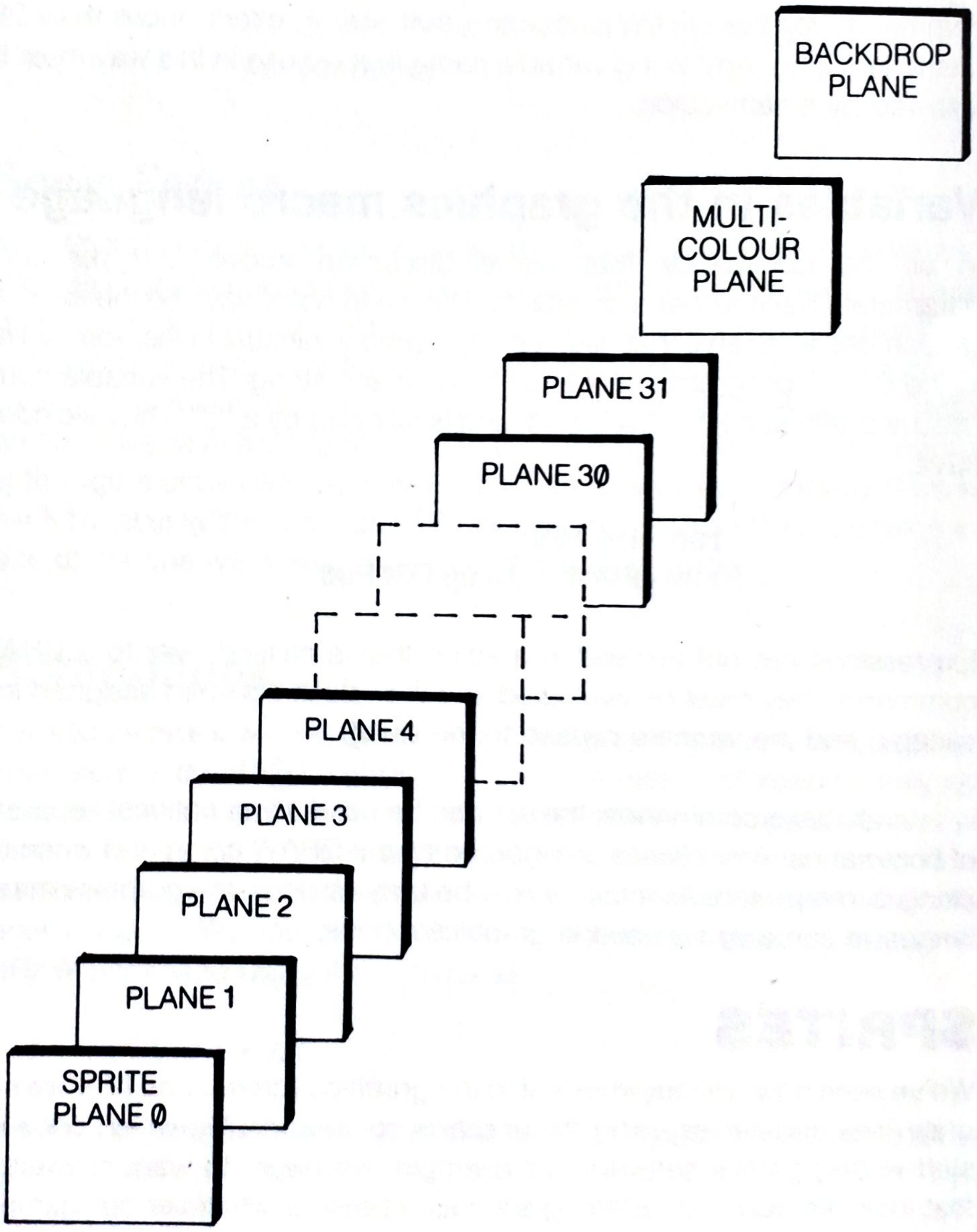


FIGURE 6.4 REPRESENTATION OF THE SPRITE, MULTICOLOUR AND BACK DROP PLANES

The picture that the VDP presents to the television screen or monitor can be imagined as being made up of a series of layers, which are called PLANES. These are shown diagrammatically in Figure 6.4. Images that are on the Display Planes closest to the observer appear to pass in front of images that are on Display Planes further back. This is what gives sprites their useful ability to pass in front of or behind other sprites. Note that it is not possible for a sprite to pass behind an object that is on a Display Plane that is further back than the sprite in question. The Display Planes closer to the viewer are said to have a higher PRIORITY than those planes that are further away from the viewer. Thus the Display Plane upon which Sprite Ø is seen has a higher priority than the plane on which Sprite 3 is normally seen. There are 32 of these Sprite Display Planes; this explains why it is not possible, except by the use of very advanced programming techniques, to have more than 32 sprites on the screen at once, as there is only one sprite per Plane. After the Sprite Planes comes a Display Plane known as the Multicolour Plane. It is on this plane that the images created by the normal text handling and graphics commands are displayed. Thus images that are produced by commands such as PRINT and DRAW always have a lower priority than images that are sprites. The final Display Plane that is accessible from software is called the Backdrop. We normally see this as the border to the display screen.

Sprites are not available in Mode Ø, but are available in the other modes, including text mode 1. Sprites can be of various sizes on the screen. The smallest sprite size is the 8 by 8 pixel sprite; this gives an image size the same as that given by printing a text character to a Mode 2 screen. The other sprite size is 16 by 16 pixels. Both of these sprites can be 'magnified' on the screen, thus making them bigger. There are thus 4 different sprite sizes available to the MSX programmer. A sprite's colour is set when we position it on the screen. If a pixel of a sprite is not coloured in, then we can see the images present on lower priority planes through the transparent pixels of the sprite. Thus if one sprite is over another sprite, the colour of the lower priority sprite will be visible through any transparent areas of the higher priority sprite.

The information that is required to define a sprite is stored in Video RAM as an area of bytes called a SPRITE GENERATOR TABLE. A detailed examination of such a table will come later in the chapter. As there is always a given amount of memory available for defining sprites, the larger the sprite is the fewer sprites we can define, as a 16 by 16 pixel sprite takes more memory to define than an 8 by 8 pixel sprite does. The memory needed to define an 8 by 8 pixel sprite stays the same whether

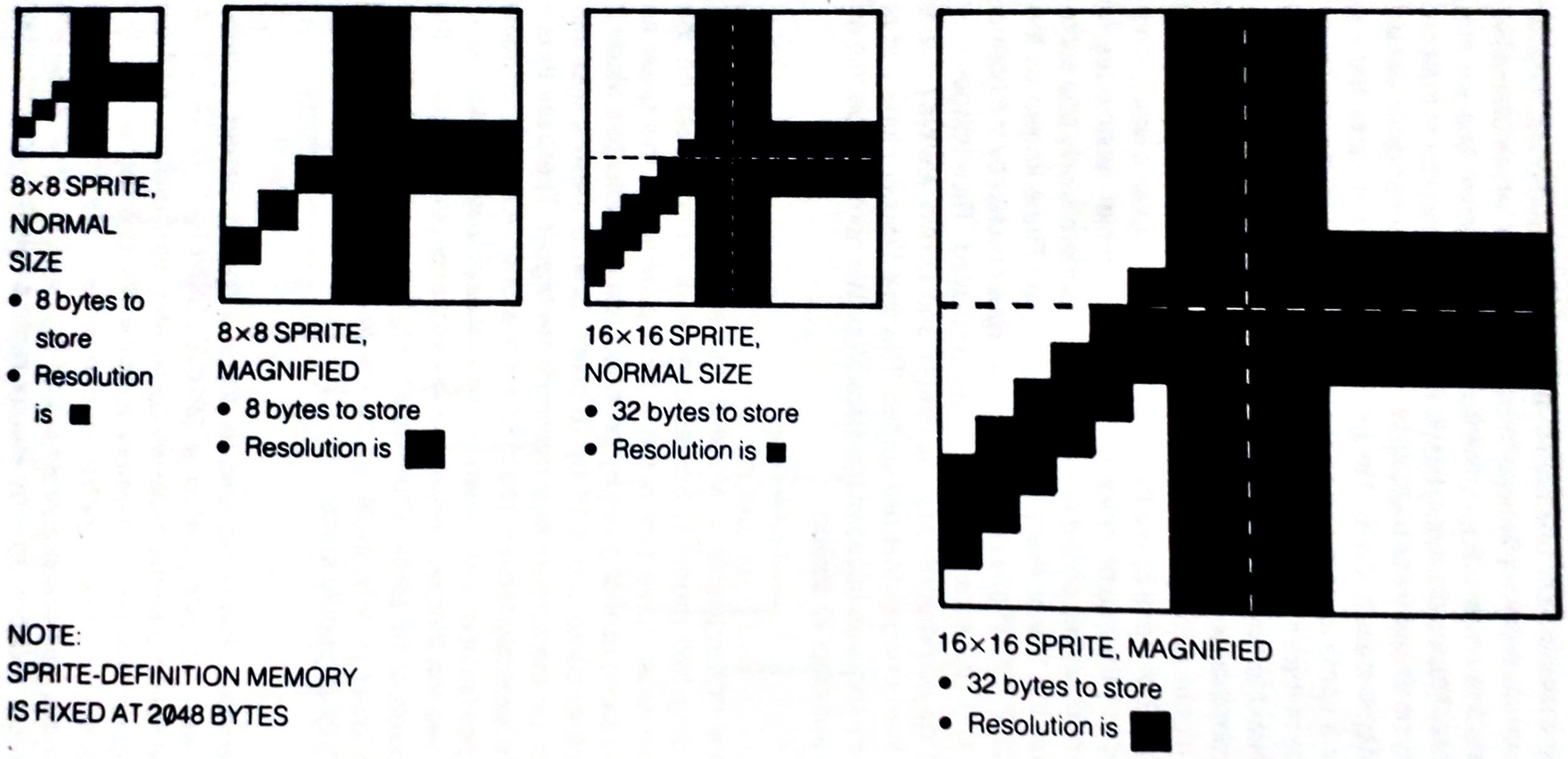


FIGURE 6.5 SPRITE DEFINITION (1): SIZE, MEMORY AND RESOLUTION TRADE-OFFS

the sprite is magnified or unmagnified. We can have up to 256 separate sprite patterns for 8 by 8 sprites and up to 64 separate patterns for 16 by 16 sprites.

Sprite Size

The size of sprites that will be displayed on the screen is set by a SCREEN command of the form:

```
SCREEN mode, size
```

where mode is a screen display mode that is capable of displaying sprites and size is a number between Ø and 3. The sizes of sprite specified by the size parameter are as follows.

Size	**Sprite size**
Ø	8 by 8 unmagnified
1	8 by 8 magnified
2	16 by 16 unmagnified
3	16 by 16 magnified

Figure 6.5 shows the appearance of the different sprite sizes on the screen.

An 8 by 8 sprite can be defined by simply defining a single character, but the 16 by 16 pixel sprite requires the user to define four 8 by 8 pixel sections, each section representing one quarter of the complete sprite. Thus using 16 by 16 pixel sprites we can define large images that we can move around the screen just as if they were 8 by 8 pixel sprites. So, how do we go about defining sprites?

Sprite Definition

No matter what size the sprite is, the basis of sprite definition is the use of an 8 by 8 grid of squares, as shown in Figure 6.6. Note the numbers running along the top of the grid; those of you who have consulted the appendix will recognise these numbers as being powers of 2. Armed with this grid, on which we can represent an 8 by 8 pixel sprite, we can start defining the sprite. We do this by shading in each square of the grid that we will want to appear in the foreground colour when the sprite is displayed. Thus, if we wanted to have a sprite that represented a little man, we could end up with the grid shaded in as in Figure 6.7.

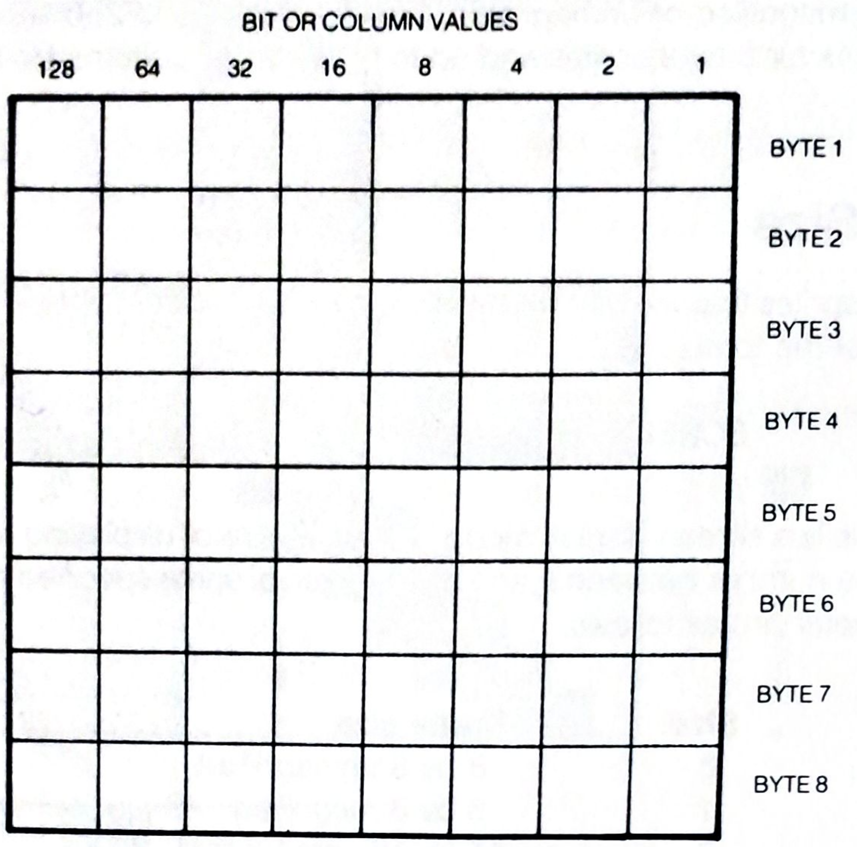


FIGURE 6.6 SPRITE DEFINITION (2): MAPPING ON 8 x 8 BLOCK

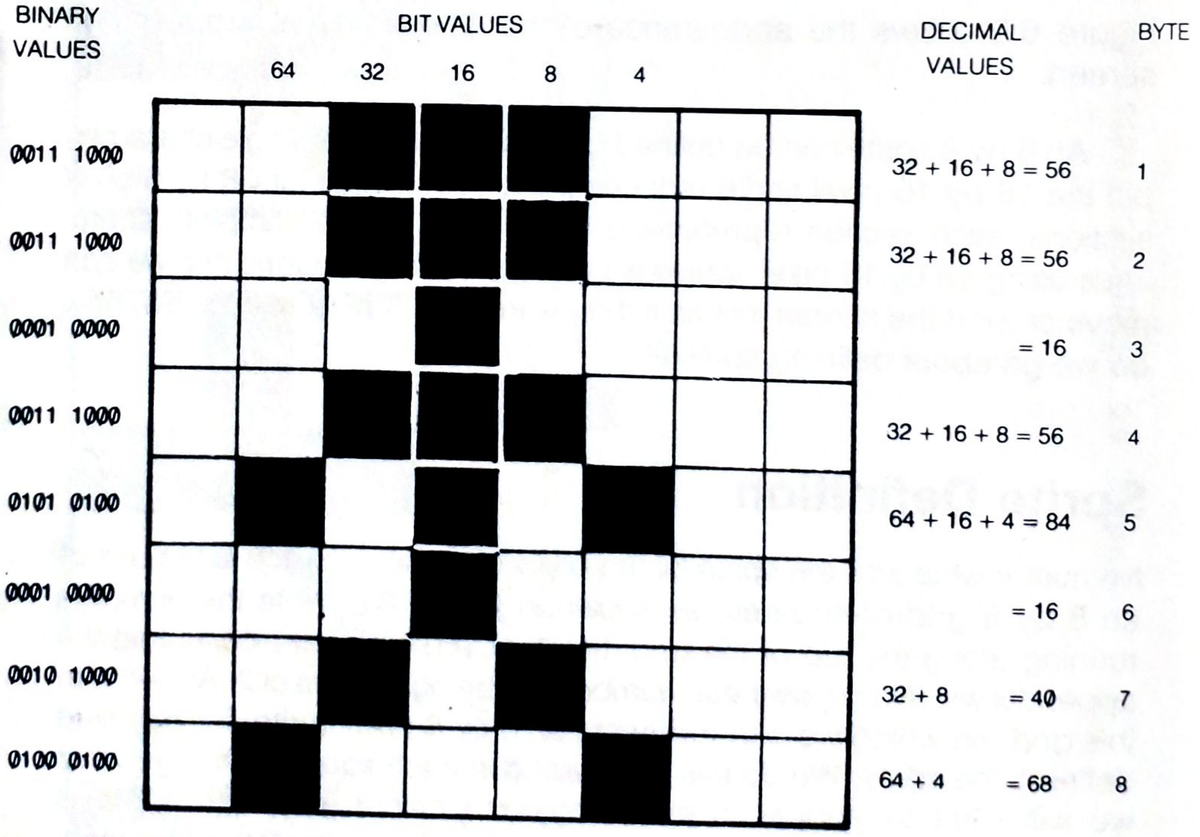
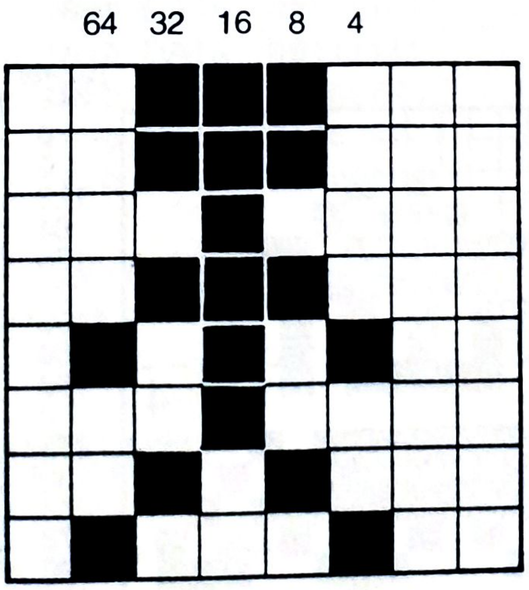


FIGURE 6.7 SPRITE DEFINITION (3): TRANSLATING THE 8 x 8 SPRITE SHAPE

To get this into a form that is understandable by the computer we use a series of special string variables called SPRITE$(n). n is an integer between 0 and 255 when the sprite size is 0 or 1 and between 0 and 63 when the sprite size is 2 or 3. These variables are used to hold the pattern for a sprite, and are 32 bytes long in memory. 8 bytes are needed to define an 8 by 8 pixel sprite, and 32 bytes are needed to define a 16 by 16 pixel sprite, which can be thought of as four 8 by 8 pixel sprites. SPRITE$(1) holds the data defining the sprite that normally is displayed on sprite Display Plane 1. The SPRITE$ variables are thus a means of accessing the Sprite Generator Tables from BASIC.

To transfer the information held in the grid into a SPRITE$ variable, we convert each row of the grid into a number. This is shown for the little man in Figure 6.8. These numerical values are obtained by adding together the numerical values of the columns in the row that contain pixels that we want to set to the foreground colour when we display the sprite. Thus, in a given row, if only the leftmost pixel was to be in the foreground colour, the value assigned to that row would be 128. Once the values have been calculated for each row, we assign them to a SPRITE$ variable in the following fashion.

```
100 a$=CHR$(56)+CHR$(56)+CHR$(16)
     +CHR$(56)+CHR$(84)+CHR$ (16)+CHR$(40)
     +CHR$(68)
110 SPRITE$(1)=a$
```



a$ =

CHR$(56) +

CHR$(56) +

CHR$(16) +

CHR$(56) +

CHR$(84) +

CHR$(16) +

CHR$(40) +

CHR$(68)

FIGURE 6.8 SPRITE DEFINITION (4): ASSIGNING DATA TO THE SPRITE$ VARIABLE

The numerical contents of a given SPRITE$ variable can be examined by use of the routine below.

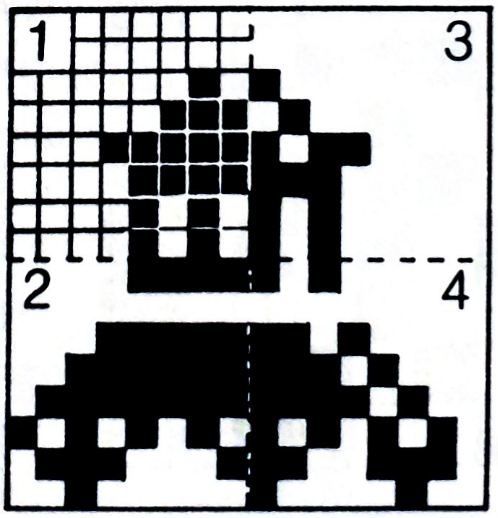
```
1Ø SCREEN 1
2Ø n=1
3Ø FOR I=1 TO LEN(SPRITE$(n))
4Ø PRINT ASC (MID$(SPRITE$(n),I,1))
5Ø NEXT I
```

You can assign a string of characters to a SPRITE$ variable in the usual fashion, as long as it is less than 32 characters long. Note that the contents of all the SPRITE$ variables can be deleted by the use of a routine which sets them all to a string of CHR$(Ø)'s. The SCREEN mode, size command also deletes all the SPRITE$ variables.

With regard to the 16 by 16 pixel sprite, the principles of sprite design remain the same. We simply design a separate 8 by 8 pixel sprite for each quarter of the large sprite, as shown in Figure 6.9. To define a 16 by 16 pixel sprite, we simply design each 8 by 8 pixel sprite in turn, following the numbering sequence in Figure 6.9. Once the sprites parts have been designed, the numerical values of each row are evaluated, and then these numerical values are placed into a SPRITE$ variable in the way that we've already seen. However, the order in which the numbers are assigned to the SPRITE$ variable is important, and below you can see the order in which the values are assigned to SPRITE$(n).

```
1ØØ SPRITE$(1)=string from quarter 1+string
    from quarter 2+string from quarter 3+string
    from quarter 4.
```

Quad-rant	Byte	Decimal Value
1	1	Ø
	2	Ø
	3	2
	4	7
	5	31
	6	15
	7	1Ø
2	8	1Ø
	9	15
	1Ø	Ø
	11	31
	12	63
	13	127
	14	17Ø
	15	113
	16	32



Decimal Value	Byte	Quad-rant
Ø	17	3
Ø	18	
128	19	
64	2Ø	
176	21	
224	22	
16Ø	23	
16Ø	24	
16Ø	25	4
Ø	26	
2Ø8	27	
232	28	
244	29	
17Ø	3Ø	
2Ø6	31	
132	32	

FIGURE 6.9 SPRITE DEFINITION (5): TRANSLATING THE 16x16 SPRITE SHAPE

To make the job of defining an 8 by 8 sprite slightly easier, try the program below. There are no frills to it and it can easily be improved upon. The 8 DATA statements at the end of the program represent the 8 rows of pixels in an 8 by 8 pixel sprite. Each '1' in these statements represents a pixel in the sprite in the foreground colour and each 'Ø' in them represents a pixel to be left blank in the finished sprite. Thus to change the sprite definition, simply edit the DATA statements and, when the pattern in the DATA statements is to your satisfaction, RUN the program. The numerical values that represent each row will then be printed out so that you can include them in your program. These numbers are in the correct order for direct inclusion in SPRITE$ assignments.

```
  10 SCREEN Ø
  20 RESTORE
  30 FOR J=1 TO 8
  40 READ n$
  50 char=Ø
  60 FOR I=8 TO 1 STEP -1
  70 X$=MID$(n$,I,1):n=VAL(X$)
  80 IF n=1 THEN char=char+2^(8-I)
  90 NEXT
 100 PRINT char
 110 NEXT
 120 END
1000 DATA "1ØØØ1ØØØ"
1010 DATA "Ø1Ø1ØØØØ"
1020 DATA "ØØ1ØØØØØ"
1030 DATA "ØØ111111"
1040 DATA "ØØ111111"
1050 DATA "ØØ1ØØØØ1"
1060 DATA "ØØ1ØØØØ1"
1070 DATA "ØØ1ØØØØ1"
```

Having defined our sprites, the next thing to do is to use them! To place them on the screen we use the PUT SPRITE command.

Positioning Sprites

The full syntax of the PUT SPRITE command is shown below, and of the parameters listed, colour, (X,Y) and pattern number are all optional.

PUT SPRITE plane number, (X,Y), colour, pattern number

If any of the optional parameters are omitted, then the current values are used. Thus if the colour and (X,Y) position were to be omitted, the colour used to display the sprite would be the current foreground colour and the position at which the sprite would be placed would be the last position accessed by a graphics command. The position of the sprite always refers to the top left hand corner of the sprite. The (X,Y) position of the sprite can also be defined in terms of the STEP (X,Y) command that we have already seen in use in the graphics commands. So, as a start with sprites, let's position a sprite in the middle of the screen. Before we look at a program to do this, a few notes about the range of values that is allowed for X and Y coordinates.

The X coordinate must be betwen -32 and 255; the Y coordinate is from -32 to 191. This is true in all the screen modes that support sprites, thus giving you a large degree of control over the positioning of sprites. Some values of Y give strange results; if we set Y to a value of 208, all sprites on Display Planes of lower priority disappear from the display. This situation remains in force until the Y value is changed. A value of 209 causes the sprite itself to disappear.

Anyway, let's now position our sprite on the screen. Try the program below.

```
 0 SCREEN 2,1
20 FOR I=1 To 8: READ n:a$+CHR$(n)
30 NEXT
40 SPRITE$(1)=a$
50 PUT SPRITE 1,(100,100),1
60 GOTO 60
70 DATA 56,56,16,56,84,16,40,68
```

Line 10 sets the mode of display and the sprite size, in this case an 8 by 8 pixel magnified sprite has been selected. This is positioned at position 100,100 on the screen by line 50, and is coloured black. The plane number here is 1, and this indicates that the sprite defined in SPRITE$(1) will be displayed at this position. Under normal circumstances, there is a direct relationship between the value of n in the SPRITE$(n) command and the Sprite Plane that particular sprite will be displayed on. Thus the sprite definition held in SPRITE$(1) will normally be displayed on Plane 1. Add the lines below to the program to demonstrate this; the sprite defined in

SPRITE$(2) will be of a lower priority than the one in SPRITE$(1) in this demonstration.

```
35 FOR I=1 TO 8:READ n:b$=b$+CHR$(n): NEXT
45 SPRITE$(2)=b$
55 PUT SPRITE 2, (100,100), 15
80 DATA 0,0,0,255,255,0,0,0
```

However, what if we want to place the sprite defined in SPRITE$(2) on a higher priority plane than the sprite defined in SPRITE$(1)? This is where the Pattern Number parameter in the PUT SPRITE comes in useful. Run the program listed below, and by pressing a key you will see how we can use the pattern number to position a sprite on a plane other than that to which it was originally assigned using the SPRITE$ command.

```
 10 SCREEN 1,1
 20 FOR I=1 TO 8:READ n:a$=a$+CHR$(n):NEXT
 30 FOR I=1 TO 8:READ n:b$=b$+CHR$(n):NEXT
 40 SPRITE$(1)=a$:SPRITE$(2)=b$
 50 PUT SPRITE 1, (100,100),1,1
 60 PUT SPRITE 2, (100,100),15,2
 70 a$=INPUT$(1)
 80 PUT SPRITE 1,(150,150),1,2
 90 PUT SPRITE 2,(150,150),15,1
100 a$=INPUT$(1)
110 GOTO 50
120 DATA 56,56,16,56,84,16,40,68
130 DATA 0,0,0,255,255,0,0,0
```

One of the most useful properties of a sprite is that it can be moved around the screen with the minimum of fuss. We do this by simply changing the X and Y coordinate. The sprite in question will then be deleted from its current position on the screen and moved to the new position specified by the new coordinates almost instantaneously. Thus is we only alter the coordinates by, say, 1, we can get the illusion of smooth, continuous movement. The program below moves the sprite from the top left of the screen to the bottom right.

```
10 SCREEN 1,1
20 FOR I=1 TO 8: READ n:a$=a$+CHR$(n):NEXT
30 SPRITE$(1)=a$
40 FOR I=0 TO 190
50 PUT SPRITE 1, (I,I),1
60 NEXT
70 GOTO 40
80 DATA 56,56,16,56,84,16,40,68
```

If we want to, we can use the ON INTERVAL GOSUB command to update the sprite position automatically. The program below demonstrates this in operation.

```
  10 ON INTERVAL=10 GOSUB 1000
  30 SCREEN 1,1
  40 FOR I=1 TO 8: READ n:a$=a$+CHR$(n):
     NEXT
  50 SPRITE$(1)=a$
  60 X%=128:Y%=96:XI%=1:YI%=1
  65 INTERVAL ON
  70 FOR I=1 TO 10000:PRINT I;:NEXT
  80 END
  90 DATA 56,56,16,56,84,16,40,68
1000 PUT SPRITE 1, (X%,Y%), 1
1010 X%=X%+YI%:Y%=Y%+YI%
1020 IF RND(2)*6>4 THEN XI%=-XI%
1030 IF RND(2)*4>3 THEN YI%=-YI%
1040 RETURN
```

This program quite usefully demonstrates the concept of timesharing, where the computer is apparently performing 2 tasks at once. This is useful in application where we might want to do extra jobs, such as redrawing a graphics background, whilst still moving the sprites around the screen.

We are not limited to just moving one sprite; we can move several around the screen using exactly the same sort of routines as we've seen here, just having a different set of X,Y coordinates for each sprite. However, if we are moving them using the ON INTERVAL method, remember to allow more time between each entry to the interval trap. If you don't you could well find yourself continually executing the interval trap. Also, when handling several sprites at once on the screen, remember that you can have no more than 4 sprites on the same line of the screen in the horizontal direction at one time.

Sprite Coincidence

When one sprite crosses another one on the screen, they are said to be COINCIDENT. When designing and programming graphics games, it is useful to know when two sprites coincide on the screen, and we can trap such an event when it occurs using the ON SPRITE command. When a pixel of one sprite on one plane coincides with a pixel of a sprite on a different plane, then the CPU is informed of the fact by the VDP and, if the

ON SPRITE trap is active, the trap subroutine is entered. The program below shows this in action. Note the SPRITE OFF command within the trap routine; this will disable the event until the next SPRITE ON command is executed. If this command were not present, then as soon as the two sprites coincided the trap outine would be entered, executed and returned from. However, if the two sprites were still in coincidence, the trap would be re-entered almost immediately with no other processing being done. Thus the SPRITE OFF command allows the position of the sprite to be changed after a coincidence has been detected and hence allows the program to continue without repeatedly calling the trap routine. The trap is turned on again in line 160.

```
 10 SCREEN 1,1
 20 FOR I=1 TO 8:READ N:a$=a$
    +CHR$(N):NEXT
 30 SPRITE$(1)=a$:SPRITE$(2)=a$
 40 X=100:Y=50:X1=100:Y1=70
 50 X3=2:Y3=2:X2=2:Y3=2
 60 ON SPRITE GOSUB 180
 80 PUT SPRITE 1, (X,Y),1
 90 PUT SPRITE 2, (X1,Y1),15
100 IF RND(2)*6>3 THEN X2=-X2
110 IF RND(2)*4>3 THEN Y2=-Y2
120 IF RND(2)*6>3 THEN X3=-X3
130 IF RND(2)*4>3 THEN Y3=-Y3
140 X=X2+X:Y=Y2+Y
150 X1=X3+X1:Y1=Y3+Y1
160 SPRITE ON
170 GOTO 80
180 BEEP:BEEP
190 PRINT "Ouch!"
200 SPRITE OFF
210 RETURN
220 DATA 56,56,16,56,84,16,40,68
```

That just about sums up how the MSX BASIC programmer can use BASIC commands and statements to control the Video Display Processor. We are now ready to move on to look at how we can extend the ability of our MSX computer by accessing the Video RAM and the VDP directly.

Direct Access of VRAM and the VDP

MSX BASIC comes equipped with some commands that are specifically for accessing Video memory and the VDP registers. Thus, we'll examine these commands first. As we've already seen, the command POKE is used to directly modify the contents of memory; in a similar fashion we use the command VPOKE to directly alter the contents of the Video RAM. The full syntax is

```
VPOKE address, value
```

where address is between Ø and 16384 and value is between Ø and 255. We use the VPEEK (address) command to find out what a particular Video memory location holds, address again being between Ø and 16384.

When it comes to directly accessing the Video Display Processor (VDP) registers, we use the command VDP(n), where n is the register number between Ø and 8. The VDP command enables us to see what the current value is in the write only VDP registers and also allows us to alter the contents of these registers. We can also use it to read the read only VDP register. The command

```
PRINT VDP(1)
```

will print to the display the current value held in register 1 of the VDP. If we wish to assign a value to a register, then we use the command below.

```
VDP(1) = value
```

This will set the register to the value on the right of the equals sign. One word of warning here; certain registers can cause the VDP to behave in an extremely peculiar fashion if they are assigned certain values. No damage will be done to the computer, but you may have to press reset to regain control of the machine! Anyway, let's now go on to look at the function of each VDP register, and see how we can use them to improve our programming techniques.

General Notes on VDP Registers

Before progressing into the programming of registers, you are advised to consult the appendix on Number Systems, paying particular attention to the section on Binary Numbers. This will make this section much easier to follow. The registers of the VDP can each hold an 8 bit number, which can have a value of between Ø and 255; this number that is held in the register can be treated as either indicating an address in VRAM or as a series of bits, each bit controlling some facet of the operation of the VDP. If the contents of a register are used to control the VDP operation, and we only want to alter one bit of the register, we must take care not to alter the other bits of the register by accident.

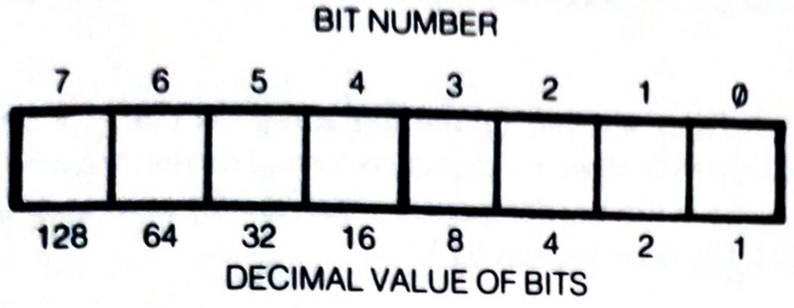


FIGURE 6.10 THE VIDEO DISPLAY PROCESSOR REGISTERS

Each bit position has, as you will know if you've read the appendix, a numerical value associated with it. These values are shown above. To avoid altering other bits of the register when we change one bit, we use the bitwise AND and OR operators, which we first encountered in Chapter 3. For example, to set bit 3 or register n to 1, whilst leaving the other bits as they are, we use the line of code:

```
VDP(n)=VDP(n) OR &BØØØØ1ØØØ
```

I have used binary here as it makes the bitwise OR operation more obvious. In a similar fashion, to set the same bit to Ø we use the bitwise AND.

```
VDP(n)=VDP(n) AND &B1111Ø111
```

This is quite important when dealing with, for example, VDP register 1, which is totally dedicated to controlling the VDP chip. If you wish to test whether a given bit in a register is set to 1 or Ø, then we can again use the bitwise operators, as we saw in Chapter 3.

Register Ø

Only bits Ø and 1 of this register are used at this time; all the higher order bits are being kept for future use by the manufacturer. Of these two bits, the only one of relevance to the MSX programmer is bit 1, which is involved with the screen mode selection. This bit is called M3, and we shall see how it is used when we look at Register 1.

Register 1

This is the main control register for the VDP. Some bits are quite useful, whilst a couple are best left alone.

Bit 7. This is one of the latter; it tells the VDP what chips are used for the video memory. If you alter it from what your particular computer needs, then the display is rendered unreadable.

Bit 6. This controls whether or not the screen is blank or whether it is active. If we set it to Ø, then the display is turned off and the only part of the screen left visible is the border colour, which is now seen in all parts of the display. This bit is normally set to 1.

Bit 5. This bit is called the VDP INTERRUPT ENABLE bit, and is a bit that is best dealt with carefully. In the MSX computers, the VDP generates an interrupt signal, every 1/5Øth of a second in the UK machines, that causes the CPU to do various 'housekeeping' jobs, such as read the keyboard, update the TIME variable and update the counters used by the ON INTERVAL command. If this bit is set to Ø then the interrupt is said to be disabled, and none of these tasks are carried out by the CPU. Thus setting this bit to Ø disables the keyboard read operation. Doing this from Direct Mode is thus fatal! The only way you will regain control of the computer is to reset it. However, it can be done from within a program, as the program will carry on executing regardless of the status of this interrupt. Obviously, the program should not request any input from the keyboard, as this would cause the program to wait for a keypress that will never come. Two possible benefits that can come from disabling this interrupt are as follows.

- i If you disable this interrupt before entering a very long loop, that contains no inputs or references to the TIME variable, it will cause a slight improvement in the execution time of the loop. Remember to set the bit back to 1 before going on with the program.
- ii If you are keen on program protection, then you can issue a command to disable this interrupt as part of an ON STOP trap. This is an extremely final way of preventing people looking at your program!

The interrupt is enabled by

```
VDP(1)=VDP(1) OR &BØØ1ØØØØØ
```

and is disabled by the command

```
VDP(1)=VDP(1) AND &B11Ø11111
```

Bits 4 and 3. Together with the M3 bit of register Ø, these two bits control the display mode selected by the VDP. Bit 4 is called M2 and Bit 3 is called M1. The screen mode that is selected depends upon the values of these 3 bits.

M1	M2	M3	Screen Mode
Ø	Ø	Ø	1
Ø	Ø	1	2
Ø	1	Ø	3
1	Ø	Ø	Ø

Bit 1. This selects the size of sprite in use. If it is set to Ø then the 8 by 8 pixel sprite size will be selected. If it is set to 1 then the larger, 16 by 16 pixel sprite size will be selected.

Bit Ø. This selects whether or not the sprites will be magnified or not; setting it to Ø will give unmagnified sprites and setting to 1 will give magnified sprites.

The rest of the VDP registers, with the exception of the status register, hold data about the position of various tables of information held in VRAM that are required by the VDP.

Register 2

This register holds a value between Ø and 15. The contents, when multiplied by &H4ØØ, give the start of what is called the NAME TABLE for a particular display mode. This will be discussed in greater detail when we look at VRAM allocation shortly.

Register 3

This register holds a value of between Ø and 255, and, when multiplied by &H4Ø, gives the address in VRAM of what is called the COLOUR TABLE for a particular mode.

Register 4

This defines the start address in VRAM of the PATTERN TABLE area of VRAM for a given mode. To get this VRAM address, the contents of the register are multiplied by &H8ØØ.

Register 5

The contents of this register, when multiplied by &H8Ø, gives the address in VRAM of the SPRITE ATTRIBUTE TABLE.

Register 6

When multiplied by &H800 the contents of this register give the VRAM start address of the SPRITE PATTERN GENERATOR TABLE.

Register 7

This register is of most use in screen Mode 0. The upper 4 bits of the register hold the code for the foreground colour that is active in Mode 0. The lower 4 bits hold the code for the background colour of the Mode 0 screen and the border colour in the other screen modes. The codes used to represent each colour are the same as those we use in the COLOR command. Try the program below, which will display all combinations of foreground and background colour. Remember that some of these combinations will be unreadable.

```
10 SCREEN 0
20 PRINT"Hello!!"
30 FOR I=0 TO 255
40 VDP(7)=I
50 TIME=0
60 IF TIME<20 THEN GOTO 60
70 NEXT
80 SCREEN 0
```

Line 80 will set the colour codes held in the register back to their default values; in this case, the normal value held in the register is 244.

The Status Register

This is the read only register of the VDP, and we cannot alter its value directly. The only way that this register changes its value is in response to changes in the operation of the VDP. It can thus only be accessed by such commands as

```
var=VDP(8)
```

or

```
PRINT VDP(8)
```

3 bits of the register are used to signal changes in the status of the VDP to the CPU. These bits are called FLAGS.

Bit 7. This is only of real use to the machine code programmer. It indicates that the VDP wishes to send an interrupt to the CPU. This interrupt will only be sent by the VDP if the Interrupt Enable bit of Register 1 is set to 1. If this bit is set to zero then the interrupt is not sent. If you do write programs that use this bit, then it is only set back to zero by reading the status register. If you don't read it, then the VDP will not be able to signal when it wishes to send its next interrupt.

Bit 5. This is called the COINCIDENCE flag; it is set to 1 whenever two sprites coincide, and is again only reset to zero by a read from the status register. Again, it is of real use only to the machine code programmer, as MSX BASIC makes full use of this facility with the ON SPRITE command. In UK MSX machines, that use the PAL television system, the VDP checks for sprite coincidence 50 times a second. As it is only cleared to zero by reading from the status register, any machine code programs that use this facility should read the status register fairly early on in the program to ensure that the flag is clear.

Bit 6. This is called the FIFTH SPRITE flag; not long ago I said that we could only have 4 sprites per horizontal display line; this flag is set to 1 whenever a display line is about to hold a fifth sprite. The number of the sprite that is causing the problem (the Sprite Plane Number, to be precise) is then stored in bits Ø to 4 of this register.

Machine code access to the VDP registers will be discussed in Chapter 11.

VRAM Usage in Screen Modes

The MSX system has 16k of memory dedicated totally to the VDP. This is called VRAM, and the way that it is allocated depends upon the screen mode that is in use. The memory used by a given Mode is used for various purposes, and each area of RAM has a name. The NAME TABLE is an area of VRAM that tells the VDP what image is to appear on a certain part of the screen. The way the contents of the Name Table are converted into actual screen images depends upon the Mode in use, as we shall soon see. The PATTERN GENERATOR TABLE is the area of VRAM that defines the image to be placed on the screen for a given value in the Name Table. The COLOUR TABLE informs the VDP of the colours to be used in that Mode. The SPRITE ATTRIBUTE TABLE and the SPRITE GENERATOR TABLE will be discussed later in this chapter. When we turn on our computer, the start addresses of these tables in memory are set by the ROM in our machine for a given Mode. The appropriate start addresses are then placed in the correct VDP registers. We can find these addresses in VRAM by the use of a command called BASE. This command is used to read the start addresses of the various tables in a given screen Mode, and its full syntax is shown below.

start = BASE(n)

where n is a value between Ø and 19. Some values of n in this range will return senseless results, but the valid n values will be mentioned when we come to look at the various Modes. These Table start addresses can be altered by writing to the relevant VDP register, not by the BASE command. So, on to the Modes.

Mode Ø

This is the 4Ø*24 text mode. It only has two Tables to deal with, as the colours used in this mode are stored in VDP register 7. BASE(Ø) will return the start address of the Name Table for this mode and BASE(2) will return the start address of the Pattern Table.

The Name Table holds information that is used by the VDP to work out which part of the Pattern Table is used to define the screen image at a particular position on the screen. The role of this table is effectively the same in each screen Mode, although its actual length may be different. The bytes of the Name Table correspond to screen positions, and the contents of the Table are used to access the Pattern Table in each mode.

In Mode Ø the Name Table is 96Ø bytes long, and is arranged with respect to the screen as shown in Figure 6.11. Thus the value held in location 2Ø of the Name Table will define the image displayed at column 2Ø of row 1 of the screen.

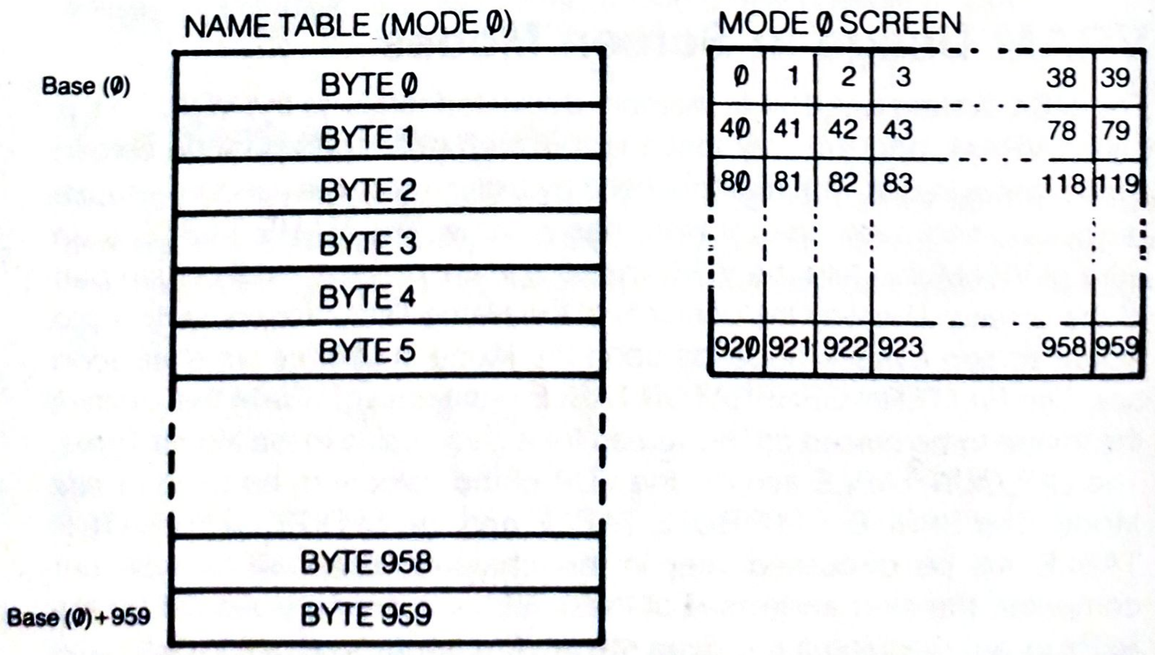


FIGURE 6.11 MAPPING THE NAME TABLE TO THE SCREEN, MODE Ø

The program below will write to each location in the Name Table, using the VPOKE command. Each location is set to hold the number 35, which is the ASCII code for "#". The numbers in the Name Table correspond to the ASCII codes of characters, and so this program fills the screen with "#" signs.

```
10 SCREEN 0
20 start=BASE (0)
30 FOR I%=start TO start+960
40 VPOKE I%, 35
50 NEXT
60 I$=INPUT$(1)
70 CLS
```

When the screen is full, simply press any key to go on. It is also possible to see what character is displayed on the screen at any point by using VPEEK to look at the appropriate location in the Name Table. In Mode 0, the various bytes of the Name Table can be referred to as Text Positions, because they do in fact hold textual character codes.

The Pattern Table holds the data needed for the VDP to convert the ASCII character code held in the name Table to an image on the screen, such as the "#" sign. For each character in the ASCII set, 8 bytes are needed to store the screen image. There are 256 separate ASCII codes, and the Pattern Table is long enough to include all these codes. It is 2048 bytes long in Mode 0. The first 8 bytes of the Table hold the information for the screen image of the character with the ASCII code of 0, the next 8 bytes the data for character 1, and so on, right up to the 8 bytes from location 2040 of the Table, which holds the data for the screen image of character 255. The data for the screen image is stored in an identical fashion to the way in which we defined sprites; the first of the 8 bytes representing a given character gives the data for row 1 of the character, the second byte codes the second row and so on.

The best way to explore the pattern Table is to try and alter the characters set that is normally displayed. Let's alter the "space" character in some way. Well, the first thing that we've got to do is find out where the "space" character definition begins in the Pattern Table. This is fairly easy;

```
space=BASE(2)+(32*8)
```

will return the start of the "space" character definition. As a more general method, the line below will return the start address within VRAM of the character definition for the character with the ASCII code n.

start = BASE(2) + (n*8)

This is only, of course, applicable to Mode Ø. This address will be the first row of the character definition. The below program will modify the first row of the "space" character definition.

```
10 SCREEN Ø
20 start = BASE(2)
30 VPOKE (start + (32*8) ),255
```

Run this program. Lo and behold, a lined screen! If we did the VPOKE to address start + (32*8) + 7 then the line would appear at the bottom of each space on the screen. Any character in the Mode Ø character set can be redefined in this way. As an example, let's redefine the character with the ASCII code 254 as the block shown in Figure 6.12.

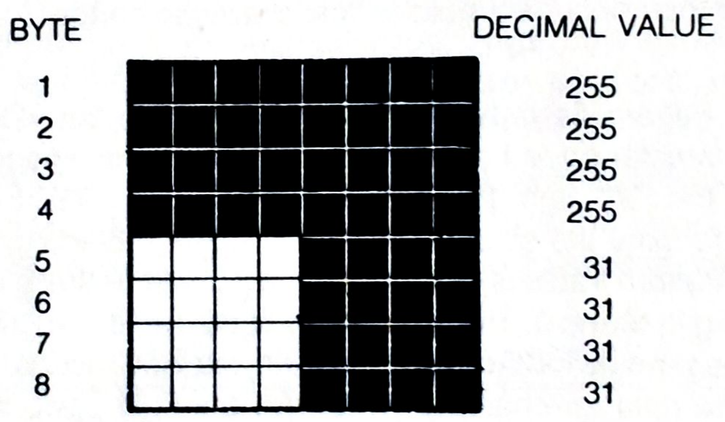


FIGURE 6.12 CHARACTER DEFINITION

Simply follow the grid procedure detailed in the section of this chapter concerning Sprite definition. I have done this, and the DATA statement in the program below shows the results. Note that in this mode, the characters appear to be based upon an 8 row by 6 column grid, the two right-most columns of the grid being disregarded for definition purposes.

```
10 SCREEN Ø
20 FOR I=Ø TO 7: READ n:VPOKE
     (BASE(2) + ( (254*8) + I) ),n
30 NEXT
40 DATA 255,255,255,255,31,31,31,31
```

Run the program, and then type in PRINT CHR$(254). If all has gone well, your little block should be printed to the screen. This new USER DEFINED CHARACTER can now be treated as if it were a standard ASCII character. There is only one drawback to defining characters in this way; when we execute a SCREEN Ø command, the computer automatically clears the Name Table, thus clearing the screen. It also resets all character definitions in the Pattern Table to what they were when the machine was turned on. Thus after each mode change any special characters required would have to be redefined. The colours that your new character will appear in are the same as the other characters. Any '1's in your defining bytes will be in the foreground colour and 'Ø's will be in the background color.

By altering the contents of the appropriate VDP register, it is possible to have two alternative pattern Tables in the memory at one time. Thus you can set up an alternate set of characters and switch between them at will by altering the value of VDP register 4. A similar thing can be done with the Name Table, enabling us to switch from one screen of data to another instantaneously. The program below shows this in operation. It could be vastly improved upon but the routine shows the principles involved. We are setting the new Name Table to start at location 12*1Ø24 in VRAM by setting register 2 of the VDP to 12. We then fill this area of RAM with the VPOKE command. Simply changing the value held in Register 2 displays the two different Name Tables to the screen alternately.

```
 1Ø SCREEN Ø
 2Ø VDP(2)=12:start=12*1Ø24
 3Ø FOR I%=start TO start+960:VPOKE I%,65
 4Ø NEXT
 5Ø PRINT "Hello'
 6Ø VDP(2)=Ø:REM default value
 7Ø I$=INPUT$(1)
 8Ø VDP(2)=12
 9Ø I$=INPUT$(1)
1ØØ GOTO 6Ø
```

Line 5Ø shows that we can write information to the normal screen by the PRINT command whilst displaying the contents of the alternative Name Table; this is because the ROM routines that handle writing to the screen write to specific locations within VRAM. Pressing a key will toggle between the two Name Tables and hence change the screen contents.

Mode 1

Another text mode, but with 3 Tables. These are the Name and Pattern Tables, which perform similar tasks to the same tables in Mode Ø, and the COLOUR TABLE, which holds information regarding the colours to be displayed. The start of the Name Table is given by BASE(5), that of the Pattern Table by BASE(7) and that of the Colour Table by BASE(6). There are also two tables in VRAM concerned with Sprites, but we will consider these later in the chapter. Suffice to say for the moment that the start address of the SPRITE ATTRIBUTE TABLE is given by BASE(8) and the start of the SPRITE PATTERN TABLE is given by BASE(9).

The Name Table in this mode is arranged in an analogous fashion to that in Mode Ø. Here, however, it is only 768 bytes long. It is mapped onto the screen in a similar fashion to the Mode Ø Name Table, and it can be accessed in a similar fashion using VPOKE and VPEEK. In fact the demonstration program for directly poking values into the Mode Ø Name Table will work in Mode 1 by altering BASE(Ø) to BASE(5), replacing 96Ø with 768 and replacing SCREEN Ø with SCREEN 1. We can change its start address by altering VDP Register 2 in this mode.

The Pattern Table also serves the same function as it does in Mode Ø, and we can redefine characters in this mode in the same way as in Mode Ø. We simply use BASE(7) to get the Pattern Table start address instead of BASE(2).

The main difference in this mode is the presence of the Colour Table, which increases the availability of colours to the MSX programmer. In the manuals, we are told that Mode 1 is a two colour mode; this is not strictly true, as we shall now see. The Colour Table is only 32 bytes long. Its start address in VRAM can be altered by writing a different value to VDP Register 3 whilst in this mode. Each byte in the Colour Table holds a value relating to the colours used in 8 characters defined in the Pattern Table. The 4 high bits of the Colour Table entry hold the foreground colour and the 4 low value bits hold the colour to be used as the background colour for the characters concerned. The first byte of the Colour Table holds the colour information for the first 8 characters defined by the Pattern Table; that is, ASCII codes Ø to 7. The second byte defines the colours for ASCII codes 8 to 15, and so on. Thus the 32nd table entry holds the colour information for the characters 248 to 255. Thus it is possible to change the values of each entry in this table so as to have all 16 colours available on the MSX machines on a Mode 1 screen at once! The drawback with this method is that the colours are "attached" to certain characters, but by careful selection of Colour Table entries some quite striking effects can be obtained. The program below shows how to alter the colours used in the characters with ASCII codes between 64 and 71.

```
10 SCREEN 1
20 start = BASE(6)
30 VPOKE (start + 8),18
40 END
```

Run the program, and then LIST it to the screen. Pretty, isn't it?

A total of 2848 bytes are required from the VRAM to use Mode 1 Name, Pattern and Colour Tables. Additional VRAM is needed if we want to use Sprites.

Mode 2

The way in which the VRAM is arranged in this mode is somewhat different to the way in which it arranged in the two text modes we've already looked at. This is to be expected, due to the high resolution graphics capability of Mode 2. It is quite a complex mode to understand, but go slowly through this section and you should (hopefully!) be able to comprehend the allocation of Mode 2 VRAM. The Name Table start address is given by BASE(10), the Colour Table start address is given by BASE(11) and the Pattern Table start address is given by BASE(12). The Sprite Attribute Table start address is given by BASE(13) and that of the Sprite Pattern Table is given by BASE(14).

The Name Table is still only 768 bytes long, and the locations in the Name Table map onto the screen in the same way as we've seen with Modes 0 and 1. However, the difference is that the image shown at the position on the screen depends not only on the contents of the corresponding Name Table location, but also on where in the Name Table, and hence on the screen, the image is to be placed. If this sounds a little complicated, don't worry; I'll try and make it clearer soon.

The Pattern Table in this mode is 6144 bytes long; this is 3 times as big as the Pattern Table in either of the other modes we've looked at, and it allows every location in the Name Table to have a totally unique code, thus enabling every location on the screen to be different from every other screen location. The Colour Table is also 6144 bytes long, and this allows each location in the Name Table to have a totally unique colour combination in it. It is the size of these two tables that gives us the ability for graphics in Mode 2.

The screen and the related memory Tables are best treated as being made up of 3 parts. This is shown diagrammatically in Figure 6.13.

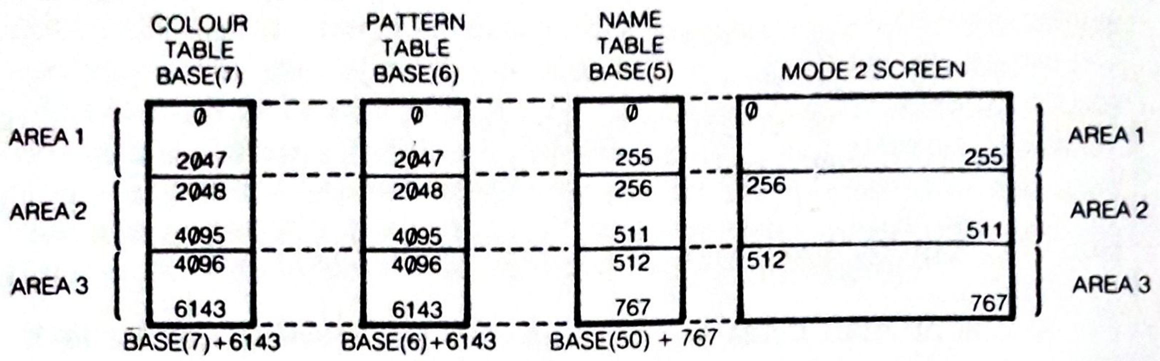


FIGURE 6.13 MAPPING COLOUR, PATTERN AND NAME TABLES TO THE SCREEN, MODE 2

The Pattern Table, despite being much larger, is arranged in a similar fashion to the Mode Ø Pattern Table; it is split into blocks of 8 bytes, each 8 byte block defining a pattern that can appear on the screen as an image. The first 2Ø48 bytes of the Pattern Table can thus be seen as 256 blocks of 8 bytes, each 8 byte block defining the image at one of the 256 Name Table locations in area 1 of the Name Table, and hence defining an image to appear at the corresponding position on the screen. Similarily, the next 2Ø48 bytes do the same job for area 2 of the Name Table and the final 2Ø48 bytes serve area 3 of the Name Table. Thus if the entry in location 2ØØ of the Name Table, which corresponds to screen location 2ØØ, was the number 2, the image that would be displayed at that screen location would be defined by the 16th to 23rd bytes of area 1 of the Pattern Table. In a similar way, if Name Table location 512, which corresponds to screen location 512, was to hold the value Ø, then the image displayed at that screen location would be defined by the first 8 bytes of area 2 of the Pattern Table — bytes 2Ø48 to 2Ø57 of the Pattern Table as a whole. Similar mapping exists between the Name Table and the Colour Table.

Two colours can be possessed by each byte of a Pattern Definition, the 4 high bits of the relevant entry in the Colour Table giving the foreground colour of that particular byte of the pattern, and the lower 4 bits of the relevant Table entry giving the background colour for that byte of the pattern being defined.

Let's now look at a couple of examples of directly accessing Mode 2 VRAM. The first example fills the top third of the screen with a pattern. Let's say that the code to appear in the Name Table wherever we want our character to appear is 0. As we are working in area 1 of the screen, we must put the data defining our character in the first 8 locations of the Pattern Table. Similarily, the colour codes for this particular character must go in to the first 8 locations in the Colour Table. In the program below, Line 20 defines the character from the DATA statement. Line 30 sets up the colour codes for the character. You might like to experiment here by putting different values in each byte of the Colour Table in use to give a technicolour character! Finally, line 40 of the program puts the code 0 into the first 255 locations of the Name Table, thus filling the top third of the screen with our character.

```
10 SCREEN 2
20 FOR I=BASE(12) TO BASE(12)+7: READ
   n:VPOKE I,n:NEXT
30 FOR I=BASE(11) TO BASE(11)+7:VPOKE
   I,18:NEXT
40 FOR I=BASE(10) TO BASE(10)+255:VPOKE
   1,0:NEXT
50 GOTO 50
60 DATA 56,56,16,56,84,16,40,68
```

You should be able to extend this program to poke little men to all parts of the screen by using the information given above.

One thing to note is that once you've defined a character in the pattern Table, the character defined immediately appears on the screen in the appropriate position. Thus placing a definition into bytes 0 to 7 of the Pattern Table causes the character thus defined to appear at screen location 0. Similar behaviour is observed with colours. To see this in action, try the program below.

```
10 SCREEN 2
20 FOR I=BASE(12) TO BASE(12)+7
30 READ n
40 VPOKE I,n: VPOKE (I+2048),n: VPOKE
   (I+4096),n
50 NEXT
60 GOTO 60
70 DATA 255,255,255,255,0,0,0,0
```

This program will put a marker on the screen to indicate the 0,256 and 512 image positions on the screen.

Movement can be programmed by directed VRAM access, as the following program shows. The principles are quite simple; we simply set a location in the Name Table to hold a value that corresponds to the required image definition in the Pattern Table for that area of the screen. We then pause, then set the location to hold a code with no pattern associated with it in the Pattern Table. After a slight delay, we repeat for the next location, and so on.

```
10 SCREEN 2:REM Reset the machine first
20 FOR I=BASE(12) TO BASE(12)+7:READ
   n:VPOKE I,n:NEXT
30 FOR I=BASE(11) TO BASE (11)+7:VPOKE
   I,18:NEXT
40 FOR I=BASE(10) TO BASE(10)+255
50 VPOKE I,0:FOR J=1 TO 30:NEXT
60 VPOKE I,1:FOR J=1 TO 30:NEXT
70 NEXT
80 GOTO 40
90 DATA 56,56,16,56,84,16,40,68
```

Pattern 1 in the Pattern Table is assumed here to be a space character, so that it will blank out the passage of the little man defined as Pattern 0.

The versatility of Mode 2 in terms of its high resolution graphics is reflected in the amount of memory taken up by the Pattern, Name and Colour Tables for this mode. 13056 bytes of screen RAM are taken up. This still, however, leaves us enough room to set up a second Name Table in VRAM by altering VDP register 2 in this mode. Take care when doing this not to overwrite parts of the Colour or Pattern Tables.

Mode 3

This is the final MSX screen mode, and gives us low resolution graphics with all 16 colours. BASE(15) gives the start address of the Name Table, and BASE(17) gives the start address of the Pattern Table. BASE(18) and BASE(19) give the start addresses of the Sprite Attribute Table and the Sprite Pattern Table respectively.

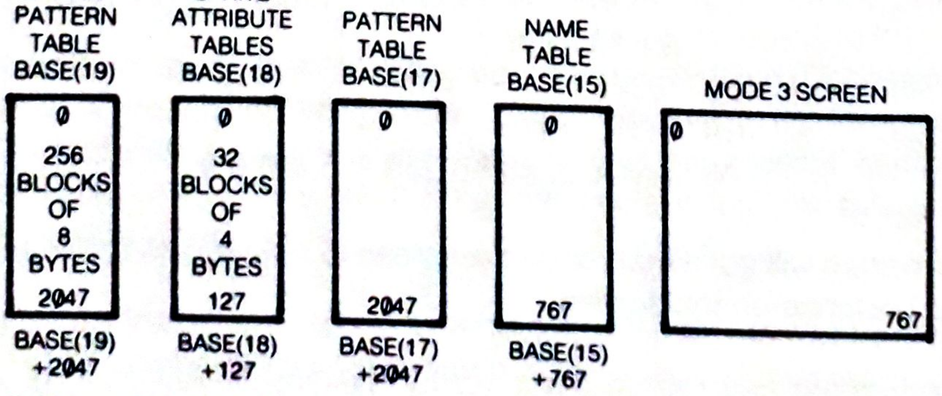


FIGURE 6.14 MAPPING THE GRAPHICS TABLES TO THE SCREEN, MODE 3

The observant amongst you will have noted the absence of a Colour Table in the above list. This is because in this mode, the Pattern Table holds data for both the image to be displayed and the colours in which it is to be displayed.

The Name Table in this mode is again 768 bytes long, each entry in the Name Table acting as a pointer to an 8 byte area of the Pattern Table and not the Colour Table as well. The Pattern Table is arranged in a rather peculiar fashion in order to code for both colour and image shape. It is 2Ø48 bytes long.

In Mode 2 we can plot single pixels to the screen; the Mode 3 resolution is not as good as this; in fact, it only allows us to plot 'super-pixels', which are 4 normal pixels by 4 pixels. These can be any colour we want, and due to the reduced resolution each character square on the screen only needs 2 bytes to define it, as shown in Figure 6.15. However, we've already said that an entry in the Name Table points to an 8 byte block in the Pattern Table, so what happens to the other 6 bytes? We'll find the answer to this question shortly.

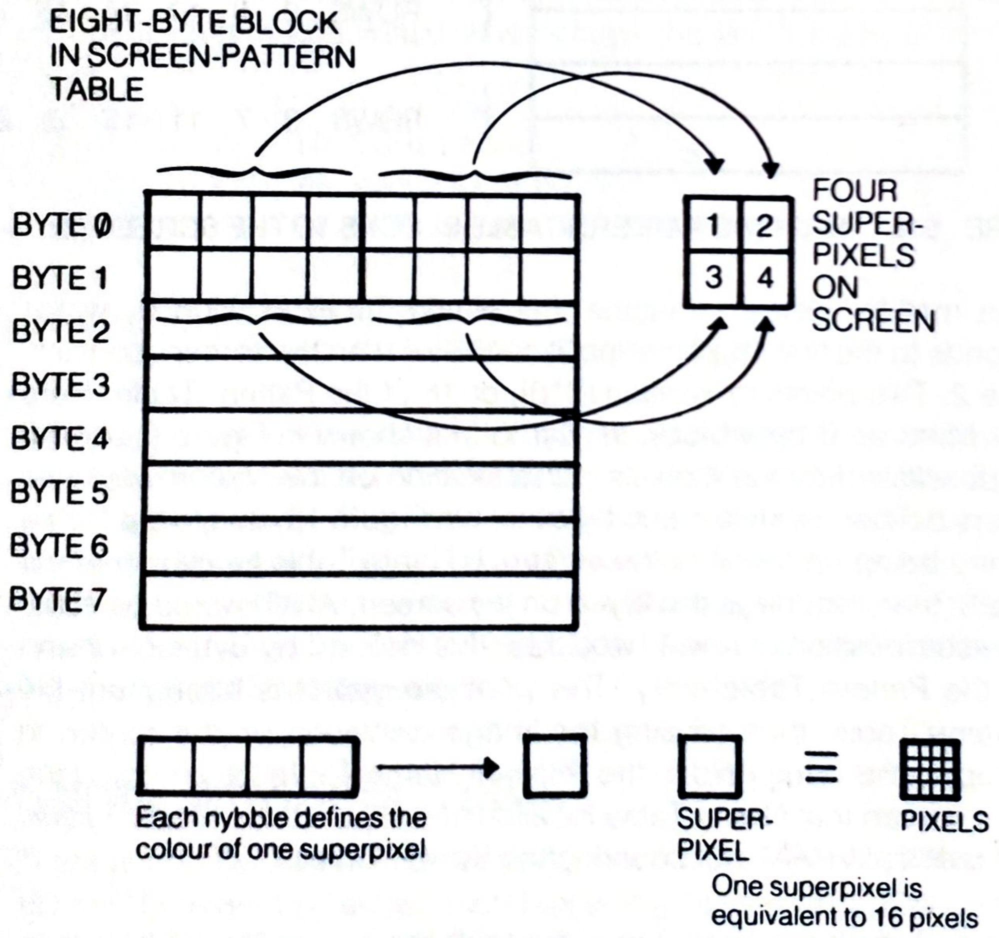


FIGURE 6.15 MAPPING PATTERN-TABLE BLOCKS TO THE SCREEN (1)

The 4 high bits of byte 1 hold the colour code for super-pixel 1, the lower 4 bits the code for super-pixel 2, the upper 4 bits of byte 2 code for the colour of super-pixel 3 and the lower 4 bits of byte 2 code for the colour of super-pixel 4.

Now, on to the missing 6 bytes. They are split into 3 similar groups of 2 bytes, arranged as above. Which two byte pattern definition gets to be displayed on the screen depends upon the screen row where the particular Name Table entry occurred. Figure 6.16 illustrates this.

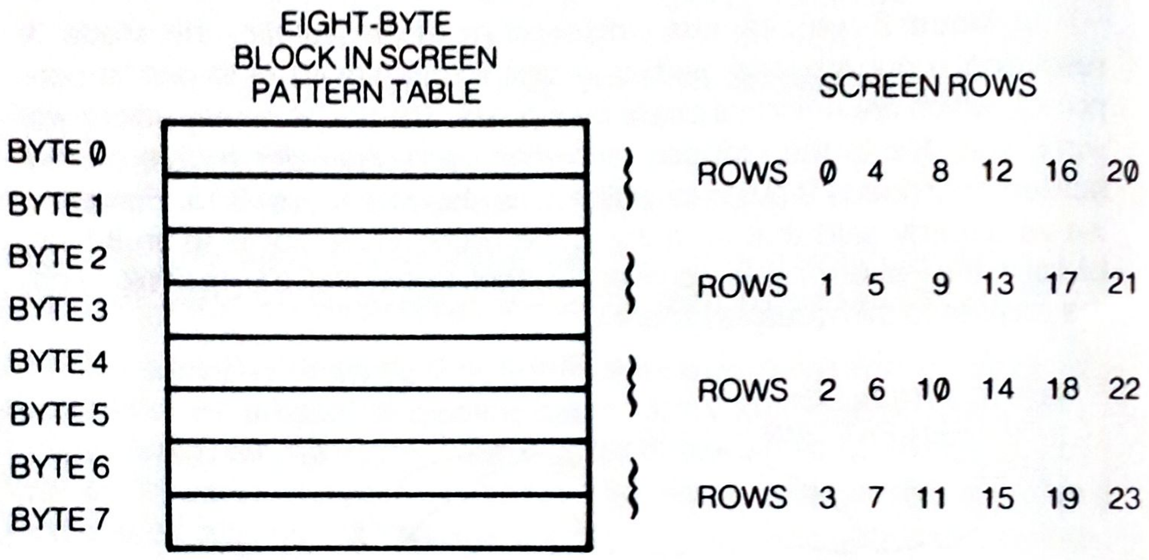


FIGURE 6.16 MAPPING PATTERN-TABLE BLOCKS TO THE SCREEN (2)

To make matters clearer, imagine that Name Table location Ø, which corresponds to the first character position of row Ø on the screen, contains the value 2. This points to location (2*8), or 16 of the Pattern Table. Here we would find an 8 byte block, similar to that shown in Figure 6.16. The colours possessed by the 4 pixels in this location on the screen would be the colours defined by byte n and byte n + 1 in Figure 13, due to the Name Table entry being on row Ø of the screen. If Name Table location 33 held the value 2, then the image displayed on the screen, which would be at the first character position of row 1, would be that defined by bytes n + 2 and n + 3 of the Pattern Table entry. This principle operates throughout the whole Name Table, thus causing the image displayed on the screen to depend upon the value held in the relevant Name Table location and the row of the screen that Name Table location corresponds to. Mode 3 takes up 2816 bytes of VRAM, not counting the Sprite Tables.

Let's now go on to look at how the VDP stores data about Sprites in Video memory.

Sprite Tables

There are two tables in VRAM that are set up by the VDP in each mode that supports Sprites. These are the Sprite Pattern Table, which holds the definition of the Sprites available, and the Sprite Attribute Table, which holds such information as colour and position for each sprite. The start address for each of these tables in a given mode is provided by the appropriate BASE command, as we have already mentioned when we were discussing the screen modes.

Sprite Pattern Table

This is a 2048 byte long area of VRAM that is functionally split into 256 blocks of 8 bytes. This table is accessed by the Pattern Number parameter in the PUT SPRITE command or the parameter n in the SPIRTE$(n) command. It holds the definitions of the Sprites and so we can define a Sprite by poking values into the Sprite Pattern Table directly instead of by the use of SPRITE$(n). The program below does this, defining Sprite Pattern 0 by poking the bytes making up the definition into the first 8 bytes of the Pattern Table. If we wanted to define Sprite Pattern 1 then we would poke the definition into locations 8 to 15 of the Pattern Table. The definition for Sprite Pattern 256 would thus occupy the last 8 bytes of the Pattern Table.

```
10 SCREEN 1,1
20 start=BASE(9)
30 FOR I=start TO start+7
40 VPOKE I,255
50 NEXT
60 PUT SPRITE 0,(100,100),1
```

Obviously, if we want to define 16 by 16 pixel sprites, 32 bytes of definition must be poked into the VRAM to define each quarter of the Sprite, as we've seen already. The need for more data to define a 16 by 16 sprite explains why we are limited to 64 of the larger sprites; there is always the same amount of VRAM available for the storage of Sprite definitions. It is possible to change the start address of the Sprite Pattern Table by altering the contents of VDP Register 6.

Sprite Attribute Table

There is one Sprite Attribute Table for each Sprite Plane; as there are 32 planes, there are 32 Tables, each one being 4 bytes long. This gives a total length for this Table of 128 bytes. Each Table is arranged as shown in Figure 6.17.

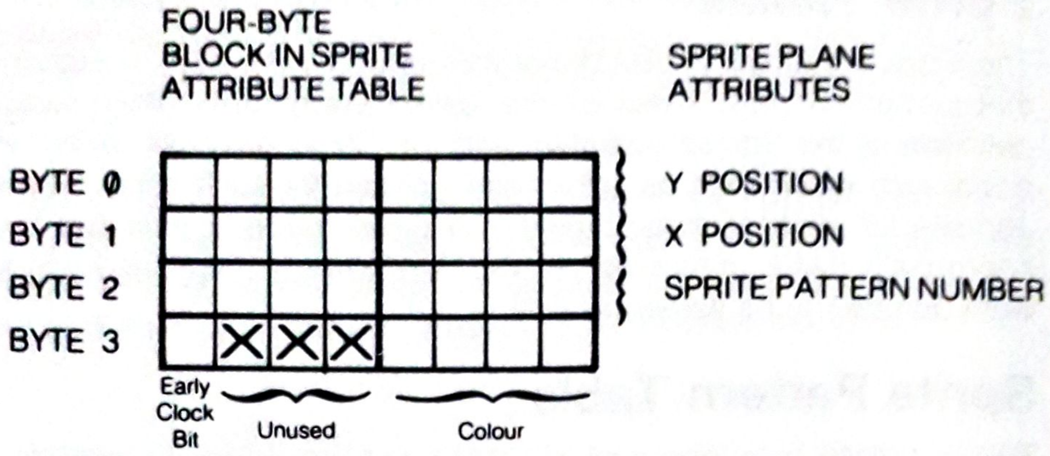


FIGURE 6.17 FUNCTIONS OF BYTES IN SPRITE-ATTRIBUTE BLOCK

The first 4 byte block in the Attribute Table refers to Sprite Plane Ø, the second 4 bytes to Sprite Plane 1 and so on. As to the Table entries, the Pattern Number is the number of the Sprite that is to be displayed on that plane, and the colour is the colour of the sprite when displayed. The first byte of the table contains the Y coordinate and the second byte contains the X coordinate of the Sprite. The only part of the Table that may cause problems to the unwary is the Early Clock Bit. If we set this bit to Ø then the top left corner of the Sprite is positioned at screen position X,Y. However, if it is set to 1, the top left corner of the sprite is positioned at position X-32,Y.

The program below shows how we can place a Sprite on the screen by using the VPOKE command to directly access the Sprite Attribute Table for the Sprite Plane of interest instead of the PUT SPRITE command.

```
10 SCREEN 1,1
20 start=BASE(9)
30 FOR I=start TO start+7:VPOKE 1,255
40 NEXT
50 start=BASE(8)
60 VPOKE start,100:REM Y position
70 VPOKE start+1,100:REM X position
80 VPOKE start+2,0:REM pattern number 0
90 VPOKE start+3,1:REM colour black
```

As you can see, handling sprites in this way is a little cumbersome, but the principles outlined here are of much greater use to the MSX machine code programmer.

A total of 2176 bytes are needed for the two Sprite Tables, assuming that all 256 Sprite Patterns are in use. If this is not so, then the appropriate VDP Registers can be changed in order to produce alternate Attribute Tables or Pattern Tables. Remember not to overwrite other areas of VRAM.

That finishes this Chapter on the VDP; we shall encounter it again, rather briefly, in Chapter 11 when we will discuss how to write to the VDP Registers and VRAM from machine code programs.

7 Joysticks

One of the interesting things about the MSX system is that as well as giving the user the possibility of plugging joysticks into the computer, MSX BASIC also provides the commands needed to handle them from BASIC. It also allows the programmer to simulate joysticks using the Cursor keys and the Space bar. This is obviously of great use for the games programmer, especially if the machine being used has large Cursor keys.

The command used to read which direction the user is pushing the joystick in is called STICK(n). n has a value between Ø and 2, Ø indicating that the command will read the Cursor keys as if they were the joystick. n = 1 refers to the joystick plugged in to port 1 and n = 2 refers to the joystick plugged in to port 2.

The value returned by the function depends upon the direction in which the joystick being read is being pushed by the user. If the Cursor keys are being read, then the value returned indicates the combination of Cursor keys being pressed at that time. Figure 7.1 shows the values returned for various combinations of Cursor keys pressed.

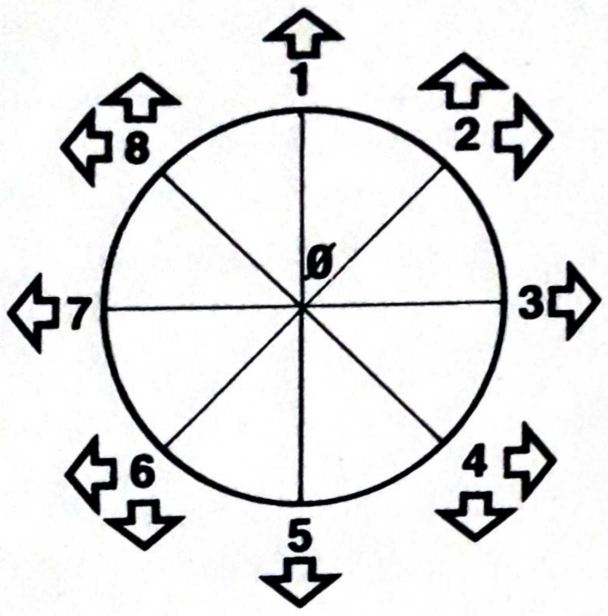


FIGURE 7.1 READING DIRECTION (1): FROM CURSOR-KEY COMBINATIONS

If no keys are being pressed when the function is evaluated, or if the joystick is not being pushed, then the function returns the value Ø.

Figure 7.2 gives the values returned for the direction in which the joystick is pushed. The directions are given as points of the compass.

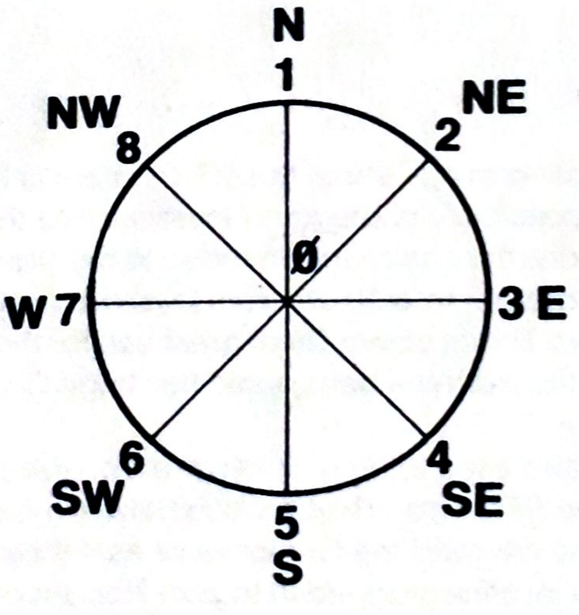


FIGURE 7.2 READING DIRECTION (2): FROM THE JOYSTICKS

If the joystick is not being deflected, then the value Ø is returned.

Proper joysticks plug into the sockets on the MSX computer marked A and B, these being accessed as ports 1 and 2 respectively. The electronic side of the joysticks is handled by the Programmable Sound Generator device, but this is only of real use to the machine code programmer.

ON STRIG

This command enables us to cause the computer to pass control to a subroutine whenever the trigger of a joystick or the space bar is pressed. The full syntax is

ON STRIG(n) GOSUB line number

where n is between 0 and 4 and line number is a valid line number to which control is passed when the appropriate trigger is pressed. With real joysticks, n = 1 or 3 will cause a jump to a subroutine when the trigger of joystick 1 is pressed, and n = 2 or 4 will cause the jump if the trigger on joystick 2 is pressed. If n = 0 then the jump will occur when the space bar is pressed.

STRIG(n) ON is used to enable the trapping of this event. STRIG(n) OFF and STRIG(n) STOP have analogous functions to the KEY(n) OFF and KEY(n) STOP commands; consult Chapter 5 for details.

We will now go on to look at the operation and programming of the Programmable Sound Generator chip in the computer.

8 The MSX Sound System

The MSX computers all contain a chip called the Programmable Sound Generator, or PSG. This device is usually the General Instruments AY-3-891Ø, and this device gives its sound output either through the TV set or through an external plug. This chapter will explore the programming of the PSG in BASIC, giving an introduction into the programming of tunes and sound effects.

The simplest BASIC command that uses the sound generator is BEEP. This produces the same tone as you obtain by typing CTRL-G. The command

```
                    PRINT CHR$(7)
```

will also generate the tone produced by the BEEP command. The other sound that our MSX computers make all the time is the Key Click, the noise that the computer makes whenever you press a key. The only thing that we can do with this click is to turn it on or off. The command

```
SCREEN mode, sprite size, Ø
```

turns the click off and the command

```
SCREEN mode, sprite size, 1
```

turns it back on. However, these noises are not exactly musical. But MSX BASIC comes complete with a command called PLAY, which, in a similar fashion to the DRAW command, implements a Music Macro Language.

This enables us to play music on the MSX computer with great ease, as we shall soon see.

PLAY

The full syntax for the PLAY command is

```
PLAY string for channel 1, string for channel 2, string
     for channel 3
```

The strings can be string constants or variables, as in the case of the DRAW command. The strings contain letters, numerals and some non-alphanumeric characters. The strings for channels 2 and 3 are optional. The PSG can play up to 3 notes at once, each note being of a different pitch and volume. One note is played on each channel, the channels being numbered from 1 to 3. The first group of characters that we can put in a PLAY command string are the letters A to G. These refer to musical notes. The command

```
PLAY "A"
```

will play the note A on channel 1. The octave to which the note belongs is the currently selected octave. The command

```
PLAY "ABCDEFG"
```

will play the notes one after another on channel 1. Each note can be followed by a '#','+', or '—'. The '#' and '+' characters indicate that the note is to be played as a sharp, and the '—' character indicates that the note is to be played as a flat note. However, the sharp or flat symbol is only accepted as being legal within the command string if it corresponds to a sharp or flat note on the piano keyboard. Thus the command below will play a series of notes.

```
PLAY "C#DD#E"
```

An octave in the Music Macro Language runs from the note C to the following C. To change octave, we use the O command. The letter O is followed by a number which gives the required octave. The Octave Number, as it is called, must be between 1 and 8, 1 being the lowest octave available and 8 being the highest octave. When we start using the computer, the octave in use is octave 4. Thus the command

```
PLAY "CDEFGABO5C"
```

will play a scale from the C of one octave to the C of the following one. Once an octave command has been issued, then all subsequent notes are played in that octave until a further octave command is issued. This is so even if the subsequent notes are played by totally different PLAY commands. Notes can be played on the 3 channels available quite easily; the notes are played simultaneously, thus giving the musicians amongst us the possibility for playing chords. For example, the command

```
PLAY "C","E","G"
```

will play the chord of C major, while the command

```
PLAY "C","D#","G"
```

will play the chord of C minor. Other chords can be played by putting suitable notes in the three strings. However, as this is not a text book on music, the interested reader is directed elsewhere for the theory and practice of playing chords.

It is possible to use numbers in the PLAY strings instead of note names. The letter N is used, followed by a number between Ø and 96. The C in octave 4 has a numerical value in this system of 36. The below command will play a string of notes defined by numbers.

```
PLAY "N1N2N3N4"
```

To me, this method has no real advantage over the use of note names.

Not all notes in a piece of music will have the same length; so, MSX BASIC equips us with a means of changing the length of time that notes are played for. The command L n within a PLAY string will set the length of subsequent notes to a value set by n until the next L command is issued. Subsequent notes have a length of 1/n times their normal length. The value of n ranges from 1 to 64, a value of 1 giving a full note played, 4 giving a quarter note and 64 giving a note 1/64th of the duration of a full note. An example of its use is given below.

```
PLAY "ABCL16DEF"
```

Again, like the Octave command, this will remain in force until the next L command is issued. Forgetting this can cause some problems in programs, so always keep it in mind. If the change in note length is only required for a couple of notes, then the notes that you want to shorten can simply be followed by the number without the L command. For example, in the command below, the note A will be played for 1/16th as long as the other notes.

```
PLAY "L1CDEA16DEF"
```

The value of n that is normally in use without us issuing any L commands is 4.

If we want to give a short pause between notes, then use the R command. The parameter of this command is again between 1 and 64, and a value of 1 following the R command will give a pause equal in length to a full note. Thus the command below will give a pause before playing the three notes in the string.

```
PLAY "R1ABC"
```

A value of 4 after the letter R would give a rest equal in length to a quarter note and a value of 64 would give a rest equal in length to 1/64 note.

We've already seen how we can shorten the time that a note plays for using the L command; it is also possible to extend the playing time of a note, using the '.'. A single dot following a note in a string will extend the playing time of that note by 1 & 1/2 times. The dot can also be used to extend the duration of a rest in a similar way. The effects of a dot are cumulative; thus the example below will cause a note to be played for 9/4 times its normal duration.

```
PLAY "A.."
```

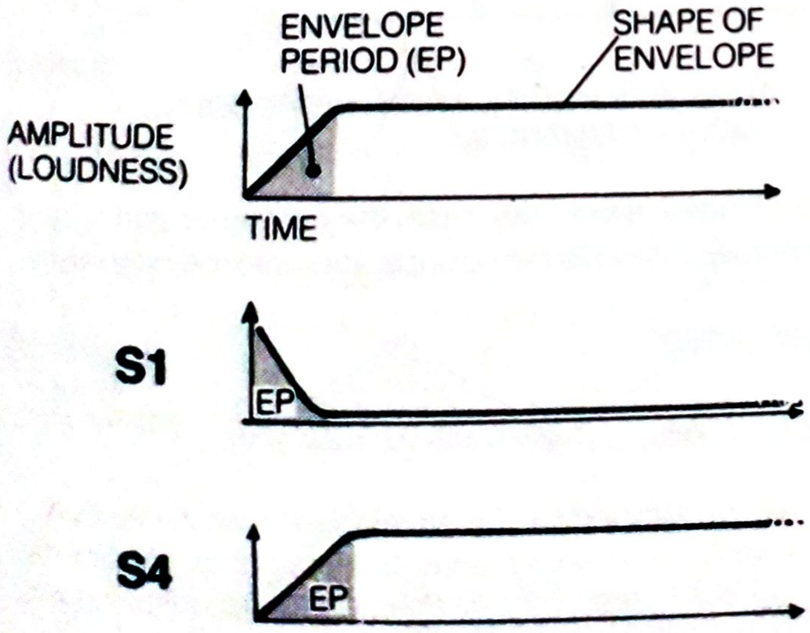
Where does 9/4 come from? Well, it's simply 3/2 * 3/2. It is clear that by using the . command repeatedly, notes with an extremely long duration could be played.

To set the speed at which our composition is played, we set the Tempo of the piece. This is done by the T n command, where n has a value of between 32 and 255. This number specifies the number of quarter notes to be played in one minute. The default value is 120.

The Volume of sound produced is set by the V command. The parameter n passed to the V command is between 0 and 15, a value of 0 being inaudible and 15 being loud. The default volume level is 8.

Envelopes

The sounds produced so far have been all of a volume that is set by the V command, and it has not been possible to vary the volume during the time that a single note is playing; i.e. the note is the same loudness throughout the time it is being played. If we compare this to a real musical instrument, like a piano, then we find that in the piano the note starts off at a loud volume and decays away slowly. The piano note is 'shaped', and this shape is called an envelope. We can simulate this to a small degree on the MSX computers, where there are 8 different note shapes that we can select. The command used to select the note shape in the Music Macro Language is called S n, and the parameter n has a value between 0 and 15, certain values of n producing the same envelope as each other. Figure 8.1 (below and overleaf) shows the various shapes of note that are available, and the value of n needed to get them.



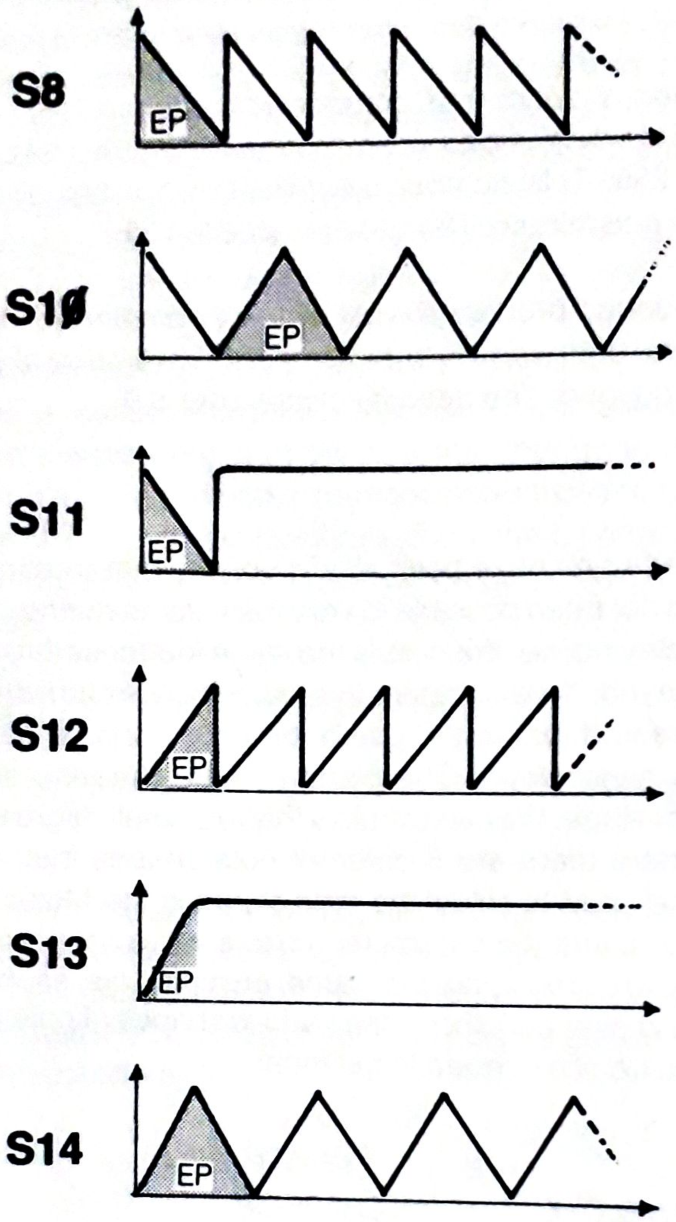


FIGURE 8.1 ENVELOPE MODIFICATION (1): THE SOUND MACROCOMMAND S

To see the effect of these envelopes, reset the computer and try the two commands listed below. Listen to the sounds and note the difference.

```
PLAY "S13A"

PLAY "S14A"
```

All subsequent notes will be shaped by the envelope shown in Figure 8.1 for shape 14, until another S command is issued. Try the other shapes, by putting in alternative values instead of 1.4. The command below shows that it is possible to change shape within a PLAY string. We can also play notes on all three channels using an envelope, but the envelope will be the same for each channel.

```
PLAY "S13ABCS14ABC"
```

Although the shape command alters the amplitude of the note being played, it may not alter the amplitude quickly enough for our purposes, or it might be too fast. We have at our disposal a command called M which changes the rate of change of amplitude given by a particular envelope. This is shown diagrammatically in Figure 8.2.

To hear this in action, try the following commands.

```
PLAY "S1M255A"

PLAY "S1M600A"
```

Note how the second command produces a high pitched 'ping' as opposed to the barely audible 'thump' produced by the first command. The value of n in the command can be between 0 and 65535. It determines the length of the area shown in Figure 14 as the 'EP', or Envelope

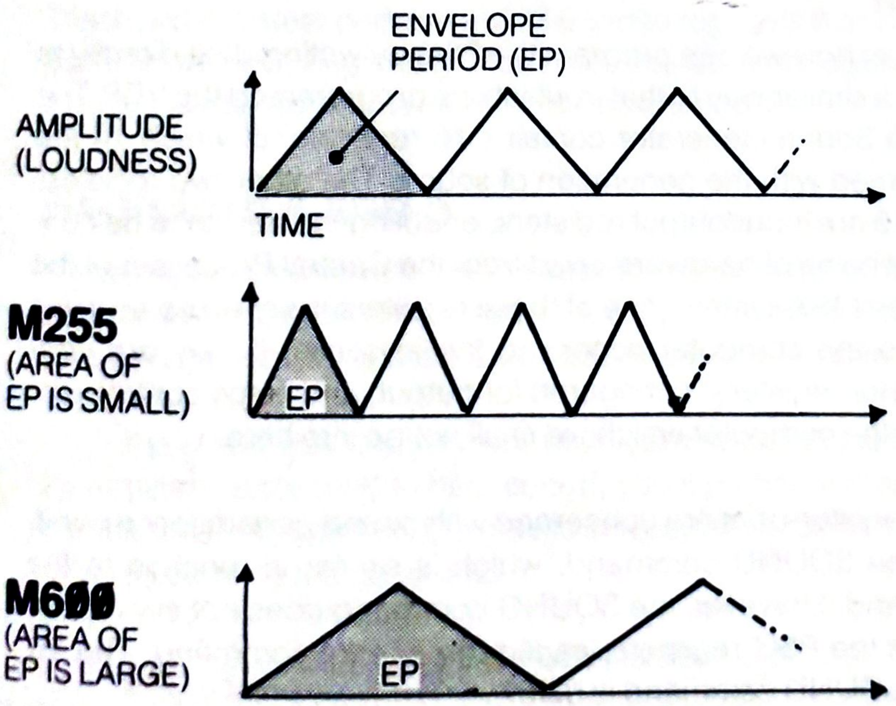


FIGURE 8.2 ENVELOPE MODIFICATION (2): THE SOUND MACROCOMMAND M

Period. Quite interesting effects can be obtained by varying the value of n passed over to the S and M commands. For example,

```
PLAY "L1S14M1000A"
```

gives an interestingly altered 'A', and the below command gives a passable impersonation of a trombone!

```
PLAY "S14M100AR14AR1R4AR14A"
```

Using the M and S commands in this way can lead us into the generation of sound effects as well as giving us a wider range of 'voices' with which to play tunes. While experimenting with the various commands in the Music Macro Language, should you wish to get things back to the way they were when you started, then typing CTRL-G will reset the PSG. Before leaving the Music Macro Language, variables can be passed into PLAY command strings in a similar way to which they were passed to DRAW command strings. Also, the letter X has the same effects in PLAY strings as it did in the DRAW command strings. The reader is directed to Chapter 6 for further details.

Direct Access of the Programmable Sound Generator

Let's now look at how we can program the PSG by writing data directly to its registers, in a similar way to that in which we programmed the VDP. The Programmable Sound Generator contains 16 registers, of which 14 are directly concerned with the generation of sound. The other two registers of the AY-3-8910 are input/output registers, enabling the device to be connected to other items of hardware apart from the Central Processor of the computer. In the MSX system, one of these registers is set up as an input register to give the computer access to the joysticks, if they are connected. The other register is configured for output, and helps control various aspects of the computer which we shall not go into here.

The 14 registers that are concerned with sound generation are written to using the SOUND command, which is similar in function to the VDP(n) command. However, the SOUND command does not allow us to read data from the PSG registers, as did the VDP(n) command. The full syntax for the SOUND command is below.

```
SOUND register, value
```

Register is a number between 0 and 13, and refers to the PSG register to which data is to be written. Value is the number to be written to the register, and is between 0 and 255. However, some registers will only accept numbers of lower value than this general rule; these will be pointed out when we consider them. We'll now go on to look at the function of each of the Programmable Sound Generator registers.

Register 0 and 1

The 8 bits of register 0 and the 4 lower bits of register 1 form a 12 bit register that controls the pitch of the note played on Channel 1. This register pair is obviously written to by a PLAY command that changes note frequency. The 4 bits from register 1 have higher significance than the other 8 bits, and a variation here has a larger effect on the pitch of the tone played than does a similar variation in the contents of register 0. For this reason, register 1 is often called the Coarse Tune Control Register and register 0 is called the Fine Tune Control Register. The actual tone generated on a given channel of the PSG depends upon the value held in the relevant pitch control registers, (registers 0 and 1 for channel 1) and the clock frequency being applied to the PSG. The higher the value placed in the Pitch control registers, the lower pitched the tone is that is generated.

Registers 2 and 3

These two registers perform a similar job to registers 0 and 1 respectively, but for channel 2. Register 2 is the Fine Tune Control Register, and register 3 is the Coarse Tune Control Register for channel 2.

Registers 4 and 5

These registers perform the same jobs as registers 0 and 1 but control the pitch of the note played on channel 3. Register 4 is the Fine Tune Control Register and register 5 is the Coarse Tune Control Register.

If you now go off to your computer, and write values to the appropriate registers, expecting to hear sound, you'll be rather disappointed. The information for the pitch of the note to be played will be written to the channel 1 registers by the command

```
SOUND 0,255:SOUND 1,0
```

but no data will have been sent to the computer regarding the volume of the note. To generate a pure tone, we need to send the pitch information and then send information to the PSG concerning the amplitude of the note that is to be played. Three registers within the PSG are used to control the amplitude of sound produced, one register controlling each of the three channels.

Register 8

This register controls the amplitude of sound played on channel 1. The register will hold a value between 0 and 15, 0 being the minimum volume and 15 being the maximum amplitude, or volume, of the note.

Register 9

As for register 8, but storing the data for the volume of sound on channel 2.

Register 10

This register performs the same job as register 8 but stores the volume data for channel 3.

Once a value is written to one of these amplitude control registers, then a note will be played on the appropriate channel. The program below shows this in action.

```
10 SOUND 0,255:REM fine tune of channel 1
20 SOUND 1,0:REM coarse tune of channel 1
30 SOUND 8,10:REM amplitude of channel 1
40 TIME=0
50 IF TIME<50 THEN GOTO 50
60 SOUND 8,0:REM after delay, turn note off
```

Note how we have to deliberately set the amplitude of the channel to 0 to turn the sound off. If we wanted to play tones on all three channels simultaneously, then we simply write the appropriate values to the relevant pitch control registers, and then finish the job by setting up the amplitude control registers, thus turning on the sound at the selected volume. The program below does this. Note the use of a DATA statement to give more efficient register setting up.

```
10 FOR I=1 TO 6
20 READ reg,v
30 SOUND reg,v
40 NEXT
50 DATA 0,255,1,0,2,200,3,0,4,150,5,0
```

Run the above, then issue the below line as a direct command.

```
SOUND 8,10:SOUND 9,10:SOUND 10,10
```

This will set the three amplitude registers up and turn on the sound. To turn off the sound, try CTRL-G or setting the 3 amplitude registers to zero. Once we've started a tone playing in this way, we can vary the tone's pitch by altering the Pitch Control registers for that channel. The program below shows this in action.

```
10 SOUND 1,3
20 SOUND 8,10
30 FOR I% = 0 TO 255
40 SOUND 0,I%
50 NEXT
60 SOUND 8,0
```

Line 10 sets up the Coarse Tune Control Register for channel 1, and line 20 sets the tone going. Lines 30 to 50 then continuously change the contents of the Fine Tune Control Register, resulting in a smooth variation in the pitch of the tone generated. Once the loop is finished, line 60 turns off the tone. As well as changing the frequency of the tone in this way, we can also alter the amplitude of the note while it is being played. The program below shows this in action.

```
10 SOUND 0,255:SOUND 1,0:REM set tone pitch
20 FOR I = 0 TO 15
30 FOR J = 0 TO 50:NEXT:REM delay loop
40 SOUND 8,I
50 NEXT
60 SOUND 8,0
```

This gives a tone of steadily increasing volume, which then is turned off altogether by line 60. So far, the way we've seen to stop a tone playing is to set its amplitude to 0. However, there is another method, which is useful if we just want to stop the note for a moment but then wish it to continue at the same volume. It uses register 7, known as the Enable Register.

Register 7

We will consider this register 1 bit at a time, as each bit controls a different aspect of the PSG. Bit numberings refer to the diagram below.

Bit 0. This is the Channel 1 Tone Enable. When set to 1, tone output from channel 1 is disabled totally, even if the amplitude set for channel 1 is not zero. Taking this bit to 0 will re-enable tone output on channel 1. As soon as this bit goes to zero, a tone will be played at the amplitude set by register 8, the channel 1 amplitude register. Obviously, if the contents of register 8 are 0, no tone is played.

Bit 1. Performs the same function as bit 0, but for channel 2.

Bit 2. Performs the same function as bit 0, but for channel 3.

Bit 3. When set to zero, this bit enables the playing of white noise on channel 1, at the amplitude held in the amplitude register for channel 1. When set to 1, white noise output on channel 1 is disabled. If both this bit and bit 0 of this register are set to zero, then both tone and noise will be played on this channel at the amplitude selected by register 8. More details about white noise will be given shortly.

Bit 4. Performs a similar job to bit 3, but for channel 2.

Bit 5. As bit 3, but acts on channel 3.

Bits 6 and 7. These two bits control the input/output registers of the PSG and so are not used for sound production. It appears from the MSX sys-

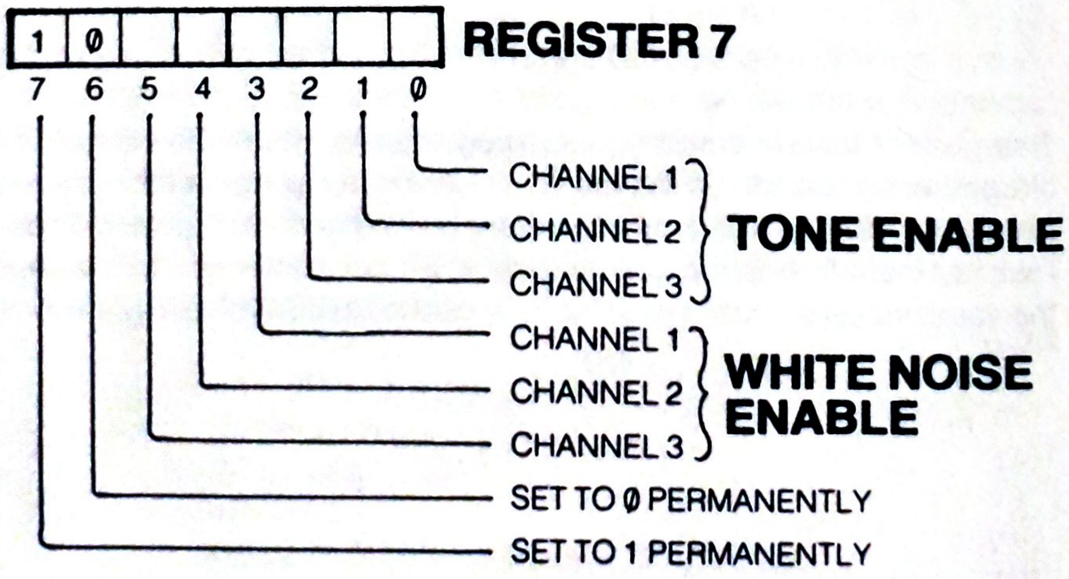


FIGURE 8.3 FUNCTIONS OF BITS IN REGISTER 7, PROGRAMMABLE SOUND GENERATOR

tem specification that bit 6 should be left at 0 and bit 7 be set to 1 to ensure correct operation of the system.

I find that the best way to write to this register is to represent the value to be written to the register as a binary number, as this makes it easier to select the pattern of 1's and 0's that are needed to set up the desired tone and noise outputs. For example, the command

```
SOUND 7,&B10111110
```

will enable tone on channel 1 and disable all other sound outputs. Bit 6 is set to zero for the reason outlined above. The command

```
SOUND 7,&B10110110
```

will enable both tone and noise on channel 1. Similar bit patterns written to the register will enable other channels in a similar way. For example, the command

```
SOUND 7,&B10111000
```

will enable tone on all three channels.

Noise

We've already mentioned noise, or White Noise, as it is often called, when discussing register 7 of the PSG. White Noise is best described in non technical terms as a rushing, hissing noise – try the program below to see it in action.

```
10 SOUND 7,&B10110111:REM enable noise on
channel 1
20 SOUND 8,15
30 TIME=0
40 IF TIME<50 THEN GOTO 40
50 SOUND 8,0
```

Many natural sounds are made up of noise; typical ones are rainfall, wind and waves. In games, we can use noise to simulate gunshots or explosions, etc. Noise is defined in terms of its volume, or amplitude, and its frequency. The amplitude of noise being played on a particular channel is controlled by the amplitude control register for that channel. So, to play a loud burst of noise on channel 1 we would use register 8 of the PSG to control the volume. Remember that to play noise on a particular channel, the relevant bit in register 7 of the PSG will have to be set to zero.

The frequency of the noise is a measure of the relative amounts of high and low frequency sounds in the noise. It is controlled by register 6 of the PSG, the Noise Pitch Control Register. The value placed in this register should be between 0 and 31; 31 gives the lowest pitched noise available and 0 gives a very hissy sounding noise. To hear the difference for the different values in register 6, try the below program.

```
10 SOUND 7,&B10110111
20 SOUND 8,6
30 FOR I=0 TO 31
40 SOUND 6,I
50 I$=INPUT$(1)
60 NEXT
70 SOUND 8,0
```

Run the program; pressing a key will cause the next value to be placed in the register, and so will modify the pitch of the noise being played. The routine below gives an example of a simple sound effect using the noise facility; a 'gunshot'. Line 20 sets the maximum volume for the noise on channel 1. Lines 30 to 50 vary the quality of the sound produced by varying the values placed in the noise pitch control register and the channel 1 amplitude register. Line 60 then turns the sound off.

```
10 SOUND 7,&B10110111
20 SOUND 8,15
30 FOR I=31 TO 0 STEP -0.5
40 SOUND 6,I:SOUND 8,I/2
50 NEXT
60 SOUND 8,0
```

A much easier way to produce sound effects is to use the facility that the PSG has for shaping the sounds produced. We saw this in action with the S command in the Music Macro Language.

Envelopes

Three registers are involved in the control of envelopes; these are registers 11,12 and 13. Let's look at these in turn to see what they do.

Register 13.

This is the Envelope Shape Control register, and can hold values between 0 and 15. The envelope shapes obtained for various values in the register are the same as those obtained for that value in the S command of the

Music Macro Language. Look at Figure 8.1 to see the various envelope shapes.

Registers 11 and 12.

These are the Envelope Period control registers and are the PSG registers written to by the M command in the Music Macro Language. Register 11 holds the lower 8 bits of the 16 bit parameter that would be passed over with the M command and Register 12 holds the upper 8 bits of this parameter. Thus the commands

```
SOUND 11,255:SOUND 12,255
```

and

```
PLAY "M65535"
```

perform the same job.

So, we can set the envelope required and the envelope period that we require, but how can we apply it to a particular tone being played? We do this by setting the amplitude register for the channel on which we want the shaped note played to hold a value of 16. Thus to get the note played on channel 1 to follow the envelope set up by registers 11, 12 and 13 we set register 8 to 16.

The program below plays a tone under envelope control on channel 1.

```
5 SOUND 7,&B10111110
20 SOUND 13,14:REM set envelope shape
30 SOUND 11,255:REM set up the envelope
40 SOUND 12,0:REM period
50 SOUND 0,255:REM set up tone
60 SOUND 1,0
70 SOUND 8,16:REM play tone
```

This sound will carry on until you set the contents of register 8 to zero, or until you type CTRL-G. The envelope thus defined can be used to shape the amplitude of noise on a channel as well as tone. This can be demonstrated by changing line 5 to the below.

```
SOUND 7,&B10110111
```

thus enabling channel 1 noise instead of tone. You could, of course, enable both at once if you want.

As you may have realised by now, pressing CTRL-G will reset the PSG registers to the original values that they held before you started reprogramming them. This can be a very useful thing to know if you've got into a mess by altering the registers!

That completes the overview of programming the PSG from BASIC. This device really needs much experimentation to get the best from it, and so I suggest that you go away and try some effects of your own. To start you off, try the following below.

The program below is a loader program reading values from a DATA statement and putting them in the appropriate PSG register. To change the sound produced, simply alter the DATA statement and run the program again. The first item in the DATA statement is the value for PSG register 0, the second item the data for PSG register 1 and so on.

```
10 FOR I%=0 TO 13
20 READ N
40 SOUND 1%,N
50 NEXT
60 DATA appropriate data items
```

Gunshot Plays on channel 1,2 and 3
DATA 0,0,0,0,0,0,17,7,16,16,16,1,5,1

Explosion Plays on channels 1,2 and 3
DATA 0,0,0,0,0,0,31,7,16,16,16,0,60,0

Boing Channel 3
DATA 0,0,0,0,0,0,9,&B10110111,16,0,0,191,5,1

Laser Beam Channel 3
DATA 128,1,0,0,0,0,1,54,16,0,0,251,10,15

Other sound effects can be easily synthesised by using FOR.. NEXT loops to modify the contents of PSG registers as we've already seen. For those of you with musical interests, it is possible to synthesise effects that sound like snare drums and other percussion instruments. However, the only way to truly master the PSG is through experiment. I wish you luck.

9 The Programmable Peripheral Interface

PPI stands for Programmable Peripheral Interface. It was discussed briefly in Chapter 1 in terms of its overall function; in this Chapter we shall look at it in greater detail. The first thing to note is that it is here that the standard for MSX falls down slightly – the PPI is written to by the use of commands called IN and OUT, which do not write to registers, but to particular addresses, and so we need to know the address in memory of the PPI before we can access it. The problem is posed by the possibility that some manufacturers may vary the position of the PPI in memory. Thus, the addresses given for the PPI in this Chapter will be those for the Sony HB-55 MSX computer. However, the chances are that they will work properly on your computer as well. The problem of incompatibility between machines will be minimized, however, by using routines that exist in the ROM to access the PPI. However, this is beyond the scope of this book.

The best place to start in this Chapter is with the BASIC commands that we use to access the PPI. These are OUT and INP.

The full syntax of the OUT command is

```
OUT port number, value
```

Port number is a value within the range 0 to 255, as is value. This is a good point at which to look at the concept of input and output ports on the MSX system. The Z-80 Central Processor Unit can, as already mentioned, access 65535 bytes of memory. But it can also access a further 256 locations called IN/OUT ADDRESSES. These are totally separate to the normal RAM or ROM, and in the MSX system provide the means by which the CPU communicates with the Video Display Processor and the Programmable Sound Generator. Some of these addresses are also used to allow the CPU to communicate with the Programmable Peripheral Interface. Whereas normal RAM or ROM locations were accessed by PEEK and POKE, the IN/OUT addresses are accessed by the INP and OUT commands. The full syntax of the INP command is

```
INP (port number)
```

where port number is again a value from 0 to 255. Thus to read data from IN/OUT address &HA8 and print it to the screen, we would issue this command.

```
PRINT INP (&HA8)
```

To send a value to IN/OUT address &HA8, we would issue the command

```
OUT &HA8,23
```

where 23 is the value being written to that address.

Anyway, on with the PPI. It contains 4 registers, 3 of which are input/output registers interfacing with things such as the keyboard, cassette interface and slot selection electronics (the latter will be discussed in Chapter 10). The fourth register is called the MODE SELECTION REGISTER, and controls whether the 3 in/out registers are configured for input, output or both. We shall look at this register first. The main thing we can say about it is – Don't Touch! If you alter the value in this register, you can disable the keyboard and occasionally switch out your memory! If you do want to play with this chip, then I suggest that you restrict yourself to reading or writing to the 3 in/out ports. However, if you are still interested, I would suggest you get hold of a Data Sheet for the 8255 PPI. This register is configured so that 2 of the other registers are for Output and 1 for Input.

The arrangement of the 3 registers in I/O address space is shown below. The addresses given are those stated in the standard MSX specification.

```
&HA8 Register A: OUTPUT
  A9 Register B: INPUT
  AA Register C: OUTPUT
  AB Mode Selection Register
```

PPI Addresses

Remember, however, that these may not be the correct addresses for your particular system, should the manufacturer have decided to alter things around a little.

Well, what do the registers do?

Register A. This is an output register, and is used to control memory allocation in the MSX system. More details will be given about this register in Chapter 1Ø. For the moment, it will suffice to say that it is not advisable to alter the contents of this register unless you are sure of what you are doing.

Register B. This is an input register, and returns, when read, a value relating to a keyboard press. This register only returns a sensible response when used in conjunction with the lower 4 bits of register C.

Register C. This is an output register, the main function of which is to help read the keyboard. Bits Ø to 3 of this register give what is called the KEYBOARD SCAN SIGNAL. The MSX system sets these bits to a particular pattern and then reads the value of register B. For each bit pattern output from register C, 8 keys can be detected. Each of these keys, when pressed, causes a bit of register B to be taken to Ø. This changes the value read back from register B and so enables us to read the keyboard directly. Figure 9.1, the MSX Keyboard map, shows how this functions.

For example, a value of &BØ1ØØ put out on the lower 4 bits of register C will cause the keys R,Q,P,O,N,M,L and K to be detected if any of them are being pressed at that moment in time. If key R is pressed, then bit 7 of register B is taken to Ø, thus returning a value of 127 if we were to read this register. As can be seen from the table, this method of reading the keyboard gives us a means of detecting keys such as SHIFT and TAB, which are not usually directly readable. The below routine shows the principles of reading the keyboard in this way.

```
10 OUT &HAA, &B00000111
20 PRINT INP(&HA9)
30 GOTO 10
```

Run the program, and press keys such as SELECT, ESC and BS. Obviously, this particular program will only work if the manufacturer of the machine has stuck to the original specification.

REGISTER B(1) Bit patterns read from Register B								REGISTER C(2) Bit patterns written to Register C
7	6	5	4	3	2	1	Ø	321Ø
7	6	5	4	3	2	1	Ø	ØØØØ
;	[	@	¥	^	—	9	8	ØØØ1
B	A	—	/	•	,	]	:	ØØ1Ø
J	I	H	G	F	E	D	C	ØØ11
R	Q	P	O	N	M	L	K	Ø1ØØ
Z	Y	X	W	V	U	T	S	Ø1Ø1
F3	F2	F1		CAPS	GRAPH	CTRL	SHIFT	Ø11Ø
RE-TURN	SELECT	BS	STOP	TAB	ESC	F5	F4	Ø111
→	↓	↑	←	DEL	INS	HOME CLS	SPACE	1ØØØ

(1) Value read from Register B by an IN from OA9H; all bits are set to 1 except for key being pressed, which is reset to Ø.
(2) The low nybble is written to Register C by an OUT to OAAH.

FIGURE 9.1 THE MSX KEYBOARD TABLE

Bit 4. This is the Cassette Control Signal, and it operates the cassette recorder remote control relay. When this bit is set to 0, the relay is closed, thus allowing the cassette to operate. When it is set to 1, the relay is open.

Bit 5. This bit is concerned with writing data to the cassette tape.

Bit 6. This bit controls the status of the CAPS LOCK light – that is, the little lamp that indicates whether or not the Caps Lock is engaged or not. When this bit is set to 0, the lamp is on, and when set to 1 the lamp is off. This bit has no influence on the status of the Caps Lock function itself, just the status of the lamp.

Bit 7. Setting this bit to 1 and then back to 0 will cause a click to be heard via the computer sound system. The program below shows this in action. You will, I think, agree with me when I say that the nature of the sound is not exactly musical!

```
10 OUT &HAA,&B00000000
20 OUT &HAA,&B10000000
30 GOTO 10
```

We've now covered all the functions of the PPI except that of register A. We'll now go on to look at the function of this register when we examine the memory organisation of the MSX system as a whole.

10 The MSX Memory Map

The memory map of a computer system is simply a description of how the computer's memory and input/output devices are arranged. Before we proceed any further, you're advised to consult the appendix on number systems, if you haven't already done so. We've already discussed the roles of RAM and ROM briefly, when we took an overview of the system in Chapter 1. In this chapter, we'll take a much greater detailed look at the arrangement of memory in the MSX system, as well as looking at the input/output devices that we can access.

The MSX ROM always starts at location Ø in memory and extends to location &H7FFF. The memory locations between &H8ØØØ and &HFFFF are available for RAM if desired. The memory map for the minimum MSX specification is shown in Figure 10.1. The minimum amount of RAM that the MSX system needs is 8k, but this gives a limited system. This is located from address &HFFFF downwards. Although the minimum system RAM is 8k, we can only add RAM in blocks of 16k. If we add RAM to the minimum system, then the 16k we add on would occupy the locations in the memory map from &HFFFF down to &HCØØØ, a total of 16384 locations.

This might come as a bit of a surprise if you were expecting to be able to add 16k of memory to an 8k system and get 24k as a result! As can be seen, this is not so; the 8k already present is overwritten by the 16k added, due to the fact that the MSX system handles its memory in 16k blocks, called PAGES. The addresses shown in Figure 10.1 are called PAGE BOUNDARIES. Thus an MSX system with 16k of RAM is said to have 1 page of RAM, extending from &HC000 to &HFFFF, and 2 pages of ROM occupying the space from &H0000 to &H7FFF.

So far, there is nothing peculiar about the arrangement of memory in the MSX system. What makes the MSX system different to other computer systems in terms of memory management is the concept of the slot, which effectively allows us to add more memory to the system, which may already have 32k of RAM and 32k of ROM. A system like this has a full memory map; the Z-80 CPU can only handle 64k of memory at one time. The slot is a block of 65535 (64k) locations that can be filled with RAM or ROM. All MSX computers must have at least 2 slots, but can have up to 4. One of these slots, the one containing the MSX BASIC ROM and the normal RAM for that machine, is called the SYSTEM SLOT, or slot 0. The second slot possessed by all MSX computers is called Slot 1, or the CARTRIDGE SLOT. This comes to the outside world as the cartridge socket on your machine, into which you can plug additional RAM or pre-written program cartridges. The way in which the computer uses the slots is quite involved, but, put in simple terms, the computer can, if needed, make use of pages from different slots to make up its memory map. Figure 10.2 shows an example of this in action.

Slot 0 contains the MSX ROM and 1 page of RAM, running from location &HC000 to &HFFFF. Slot 1 contains nothing, but Slot 2 contains 1 page of ROM, occupying the locations from &H4000 to &H7FFF, and 1 page of RAM occupying the locations from &H8000 to &HBFFF.

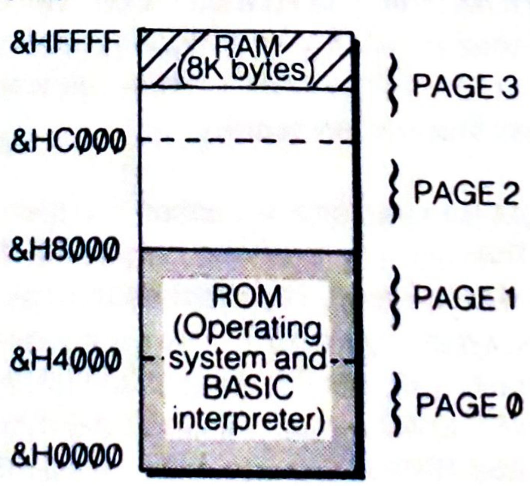


FIGURE 10.1 MEMORY MAP OF SLOT 0 (SYSTEM SLOT)

The computer then puts its memory map together using pages of memory from different slots, resulting in the RAM in slots 2 and Ø being treated as a continuous area of RAM, 32k bytes long. If the computer needs to use the Disc Control Software, then it simply gets its instructions from page 1 of slot 2 instead of page 1 of slot Ø. How does the MSX system gain access to pages of memory that are in different slots to the System slot? Well, this is where the Slot Select Register plays a vital part in the control of the computer. This register, as you will probably remember, is the A register in the PPI.

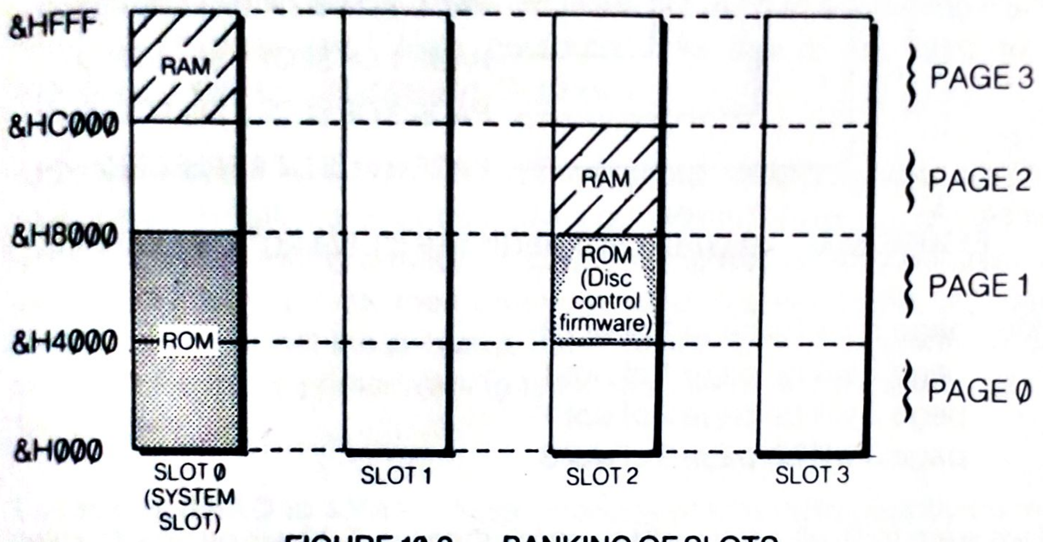


FIGURE 1Ø.2 BANKING OF SLOTS

Slot Select Register

This register informs the computer of which slot is to be used to get a particular page of memory. Bits Ø and 1 hold the slot to be accessed to get page Ø of memory, bits 2 and 3 hold the slot number for page 1, bits 4 and 5 hold the slot number for page 2 and bits 6 and 7 hold the slot number for page 3. The slot numbers are held as two bit binary numbers, ØØ representing slot Ø, Ø1 representing slot 1 and so on.

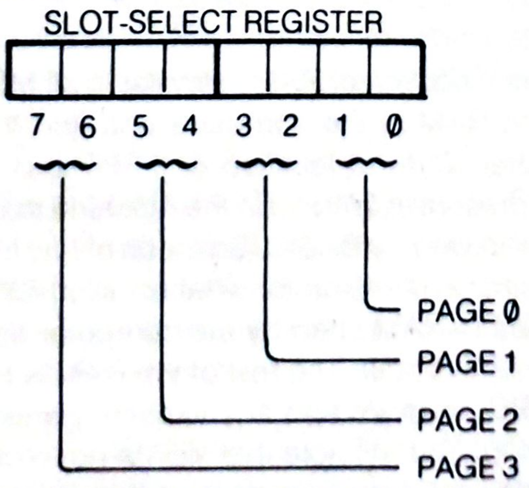
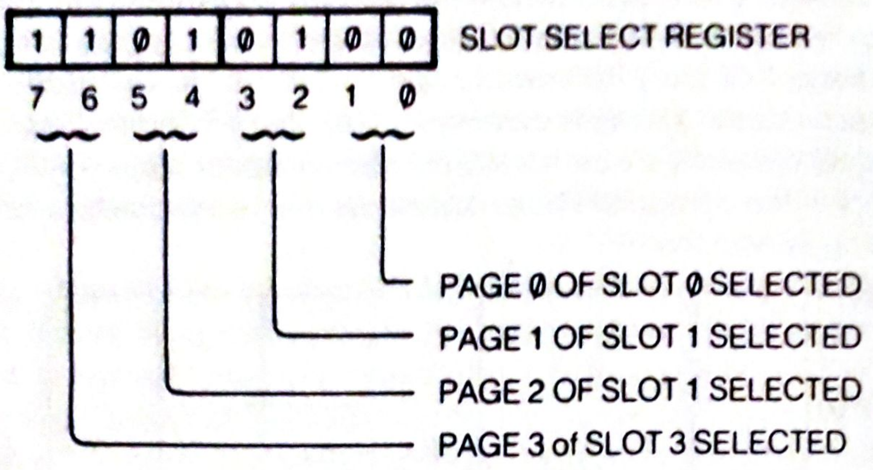


FIGURE 1Ø.3 SLOT-SELECT REGISTER (1): FUNCTIONS OF BITS

Let's look at an example.



1 1 Ø 1 Ø 1 Ø Ø
SLOT SELECT REGISTER
7 6 5 4 3 2 1 Ø
PAGE Ø OF SLOT Ø SELECTED
PAGE 1 OF SLOT 1 SELECTED
PAGE 2 OF SLOT 1 SELECTED
PAGE 3 of SLOT 3 SELECTED

FIGURE 10.4 SLOT-SELECT REGISTER (2): SELECTION EXAMPLE

Here, page Ø will be page Ø of slot Ø
page 1 will be page 1 of slot 1
page 2 will be page 2 of slot 1
page 3 will be page 3 of slot 3

If we were to fill all 4 slots with memory, then the Z-8Ø will be able to select its 64k of memory from 256k of memory that it can access through the slot system. However, this is not all – each slot can have associated with it 4 more slots, each of these slots also containing 4 pages of memory. These slots, arising from our original ones, are called SECONDARY SLOTS. The 4 slots that can be selected directly by accessing the Slot Select Register are called PRIMARY SLOTS. The control of the secondary slot structure is somewhat beyond the scope of this book, and mention of it is made here so that the reader is aware of the potential of the slot system for future expansion.

The exact allocation of RAM is similar in all MSX computers; BASIC will examine the RAM in the computer and find the largest continuous area of RAM that starts at location &HFFFF and works downwards in memory. RAM present in any slot in the machine may hence become part of the RAM that is used by BASIC. So much of this memory, at the top end of RAM, is placed on one side for what is called SYSTEM WORKSPACE, which is the area of RAM used by the computer as 'scrap paper'. More will be said about this later. The rest of the RAM is made available for the text of the BASIC program and any variable generated by the program when it is running. You will note that VRAM never appears as part of the slot structure, as it can only be accessed directly by the VDP.

We'll now look at the IN/OUT map of the MSX computer – that is, how the 256 locations that can be accessed by the OUT and INP commands discussed in Chapter 9 are arranged and allocated to various devices. The I/O locations that are listed below are those given in the MSX specification; they were all correct for my particular machine, but they may differ if some manufacturers decide to change the map around at all.

Locations &H00 to &H80. Not used in the current MSX specification.

Locations &H80 to &H88. These locations are used to control the RS232 interface fitted to some MSX computers. As this is not fitted to all machines, I will not comment on it further.

Locations &H90 to &H91. Control the printer interface, and are thus used by the BASIC commands LLIST and LPRINT. They are of little use unless you plan writing a machine code routine that directly controls the printer via this port. If so, then when you read location &H90, bit 1 indicates whether or not the printer is busy. Writing to this location sends a strobe pulse to the printer. Writing to location &H91 will write the data to the printer.

Locations &HAO to &HA2. These locations enable us to directly control the PSG through INP and OUT commands. The true value of these locations is seen when we are writing machine code programs, where the PSG can be controlled by the Z-80 IN and OUT instructions. However, it is possible to use these locations in BASIC, as we shall now see. The principles shown here will be of equal value, however, when you come to write to the PSG in machine code programs.

Location &HA0 is called the ADDRESS LATCH of the PSG. To write a value to a particular PSG register, we use an OUT command to place the register number in this location. For example, to write a value to the channel 1 amplitude control register, register 8, we would first of all write the value 8 to this location.

Location &HA1 is the DATA WRITE register for the PSG, and is used when we want to write data to a PSG register. After setting the register number up in location &HA0, we write the data that we want to put in that register to this location. Thus to write the value 15 to PSG register 8, we could use the commands below.

```
10 OUT &HA0,8
20 OUT &HA1,15
```

Location &HA2 is called the PSG DATA READ register, and is used to read data back from the PSG registers. This is a facility that is missing from the BASIC sound commands. To read, for example, the current contents of PSG register 8, the lines below of text are used. This prints the contents of the register specified in the PSG Address Latch.

```
10 OUT &HA0,8
20 PRINT INP(&HA2)
```

This is obviously quite a useful feature, enabling us to find out at any time the contents of PSG registers. It also enables us to directly access the two input/output registers in the PSG that are not normally accessible.

Locations &H98 to &H99. These two locations enable the programmer to directly access the VDP without using the VDP command. Again, it is of most use to the machine code programmer. They also enable us to directly access VRAM locations without using VPOKE or VPEEK.

Again, the exact locations that these two addresses are placed at may vary from machine to machine. However, in this particular example, the locations in the IO map that correspond to the read and write locations of the VDP are stored in locations 6 and 7 of the MSX ROM respectively.

To read from the VDP status register, we carry out an input command from location &H99. Thus the command

```
PRINT INP(&H99)
```

from BASIC will read the VDP status register. You are directed to Chapter 6 to see what effect reading the register has on its contents. It is very important to always read this register before attempting to write data to the VDP registers. This is to inform the VDP to prepare for a write operation.

Writing to a VDP register is quite straight forward. There are 3 operations involved, and these are demonstrated in the routine shown below, which gives the general method for writing data to a VDP register.

```
100 dummy=INP(&H99):REM dummy read operation
110 OUT &H99,value:REM byte to be written
120 OUT &H99,(register number + 128)
```

Value is the byte of data that is to be written to the VDP register, and so should be between 0 and 255. The second write operation sends the modified register number to the VDP. As a concrete example, the few lines below write a value of 18 to VDP register 7.

```
100 dummy = INP(&H99)
110 OUT &H99,18
120 OUT &H99,135
```

What about accessing the Video RAM via these locations? This is slightly more involved than accessing the VDP registers, but is still quite easy.

Let's start by writing a value to VRAM. Again, it's good practice to read the location &H99 before initiating a VRAM write operation. Location &H99 is then written to twice to send the address in VRAM to which we wish to write to the VDP. We then send the data byte that is to be written to that location to IO address &H98. The effect that writing a particular value to a particular VRAM address will have in any mode is outlined in Chapter 6. The lines below outline the basic technique for writing a byte to VRAM.

```
100 dummy = INP(&H99)
110 OUT &H99, VRAM address MOD 256
120 OUT &H99,(VRAM address ¥ 256) + 64
130 OUT &H98,value
```

Note the manipulation of the VRAM address that is needed. Remember that the ¥ sign stands for integer division. As a real example, the routine below will write a value of 65 to VRAM address 0. The best mode to demonstrate this in BASIC is Mode 0, as VRAM address 0 in this mode corresponds to the first location in the Name Table, and so will show on the screen.

```
100 dummy = INP(&H99)
110 OUT &H99,0
120 OUT &H99,64
130 OUT &H98,65
```

It is very rarely that we will want to write to just 1 VRAM location; the VDP takes this into account in that once a VRAM address has been set up in this way and a byte written to it, the next byte that is written to location &H98 will be written to the next location in VRAM. Thus in the example that we've just seen, the next OUT &H98 command, assuming that no other VDP accesses had been issued, would write a value to location 1 of VRAM, and so on. This feature, which is called AUTO INCREMENT, makes it possible for the machine code programmer to write sequences of bytes to VRAM without having to restate the address each time.

Obviously, this feature is only of use if the bytes to be written to VRAM follow each other in memory.

To read a value from VRAM, a similar approach is used. We simply replace the OUT operation that writes to location &H98 in line 130 with an INP instruction to read that location. Thus the routine below will read the value of the byte currently held at VRAM location 0.

```
100 dummy = INP(&H99)
110 OUT &H99,0
120 OUT &H99,0
130 PRINT INP(&H98)
```

Locations &HA8 to &HAB. These locations are mapped on to the PPI registers, so see Chapter 9 for details.

Locations &HB0 to &HB3. Used in some machines to control additional memory.

Locations &HB8 to &HBB. Used in some machines for light pen control.

That completes this study of the areas of the IO map that are of use to the MSX programmer; you are advised to remember that although these addresses are the ones set down in the MSX system specification, some manufacturers may choose to ignore these guidelines. However, the access that a knowledge of these addresses gives the machine code programmer to the various system components is quite useful, and should not be ignored by anyone programming the MSX computers.

RAM Arrangement

Most of the RAM in the computer is given to BASIC for the storage of BASIC program and variables. We looked briefly in Chapter 4 at how the basic program is written in RAM. We will look at this again now, in a little more detail. We will begin by finding out where the BASIC program is stored in memory. The easiest way to do this is to evaluate the expression below;

```
PRINT 65536-(n*1024)
```

where n is the number of k of memory possessed by your machine. For example, this will return a value of 49152 for a 16k MSX computer. In all subsequent examples, I will regard the start of the BASIC program as being at address 49152.

Let's look at how a typical, small program is stored in memory.

The program is:

```
10 REM test
20 PRINT 'Hello'
30 END
```

If we use a FOR. . .NEXT loop to PEEK out the values held in RAM from 49152 onwards, then we get the below.

49152	0	
49153	12	start of line 10: this is
49154	192	a two byte pointer to the start of line 20
49155	10	low byte of line number
49156	0	high byte of line number
49157	143	token for word REM
49158-49162		text of REM statement
49163	0	end of line 10
49164	26	start of line 20:as
49165	192	for line 10
49166-49177		rest of line 20
49178	32	start of line 30:as
49179	192	for line 10
49184	0	address pointed to by the
49185	0	pointer at start of line 30

The two numbers at address 49153 and 49154 form a single, 16 bit number. The address pointed to can be calculated by

```
PRINT 12 + 256*192
```

and this is found to point to the start of line 20. The next two bytes form the line number, again held as a 16 bit number with the low, or least significant, byte held first. The line number can be calculated by a process similar to that above. There then follows the text of the line, followed by a zero to mark the line end. This general pattern is then repeated for each line of the program until we get to the last one. The address pointed to in this case, which is held in locations 49178 and 49179, is 0. This pair of zeros at address 49184 and 49185 indicate to the computer that that is the last line of the BASIC text.

Poking locations 49153 and 49154 with the value zero results in the computer 'losing' the program; this is not, however, the same as the effect given by the NEW command – the NEW command resets the memory pointers and FRE(0) returns the value displayed on turn on of the computer. After the two POKEs, FRE(0) returns a value that indicates that the program is still present. Poking 49152 with any other value than 0 results in the computer being unable to RUN the program, even though it still lists after a LIST command.

Variable Storage

After the BASIC program is stored, it can be executed and variables are generated. These are stored in the RAM following the program, or, in the case of certain string variables, in the String Space allocated by the computer for this use. The below observations were made by PEEKing out the area of RAM that follows the program, and so I accept full responsibility for any errors in interpretation that may have arisen here.

String Variables

When the variable was assigned explicitly, eg,

a$ = "abcdefg"

then the area of memory that holds the variable is as below. It is this area of memory that is pointed to if you use the VARPTR command.

3	indicates variable type
n$	first letter of variable name
n1$	second letter of name
L	length of string
low	address of last point in program
high	text where an assignment to this variable was made. It points to the first letter of the string constant assigned to the variable.

The latter address can be evaluated in a similar fashion to the pointer addresses in the BASIC program text. As an example of its use, in the example above it would point to the position in the program of the letter 'a' at the beginning of the string 'abcdefg'.

If the assignment within the program is of the form

a$ = b$

then the pointer here will point to the first letter in the string that was assigned to b$, if b$ was assigned a value from a string constant in the program. It is worth noting at this point that when we type in a line such as

a$ = 'MSX'

into a program line, an extra byte is inserted into the line to indicate that an assignment is being made.

If the assignment is made using the CHR$ command, for example,

a$ = CHR$(65)

then extra bytes are again inserted into the program line indicating that an assignment is being made. Also, the bytes of the pointer now point to the area of memory in the String Space where the letters making the string up are stored. Thus, in the example above the pointer would point to the memory location holding the letter 'A' in String Space. The value returned by VARPTR(a$) in the above examples is the address of the string length in the variable table.

String Arrays

The block of memory that holds data about the string array contents is shown below. I am not sure of the significance of the 4th and 5th bytes of the table, but they could be indicators to the computer of the fact that this block of memory represents a string array. One point to note here, that is also applicable to normal string variables, is that if an assignment is made while in Direct Mode, then the pointer for the variable concerned points into String Space.

3	identifier
n$	first character of name
n1$	second character of name
?	significance unknown
?	significance unknown
dim	number of dimensions
elo	low byte for the total number of elements
ehi	high byte for number of elements

len0	length of first element in array
add0	low byte of address of first element contents
add0hi	high byte of address of first element contents
	the last three entries are then repeated for every element of the array.

When we say first element of the array, we mean element 0. So the first element of array a$ is a$(0). If the length and address entries for a particular array element are all set to zero, then it indicates that the element in question has not yet been assigned a value.

Integer Variables

These are the simplest numeric variables, and can be useful when we wish to pass values to and from machine code routines. They are arranged in memory as shown below.

2	identifier
n$	first character of name
n1$	second character of name
val low	low byte of numeric value
val high	high byte of numeric value

The value held by the integer variable can be easily evaluated by the method below:

```
PRINT PEEK (val low) + 256*PEEK(val high)
```

The other numeric variables are not so easy to use or understand, and I shall not go into great detail here about them.

Real Variables

There are two types of Real Variable, the Single Precision and the Double Precision variables. They are very similar in the way in which they are stored by the computer.

8	identifier
n$	first character of name
n1$	second character of name
exp	exponent (see below)

mantissa	A 4 byte mantissa for the Single Precision variables, the least significant byte being the last byte in the table.
	A 7 byte mantissa for Double Precision variables, again with the least significant byte of the mantissa being the last byte in the table.

The exponent of the number is stored in coded form, as the exponent+65. Thus an exponent of 1 is stored as 66. If the exponent is negative, then a further addition is made, negative exponents being represented by adding 128 to the value that would represent the positive exponent of the same magnitude.

Integer Arrays

These are similar in structure to the string arrays, as shown below.

2	identifier
n$	first character of name
n1$	second character of name
?	significance unknown
?	significance unknown
dims	number of dimensions
elo	low byte of number of elements
ehi	high byte of number of elements
ele0 low	low byte of element 0
ele0 high	high byte of elements 0
	repeated for rest of array elements, at two bytes per integer entry.

Real Arrays

These are arranged in a similar fashion to the Integer Arrays, but with an identifier of 8 and either Single or Double Precision numbers as the array elements. The structure of the array elements is the same as that for ordinary Real Numbers.

System Workspace

The MSX computer is a very complicated computer, and needs to utilise some RAM for housekeeping purposes. The RAM that is used by the MSX

ROM is called System Workspace, and this is why on entering BASIC on your machine you never have as much memory for your BASIC programs as you think you should. Before allowing you to type in your programs, the computer ensures that it has enough space to perform its own tasks before trying to run your programs.

In this section I will make a brief mention of a couple of areas of memory that may be useful to you in your MSX programming. It is obviously not a comprehensive view of the system, but will give a few pointers to what the System Workspace takes care of. If you wish to look at this area, then a simple program can be written that will enable you to display blocks of, say, 20 bytes on the screen at once, so that you can see if any of the bytes alter their value with time, for example, the program below is a simple means of examining any block of memory in this way. Press any key after the first 20 bytes have been displayed to see them again. Any bytes that are changing in value will thus become apparent. To type in a new start value, simply rerun the program.

```
10 INPUT start
20 FOR I = start TO start + 20
30 PRINT I; "  ";:J=PEEK(I)
40  PRINT J;
50  IF J>31 AND J<127 THEN PRINT CHR$(J) ELSE
PRINT
55 NEXT I
60 I$ = INPUT$(1):CLS:GOTO 20
```

The System Workspace consists of all the addresses in RAM above address 62336, and so you should be very careful in POKEing values into this region of memory.

Soft Key Definitions

The strings held in the function keys are stored in the System Workspace from address 63615 to address 63774. Thus it is possible to alter the strings held in the keys by directly accessing the memory. It is also possible to store key definitions to tape using BSAVE, once these addresses are known.

Input Buffer

The area of memory from location 64496 to 64576 appears to be used by the computer to keep a record of incoming key presses. This area of memory is called the Input Buffer.

Start of RAM

The start of RAM to be used by BASIC appears to be held in locations 64584 and 64585. Thus the start of available RAM can be calculated using

```
PRINT PEEK(64584) + 256*PEEK(64585)
```

Start of System Workspace

This appears to be held in locations 64586 and 64587. The start of the Workspace can be calculated in a similar fashion to the start of RAM.

TIME counter

64670 and 64671 hold the variable TIME, which is regularly incremented. Thus setting these two locations to zero will have the same effect as setting TIME to 0 via a statement such as

```
TIME = 0
```

This is obviously a useful ability from machine code, enabling us to set TIME from within machine code programs.

These are the areas of the System Workspace that are of most use to the BASIC programmer. In the next Chapter we'll examine a couple of sample BASIC routines which may be of interest, and then go on to discuss BASIC programming practice.

11 BASIC Style and Sample Routines

The role of this Chapter is twofold; firstly to introduce some simple but useful routines for you to use in your programs, and secondly to outline the techniques involved in writing BASIC programs that are as readable and error free as possible. This may not seem terribly important to you at the moment, but should you ever be in the position of returning to a program written some months previously that has suddenly developed problems, then you will come to appreciate the readability of your programs.

However, we'll start by looking at a couple of sample routines that I have found useful. The first of these overcomes an annoying feature of the MSX machines – their return to a text mode from a graphics mode whenever an INPUT instruction is executed. If there are graphics on the screen, this can be extremely annoying! The routine is based upon the techniques that we first looked at in Chapter 6.

General Purpose Input Routine

This subroutine requires a few variables to be set up before it is called. These are as follows.

BAC	background colour required
FC	foreground colour required
X,Y	position on graphics screen at which you wish the input to occur
L	Length of string to be accepted

The last parameter is the length of the longest string that the routine will accept as input. This version only accepts upper case letters being typed in from the keyboard, but it can be modified to accept other characters by modifying line 1030 of the subroutine. Also, you cannot type return to leave the routine; some characters have to be typed in first. If you make a mistake while typing in the text then use BS to delete the wrong letters. On leaving the routine, A$ holds the string that has been typed in. As an example of a modification, you could alter the subroutine to accept numbers by altering line 1030 to read;

```
1030 IF (G<47 OR G>57) AND G<>13 AND
     G<>8  THEN GOTO 1020
```

The string will now be made up of numbers containing the digits 0 to 9. To convert this into a numeric value, simply use the VAL command.

```
1000 REM input subroutine
1010 OPEN "GRP:" AS # 1
1015 A$ = " "
1020 G$=INPUT$(1):FL=0
1025 G=ASC(G$)
1030 IF (G<65 OR G>96) AND G<>32 AND
       G<>13 AND G<>8 THEN GOTO 1020
1035 IF G = 8 AND A$ = " " THEN GOTO 1015
1045 IF G = 8 THEN FL = 1
1050 IF G = 13 AND A$ = " " THEN GOTO 1050
1060 IF G=13 THEN CLOSE#1:RETURN
1070 IF FL=0 THEN A$=A$+G$
1080 IF FL=1 THEN
       C$=MID$(A$,1,LEN(A$)-1):PRESET(X,Y):
       COLOR BAC:PRINT#1,A$:COLOR FC:
       PRESET(X,Y):PRINT#1,C$;:A$=C$
1090 A$=LEFT$(A$,L):PRESET(X,Y):
       PRINT#1,A$:GOTO 1020
```

This routine can, of course, be modified for use in the text modes, where it still offers many advantages over the usual INPUT statements, as it does away with the possibility of users typing in numbers when strings are expected and vice versa. The program below will provide a demonstration of the routine above. It assumes that you've typed the routine in at line 1000, as above. If you do want to put the routine elsewhere in memory, then simply alter the line numbers in the GOTO statements.

```
10 BAC = 4:FC = 15:L = 5:X = 100:Y = 100
20 SCREEN 3
30 GOSUB 1000
40 PRINT A$
50 END
```

Detecting specific keys

The routine below offers the programmer a general purpose method of checking for the user pressing a specific key. The RETURN and SPACE keys are often used to control the flow of a program; this routine will enable the programmer to have one routine to check for all legal keys. On entry to this routine, the variable TEST$ should hold all the letters that are legal at that point in the program. Upper and lower case letters should be included in the string. If characters such as that produced by the RETURN key are to be tested for then they can be put into TEST$ using the CHR$() command. eg,

```
TEST$ = "AaBbCc" + CHR$(13)
```

On return from the routine, the variable PS holds the position, within TEST$, of the key that was pressed. The routine is not returned from until a legal key is pressed.

```
1000 REM key test routine
1010 G$ = INPUT$(1)
1020 IF INSTR(TEST$,G$) = 0 THEN GOTO 1010
1030 PS = INSTR(TEST$,G$)
1040 RETURN
```

As a demonstration of its use, try the program below, which will test for the letters Y or N, in upper or lower case.

```
10 TEST$ = "YyNn"
20 GOSUB 1000
30 IF PS>2 THEN PRINT 'No' ELSE PRINT 'Yes'
40 GOTO 20
```

Saving Display Screens

The following routines will enable you to save various sections of the Video RAM to tape, and then to reload them back into VRAM. Obviously, if you save the appropriate areas, it is possible to save the image currently on the screen. This is of particular use if the screen image has taken a long time to draw.

Two methods are outlined here; one, the one using PRINT#, will work on all MSX micros and will save all screen modes completely. However, it is rather slow. The second, which uses BSAVE, requires an area of memory equal in size to the largest area of VRAM to be saved to be set up as a buffer in normal memory. Although it is faster, this method can cause problems in the graphics modes on MSX computers with only 16k of memory. However, if you don't want to save the whole of VRAM, but just part of it, then either method can be used.

As an example, we will save the Mode 0 screen to tape, using these two methods in turn. Note that if you want to speed up either technique, you can switch the Tape Speed to 2400 baud for the write operation. See Chapter 4 for details.

The first routine, using PRINT#, takes 1 minute 20 seconds to write the Mode 0 Name Table and Pattern Table to tape.

```
1000 REM save screen
1010 NM%=BASE(0)
1020 PM%=BASE(2)
1030 OPEN "SCREEN" FOR OUTPUT AS#1
1040 FOR I%=NM% TO NM%+959
1050 C%=VPEEK(I%):PRINT#1,CHR$(C%);
1060 NEXT
1070 FOR I%=PM% TO PM%+2047
1080 C%=VPEEK(I%):PRINT#1,CHR$(C%);
1090 NEXT:CLOSE#1
1100 RETURN
```

This routine, when called, will save the areas of the VRAM that serve to define the Mode 0 screen. For it to be useful, therefore, it is a good idea to call it from the program while Mode 0 is in use and the screen to be saved is displayed!

As to reloading it back in, the routine below will perform this task.

```
2000 REM loading screen for Mode 0
2010 NM%=BASE(0):PM%=BASE(2)
2020 OPEN "SCREEN"  FOR INPUT AS#1
2030 FOR I%=NM% TO NM%+959
2040 A$=INPUT$(1,#1):VPOKE I%,ASC(A$)
2050 NEXT
2060 FOR I%=PM% TO PM%+2047
2070 A$=INPUT$(1,#1):VPOKE I%,ASC(A$)
2080 NEXT
2090 CLOSE #1:RETURN
```

When you want to call this routine, first put the computer into screen Mode 0.

The second method, as already explained, transfers the contents of the VRAM of interest to a block of normal RAM, and then uses the BSAVE command to save the RAM. This method is faster, as less writing to tape is involved, but it has higher requirements in terms of memory, especially if you are saving Mode 2 screens.

The program below demonstrates the principles involved. There is nothing to stop you writing this as a subroutine, but if you do you are advised to execute the CLEAR command in line 10 at the beginning of the program.

```
 10 CLEAR 200,54999
 20 J=55000
 30 FOR I%=BASE(0) TO BASE(0)+959
 40 POKE J,VPEEK(I%)
 50 J=J+1:NEXT
 60 J=56000
 70 FOR I%=BASE(2) TO BASE(2)+2047
 80 POKE J, VPEEK(I%)
 90 J=J+1:NEXT
100 BSAVE "SCREEN" ,55000,59000
110 END
```

To reload this screen, we use BLOAD instead of INPUT$. Again use the reload routine in Mode 0.

```
10 CLEAR 200,54999
20 SCREEN 0:BLOAD "SCREEN"
30 J = 55000:FOR I% = BASE(0) TO
   BASE(0) + 959:VPOKE I%,PEEK(J)
40 NEXT
50 J = 56000:FOR I% = BASE(2) TO BASE(2) + 2047
60 VPOKE I%,PEEK(J)
70 NEXT
80 END
```

If you use these routines for transferring other areas of VRAM, for example, the Mode 2 pattern Table, then ensure that you make a large enough buffer in normal RAM, as if you don't, you could accidentally overwrite part of the System Workspace. Any part of VRAM can be transferred to tape in this fashion. One particular use is the saving of Mode 0 and Mode 1 character sets to tape by saving the appropriate Pattern Table. Or, we could save the Sprite Pattern Table to tape, thus saving the Sprite definitions for a particular program.

Programming Style

Anyone can write a small program with very little planning; however, as soon as you begin to write large BASIC programs you can encounter problems due to lack of planning of the program. In this section of the Chapter, I hope to outline a few techniques that may help you write programs that are error free, efficient and easy to alter and amend.

Let's begin with program planning. Before turning the computer on, sit down and analyse the problem into its constituent parts. For example, let's suppose we're writing a program to draw a picture of a house. This problem can be broken down into 4 smaller parts, as below.

i Draw the Walls
ii Draw Roof
iii Draw Windows
iv Draw Door

This analysis of a single, fairly large program into 4 smaller ones, in this case, is known as TOP DOWN analysis. Each of these small problems can now be dealt with in turn, and we can write a subroutine to do each job. If the BASIC instructions needed to perform this job look like being quite long, then we can split the task again into smaller portions, and again subroutines written to perform these smaller jobs.

This particular method of program design has several advantages over simply allowing a program to develop at the keyboard. The first is that each subroutine developed to solve a particular part of the problem specified can be tested by itself, as we've seen in the demonstration program for the input routine. The subroutine can thus be debugged before it takes its place in the program, and any problems that arise after the routine has been placed in the program will thus be caused by something outside the program section that has been placed in the program.

A second point to remember is that if programs are developed around subroutines performing particular parts of the program, then if you don't like the way that that part of the program behaves, you can simply alter the appropriate subroutine.

Thirdly, after you have written a few programs, you will begin to accumulate a collection of subroutines for performing certain common tasks, such as input, or key testing routines, such as those listed at the start of this Chapter.

When you write your subroutines, start them with a REM statement that informs you of what the subroutine does, also possibly listing the variables that the routine needs to work correctly. It is also useful to note in these REM statements what variables return the results when the program control is passed back to the main body of the program from the subroutine.

Variable Names

We've already noted the 'rules' for naming variables in Chapter 3. However, it is quite important when writing programs that may need to be altered in the future to use meaningful variable names wherever possible. For example, if we are using a string variable to hold the name of a file that is to be written to tape, we could call the variable a$. But it would be much more informative if the variable was called name$, or file$.

There are problems with this approach; remember that only the first two letters of a variable name are relevant. If you don't, then you can come up against some very confusing program bugs.

GOTO's

The GOTO command is quite useful, but overuse of it can lead to an incredibly complicated mess! If you are trying to read a computer program that has GOTO instructions causing program control to pass around the program every 5th or 6th line, then you will soon get tired.

Always try to restrict the use of GOTO instructions in your programs. They are necessary; there is no way in MSX BASIC that you can totally eliminate the use of GOTO's. But you can make programs that use GOTO instructions more readable. I try and keep the destination line of a GOTO command within 6 or 7 lines of the command; this is quite easy if we've already split the program into subroutines via a Top Down programming approach.

Other things that help make the program more readable include putting the loop control variable after a NEXT command, especially if the FOR is several lines distant, or if nested FOR...NEXT loops are in use.

Areas of the Program

When I write a program, it can be split into several distinct areas of code. The first few lines I call the BODY of the program; this often consists of little more than a series of subroutine calls, as shown below.

```
10 GOSUB 1000:REM initialise functions
20 GOSUB 2000:REM set up variables
30 GOSUB 3000:REM set up screen
40 GOSUB 6000:REM play game
50 GOSUB 7000:REM new game wanted?
60 IF answer$ = 'YES' THEN GOTO 40
70 END
```

The program would then continue with the subroutine definitions from line 1000 onwards. I separate these two areas of the program by using a blank line except for a ':'. Thus

```
100 :
```

leaves a blank line within a program. After the main subroutines, which are those called by the body of the program, I define the subroutines that are called within the main subroutines. After the subroutine definitions, I have the DATA statements that are used by the program.

12 MSX Machine Code

The Z-80 CPU that is at the heart of the MSX computers is controlled by the program that is contained in the MSX ROM. This program is written in a coded form, called Machine Code. These machine code instructions are the only things that the Z-80 CPU can understand, and when we type in a BASIC program and RUN it, the BASIC program is executed as a series of machine code routines. There are, in total, almost 700 different instructions that the Z-80 CPU can carry out, and so it is obvious that this Chapter can do no more than provide an overview of machine code programming on the Z-80. What will be done in this Chapter, however, is to look at specific points about machine code programming on the Z-80 in the MSX computers.

Let's begin, therefore, by suggesting a couple of books that will go into Z-80 programming in detail. My personal favourites are 'Z-80 Assembly Language Programming' by Lance A. Leventhal and 'Z-80 Microprocessor Programming and Interfacing', Book 1 by Nichols, Nichols and Rony. The only problem with the latter of these two works is that it is a book aimed at a particular machine. However, the principles and explanations given are well worth looking at. Of these, I would suggest the latter to the absolute beginner.

So, on with the Z-8Ø. The best way to start with the processor is the Program Counter, a 16 bit register that enables the CPU to keep track of what part of a machine code program it is executing at that time. When the computer is first turned on, the PC is set to a value of zero. This causes the CPU to execute the machine code program that it finds starting at address Ø in memory, which in the case of the MSX computers is the ROM. You will remember from Chapter 1 that the CPU gains access to the memory of the computer by the address bus, and accesses the contents of a given memory location by means of the data bus. When an instruction is read from memory, the Z-8Ø automatically knows if there is data to follow, or if the byte fetched is the first byte of an instruction that consists of several bytes, and the PC is automatically updated by the correct amount so that the Z-8Ø will be able to access the next instruction once it has executed the current one.

As well as the PC, there are other registers in the CPU. These are shown in Figure 12.1, and are called the Main Register Set and the Alternate Register Set.

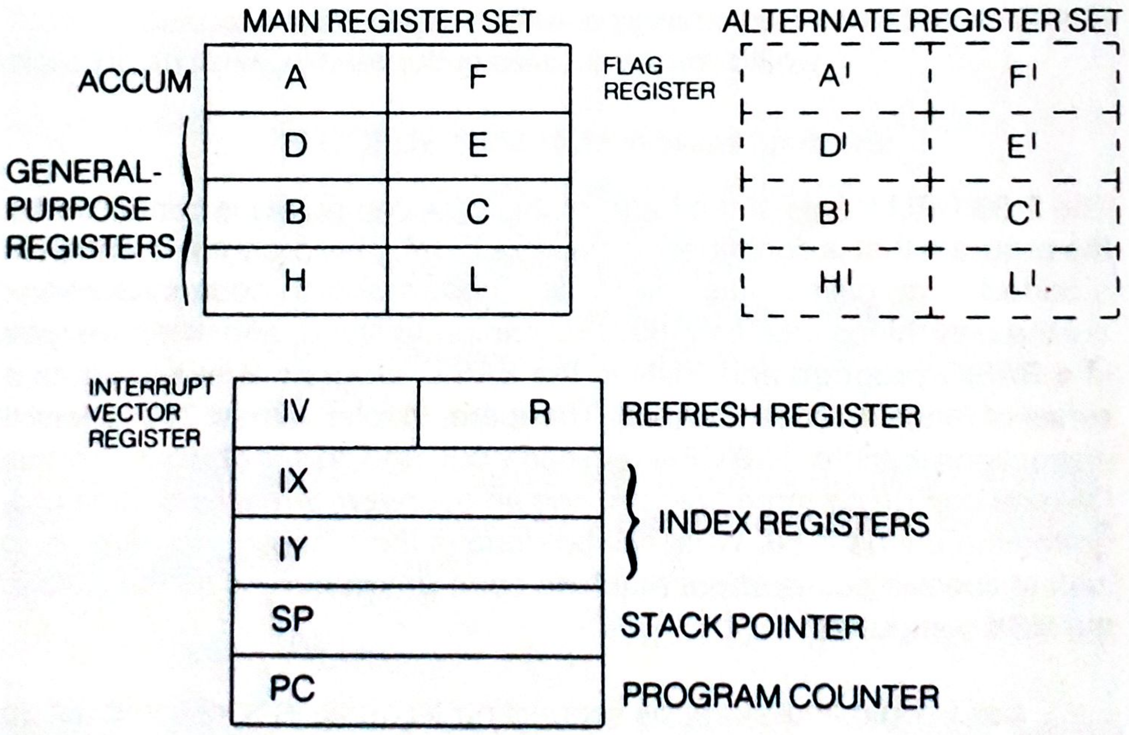


FIGURE 12.1 REGISTERS OF THE Z8Ø MICROPROCESSOR

The most important of these registers is probably the A register, or the accumulator, as it is called. The CPU performs much of its arithmetic on data stored in the accumulator, and there are a wide range of instructions in the CPU instruction set that access the accumulator. The accumulator is only large enough to allow numbers between 0 and 255 to be present in it, but the BC, DE and HL registers are each big enough to hold any number between 0 and 65535. These latter registers can also be used as single registers, each single register being capable of holding a number between 0 and 255.

The F register is the Flag Register of the CPU; this performs a job that is similar to that done by the Status Register of the VDP, but here each bit of the register signals a particular fact about the CPU. The values of the bits within the Flag register are affected by arithmetic operations, comparisons between two bytes of data and other operations.

The I,R, IY and IX and SP registers are all fairly specialised registers, and I will not discuss them here. Anyone wanting to make full use of the Z-80 CPU should now get hold of a book on the subject, study it, and apply what is learnt to the particular case of the MSX computer. The rest of this Chapter will be about the particular problems of machine code programming on the MSX computer. Some machine code routines will be listed as examples, and the interested reader is advised to work through them carefully using one of the reference works listed above.

The first problem with regards to machine code programming on the MSX computers is where to store the code and how to write machine code programs to the memory.

Storing Machine Code Programs

The most obvious way to store machine code programs on the MSX computers is to use the CLEAR command to set up an area of memory that is untouched by the BASIC program or variables.

For example, let's assume that you want to reserve 100 bytes of RAM in which to place your machine code routines. Use of the CLEAR command will enable you to set up an area of memory between the end of the BASIC workspace and the start of the System Workspace. The way to reserve memory in this way is as follows. First of all, it is useful to know the normal end of BASIC workspace. On a 16k machine, this is address 61583. You can calculate this by resetting your computer and evaluating the expression below.

```
PRINT (start of RAM) + FRE(0)
```

Armed with this knowledge, we use the command below to reserve the memory.

```
CLEAR 200,61482
```

The RAM between 61483 and 61583 is now free for your machine code routines, which will be perfectly safe from accidentally being overwritten by BASIC.

There are not many other places on an MSX computer where it is feasible to put machine code routines. Other machines have had machine code stored in rather strange places such as a REM statement in the first line of the program or as the contents of a string variable. However, these techniques are a little difficult to apply to the MSX computers.

In the first case, the position of a REM statement as the first line of the program will be the same for all computers that have the same amount of memory. However, if the amount of memory possessed by a computer changes, the position of the REM statement in RAM would alter.

With regards to storing the code in a string variable, the problem of calculating the position in RAM of the string every time you wish to call the machine code routine renders this method of code storage inefficient.

Note that it is not possible to store machine code programs in VRAM. This is because of the fact that the CPU cannot directly access the Video memory, but has to access it through the Video Display Processor. If you wished, however, you could store data for use by machine code programs in unused areas of VRAM. This is, though, a little risky, as changing screen mode could easily destroy the data stored. Also, reading this data would be a little slow, as all reads from VRAM would have to be done through the VDP.

Entering Machine Code Programs

As you will learn from any text on Z-80 machine code programming, a machine code instruction, such as

```
LD A,23
```

which loads the accumulator with the number 23, is represented in memory as numbers, which are recognised by the CPU as instructions.

The command above, for example, is coded as the two numbers

```
62,23
```

62 is the number representing the LD A,n instruction, where n is a number between 0 and 255. 23 is the data item needed for this command.

There are programs available for many home computers that translate the pseudo english instructions such as LD A,23 into the appropriate numbers. These programs are called ASSEMBLERS, but until one becomes available for the MSX computer we will have to do the translation job ourselves. (Or, if we're feeling ambitious, we could write our own....!) The technique of writing down the Z-80 instructions on paper and then using a reference book to get the numeric codes for each instruction, is called HAND ASSEMBLY, and is really only suitable for fairly short programs. Let's look at an example of a Z-80 machine code program.

```
LD A,23
LD (52000),A
RET
```

The RET instruction at the end of the program is essential to ensure that control passes back to BASIC after the program has been executed by the CPU. If it is omitted, then the computer will almost certainly 'Crash'. This is the name given to the result of a machine code program that fails to work properly. The only way to recover control is to press the RESET button, but this has the effect of wiping out your program from memory. The sequence of commands shown here has the same effect as the BASIC command

```
POKE 52000,23
```

The program below will poke the bytes that make up the program into a particular area of memory. Note how the address 52000 is stored as two bytes, the least significant byte being poked into RAM first.

```
10 CLEAR 200,61482
20 FOR I=61483 TO 61488
30 READ N:POKE I,N
40 NEXT
50 DATA 62, 23:REM data for LD A,23
60 DATA 50,32,203:REM data for LD (52000),A
70 DATA 201:REM data for RET
```

The code is now firmly ensconced in a safe area of memory. All that remains to us to do now is to execute the machine code program. This is done by using the USR command.

Use of DEFUSR and USR

Although these commands have been previously mentioned, we'll take another look at them now. Before we can call any machine code program with a USR call, we must inform the computer of the address of the machine code in question. Let's assume that we want to call our program with the USR1 command.

To set the address that we wish the USR1 command to call, we use a DEFUSR command. The full syntax of the command is

```
DEFUSR n = integer
```

where n is a digit between Ø and 9, and integer is a number between Ø and 65535 which is the address of the first byte of the machine code program that is to be called by the USR n command. As we want to call our routine with USR 1, n = 1. The start address of the machine code program is at location 61483, and so the DEFUSR command needed to tell the computer of the whereabouts of the program to be executed is

```
DEFUSR 1 = 61483
```

This expression must be executed before we try and call the machine code routine with the corresponding USR call. If we'd wanted to use another USR call, such as USR 2 to call the routine, then simply replace the digit 1 with 2. Note that if you want to call the routine with a USR Ø instruction, then the digit is not needed. Thus to call the code with a USR Ø instruction,

```
DEFUSR = 61483
```

would define the address. Also, the number Ø is not required in the USR call itself.

Once the address has been assigned in this way, then we can use the USR 1 command to call the routine, as shown below.

```
PRINT USR1(Ø)
```

The argument passed over in the brackets of the USR command is in this case a dummy argument, and is ignored by the machine code routine. All that appears to happen is that the value of the dummy argument is printed to the screen. However, if we now use the PEEK command to ascertain the value held in location 52000, it will be found to be 23. The USR command does not have to be used with the PRINT command; it can be used as part of a variable assignment. This is particularly useful if the machine code routine is to return a value to the BASIC program that called it.

```
L=USR1(0)
```

is an example of this method of using USR. Although the argument is not used in this particular application, we will shortly look at how we can make use of it in our machine code programs.

Passing Parameters

It is valuable to be able to pass parameters to a user machine code routine. This means that the values being held in BASIC variables can be made use of by the machine code routines if required. In the MSX computers, the value passed over in the brackets of a USR command is placed at a particular place in memory. The area of memory that holds either the parameter or information about where the machine code routine can find the parameter is called the PARAMETER BLOCK. It is in two parts. A single byte located at address 63075 holds information relating to the type of parameter passed to the routine – i.e. whether it is integer, real or string. The rest of the parameter block is located between addresses 63478 and 63485. When you call a USR routine, a value is placed in location 63075 relating to the type of variable as follows.

Note that the contents of the parameter block do not appear to make much sense if you PEEK them out from BASIC. This is not likely to be a hindrance, however, as we need to access these locations only from machine code.

Passing Parameters

The parameter passed to a machine code routine for use by that routine can be of any type: integer, Single or Double Precision or string. For example,

```
L=USR(I%)
L=USR("TEST")
L=USR(3.345#)
```

are all legal. The various types of parameter are passed as follows.

Integers

Location 63Ø75 will hold the value 2. The value of the integer passed over is stored in the two bytes &HF7F8 and &HF7F9. The value is stored with the least significant byte in location &HF7F8.

Strings

If a string constant or string variable is passed as a parameter of the USR routine, then location 63Ø75 will hold the value 3. Locations &HF7F8 and &HF7F9 hold the address of a STRING DESCRIPTOR, the least significant byte of the address being held in &HF7F8.

The String Descriptor is a three byte long block of memory which holds details about the string. The first byte of the block, which is pointed to by the value held in &HF7F8 and &HF7F9, is the length of the string parameter. The next two bytes hold the address of the string being passed, low byte first.

Single Precision

Location 63Ø75 holds the value 4, and the single precision number is placed in locations &HF7F6 to &HF7F9.

Double Precision

Location 63Ø75 holds the value 8 on entry to your machine code routine, and the Double Precision number is placed in locations &HF7F6 to &HF7FD.

Let's now take a look at a sample machine code routine that accepts a parameter from BASIC, operates on it, and passes the value back to BASIC. However, before we do this, we must learn how to pass values back to BASIC from the machine code routine. One way would be to leave the result of the machine code routine's operation in a specific memory location from which we could PEEK it out. A second method would be to write the routine so that we use the Parameter Block to transfer the data back to the BASIC program. This method is much more elegant, and so we'll look at this method now.

Returning Values

Normally, unless we specify otherwise, the value returned by a USR call is the value, be it string or numeric, that was pased over to the machine code routine in the brackets of the USR command. However, by suitable instructions in the machine code routine it is possible to return a value calculated by the machine code routine.

Integers

To return an integer value to BASIC, set location 63Ø75 to hold the value 2. Then write the integer value to locations &HF7F8 and &HF7F9, putting the low byte of the number to be returned in location &HF7F8.

Strings

Set location 63Ø75 to hold the value 3. Then set locations &HF7F8 and &HF7F9 to point to a String Descriptor block somewhere in memory. The String Descriptor block used here has the same structure as that for passing parameters to machine code routines.

Single Precision

Set location 63Ø75 to hold the value 4, and put the bytes that make up the number into locations &HF7F6 to &HF7F9.

Double Precision

Set location 63Ø75 to hold the value 8, and then put the bytes that make up the number into locations &HF7F6 to &HF7FD.

On return from the machine code routine, the parameter set up by any of the above methods in your routine will be returned instead of the argument that you passed over to the routine. Note that it is possible to get Type Mismatch errors if you configure your machine code to return a string value and you call the machine code routine using a command, say, like

```
100 LET L=USR(0)
```

Let's examine a routine that accepts a parameter passed over in a USR statement and returns an integer value, after first adding 3Ø to the value passed over to it. It expects an integer to be passed to it.

```
LD HL,(&HF7F8)  42,248,247
LD DE,3Ø        17,Ø,3Ø
ADD HL,DE       25
LD (&HF7F8), HL 34,248,247
LD A,2          62,2
LD (63Ø75),A    5Ø,99,246
RET             2Ø1
```

The first line of this program loads the integer passed over to the routine in the USR command. The next two lines then add 30 to the integer passed over, and the next 3 lines place the modified argument in position for the return to BASIC and set location 63075 to signal that an integer value is to be passed back. The numbers to the right of the listing are the numbers representing the instructions. Poke them into an appropriate area of memory, and then set a USR call to that address, e.g.

```
DEFUSR1 = 61483
```

This statement

```
PRINT USR1(45)
```

would then return to the value 75, which would be printed to the screen.

This ability, to return values from machine code routines to BASIC programs, is rather useful, as it enables the programmer to write routines to perform tasks that would be slow in BASIC or not feasible, but still maintain the use of BASIC variables for getting data to the machine code routine.

Accessing other Devices

We have already seen how we can improve our programming of the MSX computer by direct access of devices such as the Programmable Sound Generator and the Video Display Processor. We've already seen how we can directly modify the contents of PSG and VDP registers by accessing the IN/OUT map of the computer using the BASIC INP and OUT instructions. This was described in Chapter 10, where we also saw how to directly modify the contents of Video memory using the VDP.

Not surprisingly, we can perform a similar task from machine code by using the Z-80 commands that directly use the IN/OUT map of the CPU. Anyone wishing to use these devices from within machine code programs is advised to read the relevant Chapter, as the techniques for controlling the behaviour of the VDP, say, by altering the register contents, will remain the same whether we alter the register in BASIC or machine code.

No particular problems will be encountered in writing values to the PSG or the PPI. The command

```
OUT n,A
```

can be used from Z-8Ø machine code, writing the contents of the accumulator to I/O location n. Further details of the command will be given in either of the reference works mentioned at the start of the chapter. To read a value from a particular I/O location, the command

```
IN A,n
```

can be used, which will read a value from I/O location n and place it in the accumulator. The methods used for reading the ports and writing to the I/O ports for each device will be the same as outlined in Chapter 1Ø.

When it comes to direct access of the VDP from machine code, the guidelines mentioned in Chapter 1Ø should be followed, especially with respect to reading the status register before writing to the VDP. The only problem with accessing the VDP from machine code is one of timing. We cannot write values to the VDP registers at a frequency of more than once per 4.3 milliseconds. Thus, it may be necessary for you to put time delay loops into your machine code programs to ensure that you write to the VDP no more often than this.

Apart from this, the VDP can be accessed in an analogous fashion to the way in which it was accessed in Chapter 1Ø. It is in machine code access to the Video RAM that the auto increment feature becomes valuable; once an address has been written to the VDP, repeated OUT instructions can be used to send other bytes to the VRAM, the location to which the data byte is written being incremented each time. This makes data transfer between the CPU and the VRAM easier, as we only need to set the address up once. It is, obviously, of no use if the locations to be written to in the Video RAM are non-contiguous.

Hooks

No, this is not about fishing. A HOOK is a means by which the MSX programmer can modify the behaviour of the computer when it is actually running the MSX ROM routine. These feat is achieved by virtue of the fact that various routines in the ROM call, at some point in their execution, a location in RAM. As an example, the routine that is executed each time there is a VDP interrupt calls a RAM routine at address &HFD9A. Inspection of this location shows it to contain a RET statement, and so normally the call to this address has no effect at all. This location, and the 4 bytes following it, form a Hook SERVICE BLOCK, and on reset or turn on are all set to hold the value 2Ø1, which is the code for RET. The way we modify the behaviour of the ROM routine is to place a suitable CALL or JP instruction at the address to which the ROM routine jumps. An example of this is shown below.

```
CALL 52000
RET
RET
```

The CALL command occupies the first three bytes of the Service block, and the two RET statements are already occupying the remaining bytes of the block.

The effect of this routine is to cause a jump to the subroutine at location 52000 to be made every time an interrupt occurs. If you try this, please set up the subroutine at location 52000, or wherever you decide to put the Hook Service routine, before you change the Service Block to point to it. If you don't do this, then the result will almost certainly be an immediate crash of the computer.

That completes this review of the special knowledge that is useful when programming on the MSX computers in machine code. Use the books mentioned to develop your machine code programming skills, and practice them by writing small routines first. Don't be too ambitious; MSX BASIC is an extremely powerful implementation of the language, and much can be accomplished in it without recourse to machine code.

Appendix
Number Systems

Counting is an automatic act for most of us, but when we write down the number 234, what do we actually mean? To be able to understand this method of writing down numbers it is necessary to first examine the whole business of counting — a process that is almost intuitive to us.

The first thing to note is that the number is written in columns, across the page from left to right. These columns are not all of the same significance to the value of the number. Let's examine a typical whole number: 234. We say that the rightmost column in such a number is called the UNITS column, and in this column we count in what are called DIGITS. In our system of counting, based on 10 digits and hence called the decimal or DENARY system, the digits are 0,1,2,3,4,5,6,7,8 and 9.

In any system of counting the number of digits available has a special name—the RADIX or BASE of the system. Thus, with the 10 digits of the decimal system, we have a base of 10, as 10 different digits can appear in the units column. The process of counting in units is simply one of stepping through the available digits in the correct order. So, for the decimal system, we step through the digits Ø to 9 in that sequence. However, what happens when we've gone through them all?

In more specific terms, what happens when we get to 9 in base 1Ø, and another count occurs? Well, the units column is set back to zero, and we move a count over to the next column on the left. The count that is moved over is called the CARRY. Each count in the column to the left of the units column differs from the last count in that column by the radix of the number system. In the decimal system the radix is 1Ø and this column is thus called the tens column. Counting then proceeds in the units column until the next carry is generated and a count is added to the tens column. Similarly, once we reach 9 in the tens column, a carry is generated to the next column to the left, in which counts differ from each other by the radix multiplied by itself. In the decimal system this is equal to 1ØØ, and so we call the column to the left of the tens column in the decimal system the hundreds column. If we apply this information to the number 234, then we see that we have 4 units, 3 tens and 2 hundreds. Each count in the units column increases the value of the number by one, each count in the tens column by ten and each count in the hundreds column by one hundred.

However, there are number systems other than this decimal system. The Babylonians used a base 6Ø system, whence derives our methods of measuring time and angles. A relic of a base 12 system is used by people who still measure in feet and inches, there being 12 inches to 1 foot. This system of measuring also makes use of base 3, as there are 3 feet to the yard. In any of these number systems, no matter what the base, each count in the column to the left of the units column increases the value of the number by the radix of the number system.

If computers worked in base 1Ø then we would have no reason to bother about number bases other than 1Ø. However, owing to the fact that it is easier to build electronic circuits that detect either the presence or the absence of a signal rather than circuits able to differentiate 1Ø separate signal levels, computers work in base 2. Your MSX really recognises 2 digits only, Ø and 1, and it recognises these as voltage levels on its internal wiring. These digits are represented by a +5 volt signal for the digit 1 and a Ø volt signal for the digit Ø. This number system with a base of 2 is called the BINARY system. So, armed with the digits Ø and 1, how do we count in binary?

Well, instead of the units, tens and hundreds that we have in the decimal system, we have units, twos and fours in binary. When we have a 1 in a column, and a further count occurs, we generate a carry to the next column and replace the one with a zero. The carry goes to the column to the left of the one being added to. Thus the process of counting in binary is entirely analogous to that of counting in decimal.

As a general rule in counting we say that the any column has less significance to the value of the number than the column to its left. That is, a count in the units column adds less to the value of a number than does a count in the tens column.

In whole numbers, therefore, the units column holds what is called the LEAST SIGNIFICANT FIGURE and the column furthest to the right holds what is called the MOST SIGNIFICANT FIGURE of the number. The table shows the effects of counting in binary (the subscripts show the radix or base):

1_{10}	0001_2
2_{10}	0010_2
3_{10}	0011_2
4_{10}	0100_2

Note that the leading zeros make no difference to the value of the number. If we analyse one of these binary numbers, say 0011, then we can see that we have one unit and one two. This gives us the value of 3 for the number in decimal. If we add on a further count, then we generate a carry from the units to the twos column, placing zero in the units column. However, the twos column already has a 1 in it and so we must generate another carry from the twos column to the fours column, placing a zero in the twos column. Thus, we get 0100, or 4 in decimal. If we disregard the first zero, which makes no difference to the value of the number any way, we see that to represent the number 4 in binary takes three digits, as opposed to only one digit in decimal. In the decimal system, the largest number we can show with three digits is 999, whereas in binary the largest number we can show in three digits is 111, or 7 in decimal.

This gives us another general rule about number systems: as we decrease the number of digits available, the value of the largest number that can be represented in a given number of digits also decreases. Conversely, by increasing the number of digits available we can increase the value of the largest number that can be shown in a given number of digits. As the radix is the number of different digits in a number system, the value of the largest number that can be shown in a given number of digits depends upon the radix of the number system.

So far we have dealt with the way a number is made up, and we've looked briefly at the binary system. As our computers work in binary, it's now time to have a look in greater detail at the binary system, and how we can perform simple arithmetic in this number system.

Binary Arithmetic

The simplest arithmetic operation in the decimal system is addition, as it can be seen as an extension of the processes that we go through when we are counting. Let's look at the process of addition in binary by first examining the addition of 1 to 0010. This addition is a straight-forward act of counting, but we'll go through the normal processes of performing a binary addition. Firstly, we put the figures down so that the least significant digits of each number line up as shown below.

```
 0010
 0001
-----
```

If we ever encounter a lone number, like the 1 in the example above, we assume that it is a unit unless we are specifically told otherwise. Similarily, the binary number 10 would be assumed to be 0 in the units column and 1 in the two's column. When performing addition, in any number system, we always start at the least significant figure and work our way across from right to left.

Thus, in the above example, we start by adding the 1 to the 0 in the units column. This gives us the value of 1 in the units column, and nothing to carry over to the twos column. This gives us the situation below.

```
 0010
 0001
-----
    1
```

We now add together the digits in the twos column. In this case we add together 1 and 0. If there was no number in the twos column of one of the numbers involved in the addition problem, then we would simply assume that it was a zero, as we have done in this example. Again, this gives a value of 1 to place in the twos column and no carry, giving us the situation below.

```
 0010
 0001
-----
 0011
```

We could now go on to add the digits in the fours column, but there is no point as all the rest of the digits are zeros.

Thus we have a result of 0011 in binary. Let's now examine some more binary numbers, and then attempt some more additions. Note the fact that we need 4 digits to show binary numbers between 8 and 15. The new column that we use is called the eights column.

Decimal	**Binary**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Let us now consider the addition of 0110 to 0111. In this problem, we can examine what happens in addition when a carry is generated.

```
                0110
                0111
     Carry
              ______
                0001
```

Above, we can see the result of adding together the two units columns. No carry is generated. Let's now go on to the twos column.

```
                0110
                0111
     Carry        1
              ______
                0001
```

The carry is generated and a zero placed in the twos column. Now, on to the fours column. First of all, we add together the digits in the fours column ignoring the carry from the twos column. This gives us 1 + 1 = 0 carry 1. Now add the carry. This gives us 1 from the carry from the twos column added to the zero generated by the addition of the digits in the fours column. Thus, we get the result below.

```
          0110
          0111
Carry     11
          ----
          0101
```

Now we move on to the eights column. Although there are zeros in this column in both the figures that we are adding together, we still have to consider the carry that was generated from the addition in the fours column. Simply add the carry to zero:

```
          0110
          0111
Carry     11
          ----
          1101
```

Thus the sum is completed. If we try to add together 1111 and 0001 then we generate a whole series of carry digits.

```
          1111
          0001
Carry        1
          ----
          0000
```

Now we have:

```
          1111
          0001
Carry       11
          ----
          0000
```

This eventually results in the carry going into the fifth column from the right, which in binary is called the sixteens column. Although we have only got 4 digits in the two numbers that we are adding, we assume the existence of zeros ahead in the sixteens column in both these numbers.

```
          01111
          00001
Carry     1111
          -----
          10000
```

Thus the result of this addition is 16, as we have no units, twos, fours or eights but 1 sixteen. This indicates that by simple examination of the binary numbers we can work out the highest value number that we can represent in a given number of digits.

With four binary digits, the highest representable number is 15, and with three binary digits, the highest representable number is 7. Try it for yourself. With five digits, the highest number will be the one that has all the digits in it set to 1. This would give

$$(1 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (1 * 1) = 31$$

The largest number that can be represented in a five-digit binary number is thus 31. Examination shows us that in the binary system the largest number that can be represented in digits is

$$2^n - 1$$

Putting this into English, the largest number representable is 1 less than 2 multiplied by itself n times. As a test of this general statement, let us put some values of n into the equation and see what we get. For n = 2, 2 multiplied by itself twice is 4. Thus the highest number that is representable in two digits is 4 −1, or 3. To test this, let us count out the possible numbers with two binary digits in them, writing them down as we go:

Binary	**Decimal**
00	**0**
01	**1**
10	**2**
11	**3**

To count any further than this, i.e. to add another unit, would generate a carry to the fours column, requiring a third digit. Thus, the rule is upheld. This simple rule can be shown to be true for any number system, and we can write the equation in a more general form as

$$x = R^n - 1$$

where x is the largest number that can be represented in a given number of digits, n is the number of digits and R is the radix of the number system in use. If we test this for base 10 and two digits, then we get a maximum value of 10 * 10) −1 or 99, which is correct. To go any higher, i.e. to represent the number 100, we need three digits.

Before we go any further with binary arithmetic, it is necessary to bring in a few terms that are commonly used in this field and that you will certainly meet in machine code programming. The phrase binary digit is quite a mouthful and has been replaced in common usage with the word BIT. This comes from Binary DigIT.

Just as we have the most and least significant digits in a number in any number system, we can have the most and least significant bits in a binary number. When talking in bits, it is not really necessary to quote the number base in which you are working, it has to be binary. The Least Significant Bit of a binary number is usually called the LSB for convenience. Similarly, the Most Significant Bit is called the MSB. A carry digit is called a carry bit. The length of a binary number, in terms of the number of digits in it is quoted in bits. Thus a 4 bit number has four digits in it. As the highest value that can be shown in a binary number depends upon the number of bits available, an 8 bit number has a higher maximum value than a 4 bit number can have.

After that brief interlude, let us get back to the subject of simple arithmetic. In the decimal system, numbers may have values between Ø and 1, and between the other whole numbers. Some examples are Ø.5, 4.7 and 234.567. These numbers are called REAL numbers, and the whole numbers are called INTEGER numbers. In decimal, we call the "." in the number the DECIMAL POINT. The number to its immediate left is in the units column, and those numbers to the right of the decimal point are said to be FRACTIONAL parts of the number. In the decimal system, the first fractional column is called the tenths column, the next the hundredths and so on. Thus, the significance of a digit in the number decreases as we go from the decimal point to the rightmost column of the number. In numbers with a fractional part, the least significant digit is in the rightmost column instead of in the units column.

234.567

Thus in the above number, the digit 7 is the least significant digit of the number, and it represents 7 thousandths of the value of one count in the units column. Fractions are also possible in the binary system, where we have a binary point instead of a decimal point. In a binary number with a fractional part, the column to the right of the point is called the halves column, the next one to the right is called the quarters column and so on. We add binary fractions together in exactly the same way that we add together binary whole numbers, starting at the rightmost column and working across, generating carry bits when necessary. We can carry a bit from the halves column across the binary point to the units column if this is needed.

After addition, the most commonly used arithmetic operation is that of subtraction. This is slightly more involved than the addition of two numbers, and so we'll look at it in some detail. It is effectively the reverse of addition. If we examine the simple binary additions, we can draw up some rules for subtracting binary numbers.

```
1 + 1 = 10     therefore     10 - 1 = 1
1 + 0 = 1      therefore     1 - 1  = 0
0 + 0 = 0      therefore     0 - 0  = 0
```

The problems start when we consider the subtraction of 1 from 0. The way around this apparent problem is that we "borrow" a digit from the next most significant column of the number. The significance of the borrowed digit is that of the column from which it is borrowed, so if we were to borrow a digit from the twos column, the digit would have a value of 10 and not 1. This is the exact analogy of what we do in decimal subtraction. To provide a more concrete example, let us actually do a binary subtraction.

```
           1101
          -0101
Borrow
          -----
```

As usual, let us start in the units column, with the subtraction of 1 from 1. This gives us 0.

```
           1101
          -0101
Borrow        0
          -----
           0000
```

Moving on to the twos column, we get the situation below. Again, no borrows are necessary to do the subtraction.

```
           1101
          -0101
Borrow       00
          -----
           0000
```

We can now calculate the result from the fours column and the eights column, and it transpires that in this problem we have no need to borrow any digits. The result is below.

```
           1101
          -0101
Borrow     0000
          -----
           1000
```

We must now look at a subtraction with borrow example in decimal before we can tackle the borrow problem in binary. In a subtraction such as 432 − 56, school children are taught to decompose the two numbers as follows:

$$\begin{array}{r} 432 \\ \underline{-56} \end{array} \quad \text{is equivalent to} \quad \begin{array}{rr} & 400 + 30 + 2 \\ -[& \underline{\quad\quad 50 + 6]} \end{array}$$

The subtraction proceeds by borrowing a ten from the tens column for the ones column:

$$\begin{array}{rr} & 400 + 20 + 12 \\ -[& \underline{\quad\quad 50 + \ \ 6]} \end{array}$$

We see in the tens column that 50 is greater than 20, so another borrow is required from the hundreds column:

$$\begin{array}{rr} & 300 + 120 + 12 \\ -[& \underline{\quad\quad 50 + \ \ 6]} \end{array}$$

The problem is now easily solved . . .

$$\begin{array}{rr} & 300 + 120 + 12 \\ -[& \underline{\quad\quad 50 + \ \ 6]} \\ & 300 + \ \ 70 + \ \ 6 \end{array}$$

. . . to give the answer of 376.

A similar approach may be used with the following binary subtraction:

$$\begin{array}{r} 1100 \\ \underline{-0101} \\ 0111 \end{array} \quad \text{is equivalent to} \quad \begin{array}{r} 12 \\ \underline{- \ \ 5} \\ 7 \end{array}$$

We now proceed to decompose the two numbers in the binary subtraction:

$$\begin{array}{r} 1100 \\ \underline{-0101} \end{array} \quad \text{is equivalent to} \quad \begin{array}{rr} & 1000 + 100 + 0 + 0 \\ -[& \underline{\quad 0 + 100 + 0 + 1]} \end{array}$$

As before, borrows are required until the numbers on the bottom row are smaller than their counterparts on the top row.

The ones column borrows from the twos column; since the twos column in the top row is already 0, this leaves a value of −10;

```
  1000 + 100 +(−10) +10
−[   0 + 100 +     0 + 1]
 ________________________
```

The twos column borrows from the fours column to eliminate the −10 value:

```
  1000 +   0 +10 +10
−[   0 + 100 + 0 + 1]
 ___________________
```

The fours column borrows from the eights column to eliminate the 0 value:

```
  0 + 1000 + 10 + 10
−[0 +  100 +  0 +  1]
 ____________________
```

The subtraction now proceeds . . .

```
  0 + 1000  +10 + 10
−[0 +  100 +  0 +  1]
 ____________________
  0 +  100 + 10 +  1]
```

. . . to give an answer of 100 + 10 + 1 which is equivalent to 111 or, in decimal, 7; Q.E.D! This method of binary subtraction is just as laborious for computers as it is for humans — is there an easier way?

In decimal arithmetic, subtraction can often generate numbers that are less than zero. These are called NEGATIVE NUMBERS, and in decimal are prefixed by a "−" symbol. They are generated when we subtract a number from another, smaller number. Negative numbers can also be generated in binary arithmetic, but we shall represent them using TWOS COMPLEMENT.

Before we go into the details of where this name comes from, let us see how we can represent both positive numbers (i.e. those with a value of greater than zero) and negative numbers in 4 bit twos complement. (The reason it is essential to quote the number of bits when dealing with twos complement will become quite obvious in a short while.) Note closely what happens as we pass through zero into the negative numbers:

Decimal	Twos complement
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
−8	1000

We very quickly note several things. In four bits we can usually represent the numbers 0 to 15, that is, 16 separate numbers with 15 as the highest value that a 4 bit number can possess. Here, however, we still have 16 different numbers represented but the highest value that can be represented is 7. Half the codes represent positive whole numbers and zero, and the other half represent negative whole numbers. The part of the number that says whether the number is a positive or a negative number is the MSB of the number. In the preceeding example, it is 0 for a positive number and 1 for a negative number. If we had an 8 bit twos complement number, the MSB would still indicate whether the number was positive or negative in exactly the same way.

With regards to the range of numbers that can be represented in a given number of bits, we can work this out from the equation below.

$$-2^{n-1} < x < 2^{n-1} - 1$$

This equation gives the range of numbers that can be represented in twos complement notation in a given number of bits. Thus, if we let n equal 4, we see that the range is −8 TO +7. This is what we found by examination above. Thus, if we have been told that a number is in twos complement format we can tell whether it is positive or negative simply by looking at the MSB. A further point to note is that a positive number shown in twos complement notation is the same as it is when represented in normal binary notation.

Being able to represent numbers in this way is extremely important. First, it allows us to show numbers that have values less than zero, and, second, it allows us to reduce the process of subtraction, with all its inherent complexity, to one of addition. This latter feature is very important to us in the field of computing, as it is easier for computer circuits to do addition than subtraction. Let us see how this works. By examining the list of twos complement numbers given above, we can see that we have both negative and positive numbers. In decimal arithmetic, if we add together a number and its negative equivalent we get zero.

What happens if we attempt this with twos complement numbers? Let us try it with 1 and −1 in 4 bit twos complement addition.

```
      0001          1
  +   1111      + (−1)
     ------       ----
     1 0000         0
```

At first glance our result of 1 0000 is certainly NOT equal to zero, and we know from experience that 1 + (−1) most certainly IS equal to zero! The first thing to note is that we now have a 5 bit number. Remember earlier it was stated that the number of bits is very important in twos complement arithmetic? Well, here is where the importance becomes apparent. We started off with two 4 bit numbers, and so in twos complement arithmetic the result should be a 4 bit twos complement number as well. If we dispose of the MSB, we do in fact end up with zero as our answer! Thus, using twos complement arithmetic, we have made subtraction as easy to perform as addition. The disadvantage is, of course, that you need more bits to represent a given positive number in twos complement than you do in normal notation. However, this is easily offset by the advantages that twos complement notation gives us.

So, how do we convert a binary number into twos complement notation? It is quite a simple operation. The process of COMPLEMENTING is the first stage, and is a term used to designate the process of replacing every 1 in the number with a 0 and every zero in the number with a 1. Thus the number 1010 when complemented becomes 0101. The second stage of the operation is simply to add 1 to the number obtained by the complementing step.

Thus to find the twos complement of 0010, we perform the following steps.

1. Complement the number; this gives 1101.
2. Add 0001 to the number; this gives 1110

That is all there is to getting the twos complement of a binary number. Once this has been done, we can perform subtraction very easily.

The steps in performing twos complement subtraction of two numbers are as follows.

1. Find twos complement of first number.
2. Find twos complement of second number.
3. Add the twos complements numbers together.
4. Discard the MSB.

What happens about finding the 4 bit twos complement value of 1000? If we follow the above instructions we get the number 1000. This cannot be right, as we've ended up with the number that we started with! Also, the first number, with its MSB set to 1, should give a 4 bit twos complement that has its MSB set to zero. An examination of the equation on page 199 gives us the solution to this apparent problem. This equation shows that it is not possible to represent the number +8 in 4 bit twos complement notation, and so it is not surprising that we get a funny value for the twos complement. This problem is called OVERFLOW, and occurs as the result of a twos complement operation that goes outside the range of numbers that can be represented in the number of bits in use. This again indicates the importance of the number of bits in use being specified for twos complement operations.

When using twos complement notation, not only is it important to quote the number of bits in the representation, but also that the numbers are in twos complement notation and not in normal binary. This type of problem can occur when working with different number bases. How do we tell what radix we are working in? A technique called SUBSCRIPT NOTATION is used in which a letter or number follows the number being considered. For example, 123D and 123_{10} both mean 123 decimal.

Addition and subtraction are really the only mathematical operations in the binary system that we need bother about, as the computer ultimately breaks down all its arithmetic operations into these simple procedures. Multiplication can be regarded as repeated addition, though there are other ways of performing binary multiplication in computers that will be considered later. Similarly, division can be viewed as repeated subtraction.

There is one more number system that we must examine before leaving this chapter, and that is a number system with a radix of 16. It is called the HEXADECIMAL system, and is of vital importance to the computing field as a means of conveniently representing binary numbers.

When we examine a binary number, it makes very little sense to the human eye but a great deal of sense to the computer. All numbers that we put into a computer eventually end up being represented in the computer as binary digits and it is often useful to have a rough idea of what a number will look like in binary without having to go through the long-winded process of writing down a string of 0s and 1s. Decimal is not really suitable for this representation, as there is no way, in normal binary representation, that you can easily convert a decimal number into a binary number. The decimal number 7 is easy to visualise, it being a short binary number: —0111. But what about 42 or 146? These are not at all easily visualised.

However, in the hexadecimal system, base 16, a single digit represents 4 bits of binary. As it has a base of 16, we will need some more characters to represent the digits between 9 and 15. We simply use letters of the alphabet, and so the digits used in the hexadecimal or HEX system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. Below we can see a binary representation of all extra digits.

Hexa-decimal	**Binary**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Thus, it is easy to see that we can convert from a hexadecimal number to a binary number simply by replacing each hexadecimal digit with its equivalent in binary. The hex number 5E can be converted to binary by simply writing 0101 instead of 5 and 1110 instead of E, giving the binary number 0101 1110. Addition and subtraction in the hexadecimal system work on exactly the same lines as they do in the decimal and binary systems.

In most microcomputer systems, including the MSX, the numbers are all represented in 8 bits, giving a maximum possible value of 1111 1111 in binary or FF in hexadecimal. Hexadecimal numbers can either be written as FFH, &FF or $FF, depending upon the notation used. A further advantage of hexadecimal is that it enables us to cut down the number of digits needed to represent an 8 bit number, from three digits in decimal to two digits in hex. Although the largest number that the computer's central processor can handle at once is 255, larger numbers can obviously be represented by various types of internal coding. The 8 bit number has reached such heights in the computing field that it has been given a name of it's own, the BYTE. The byte can be further subdivided into two 4 bit numbers, each of which is prosaically called a NYBBLE.

Finally, what about converting between the number systems that we've looked at?

We have already considered the conversion between hexadecimal and binary, so we'll go on to look at the conversion of normal binary numbers into decimal. If we examine the binary number 0110 then we see that we have 0 eights, 1 four, 1 two and 0 ones. We can find the decimal value as follows:

$$(0 \times 8) + (1 \times 4) + (1 \times 2) + (1 \times 0) = 6$$

This process can be carried out on any normal binary number. The reverse of this conversion, decimal to binary, is the most complex conversion we will deal with. It is essentially a process of repeated division. Let us convert 25 into binary:

$25 \div 2 = 12$	remainder	1 LSB of result
$12 \div 2 = 6$	remainder	0
$6 \div 2 = 3$	remainder	0
$3 \div 2 = 1$	remainder	1
$1 \div 2 = 0$	remainder	1 MSB of result

The binary equivalent of 25 is 1 1001, which we can verify by converting back to decimal. (Try it now.) Note how the digits of the binary number are generated from the remainders of the divisions. The process continues until the result of the final division is zero, irrespective of the value of the remainder. The remainder then becomes the MSB. This method is the basis of a general method of converting from decimal to any other base, by simply replacing the division 2 with the radix of the number base into which we wish to convert.

* * *

Of the number systems that we have talked about here, your MSX machine will understand integers, decimal numbers, hexadecimal numbers, octal and binary numbers. We will also run into binary numbers when we begin to write machine code programs.

We also need to know the PRECISION of numbers that we use in our programs. The best way to view the precision of a number is in terms of "accuracy". A SINGLE PRECISION number in the MSX system is 6 digits long, and a DOUBLE PRECISION number is 14 digits long. To see what this means, the number 10 000 001 would be represented in single precision as 1E6; the 'E6' representing 'ten to the power of 6'. Note that LSD (the 1) at the end of the number has disappeared. In double precision format the number can be represented in its entirely as 10 0 00 001, i.e. with more accuracy. In the MSX system, numeric constants are indicated by a trailing! or by containing an E. Thus, 10.76! and 1E-3 are both single precision constants. Double precision numbers are represented to the system by a trailing #, by containing a D instead of an E as an exponential sign, or by an absence of any type specifier. Thus typing in 1.2 will cause the number 1.2 to be represented in the machine in double precision format. Other double precision numbers are 123.4, 1.234D6 and 1.000345. The double precision format for numbers is the default setting of the machine and, unless told otherwise, the computer will treat all numbers as double precision.

[illegible]

Index

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 offending line numbers, or the paragraph of text.

MSX Exposed

Customer Registration Card

Please fill out this page (or a photocopy of it) and return it so that we may keep you informed of new books, software and special offers. Post to the appropriate address on the back.

Date . . . . .19 . . . .

Name . . . .

Street & No. . . . .

City . . . . .Postcode/Zipcode . . . .

Model of computer owned . . . .

Where did you learn of this book:

☐ FRIEND ☐ RETAIL SHOP

☐ MAGAZINE (give name) . . . .

☐ OTHER (specify) . . . .

Age? ☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ 25 and over

How would you rate this book?

QUALITY: ☐ Excellent ☐ Good ☐ Poor

VALUE: ☐ Overpriced ☐ Good ☐ Underpriced

What other books and software would you like to see produced for your computer?

. . . .

. . . .

. . . .

EDITION 7 6 5 4 3 2 1

Melbourne House addresses

Put this Registration Card (or photocopy) in an envelope and post it to the appropriate address:

United Kingdom

Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

United States of America

Melbourne House Software Inc.
347 Reedwood Drive
Nashville TN 37217

Australia and New Zealand

Melbourne House (Australia) Pty Ltd
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

THIS BOOK SCAN WAS COMPLETED
ON TUESDAY 18TH JUNE 2019
FOR THE BENEFIT OF ALL
BY PAUL KENNEDY OF
THURSO, SCOTLAND.

MSX

Melbourne House

£7.95

Here is a comprehensive guide to make you total master of your MSX. MSX EXPOSED is an encyclopaedia of solutions which begins with BASIC programming and takes you through to machine language.

The step-by-step format of MSX EXPOSED is designed to ensure that you will understand exactly how your MSX works, enabling you to take full advantage of the machine's capabilities. Every feature and program statement is carefully explained with the aid of simple demonstration programs, tables of vital memory locations and system variables.

MSX EXPOSED is the indispensable work book for every MSX owner — from first time users to the serious programmer.

Melbourne House Publishers

ISBN 0-86161-182-9

9 780861 611829