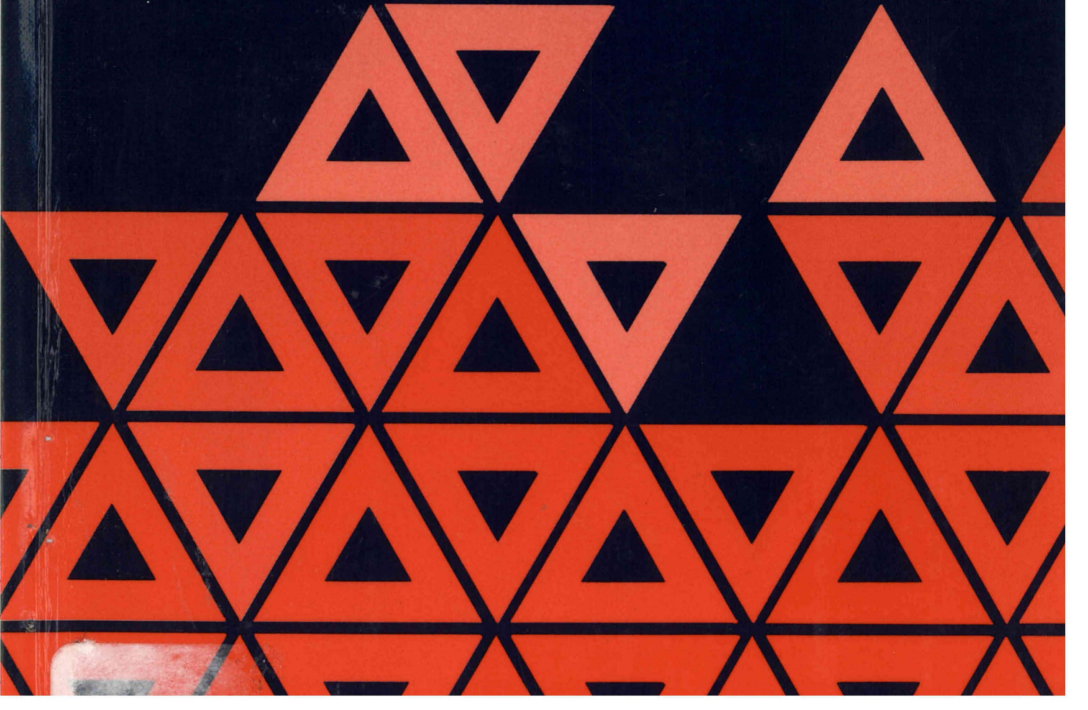
MSX Programming

Graham Bland



MSX Programming

Graham Bland

Pitman

PITMAN PUBLISHING LIMITED
128 Long Acre, London WC2E 9AN

A Longman Group Company

First published 1986

British Library Cataloguing in Publication Data

Bland, Graham
MSX programming.
1. MSX microcomputers – Programming 2. MSX
BASIC (Computer program language)
I. Title
001.64'24 QA76.8.M8

ISBN 0-273-02302-0

Printed in Great Britain at the Bath Press, Avon

Contents

Preface

This book is intended to teach the fundamentals of MSX–BASIC programming. It is assumed that the reader has some degree of familiarity with his computer — using the various functions of the keyboard, for example.

The program examples are included to demonstrate features of the language as they are introduced. Wherever possible, these examples are made to be useful in their own right. For example, Chapter 5 introduces a function and shows how it may be used to centre text on a display.

Readers may note a distinct bias towards graphics, with two major chapters devoted to the topic. Graphics are possibly the most interesting and dynamic feature of home or any computers, and are extremely well supported by MSX–BASIC. The advanced graphics chapter, in particular, was included to correct the MSX manufacturer's poor, or simply non-existent documentation relating to the use of Video Memory and the Video Display Processor.

No attempt has been made to broach the subject of machine code programming. Given the scope of this topic and the fact that this is mainly a book about BASIC programming, I feel that this omission is justified. However, the mechanism for including machine code routines in MSX–BASIC programs is discussed in the Appendices.

MSX–BASIC is an ideal introduction to the BASIC language on a number of computers. A glance at BASICA of the IBM Personal Computer shows that MSX–BASIC is virtually identical to the language of the 16-bit machine. A sound knowledge of MSX–BASIC will allow the interested programmer to move on to programming GW–BASIC 2.0, which is offered by a number of 16-bit computer manufacturers including: Compaq, Digital Equipment Corporation, Hitachi, Olivetti, Sanyo, Wang, and a host of others worldwide.

Graham Bland
October 1985

Acknowledgements

In the preparation of this book, I am indebted to Janet Morrison, who provided useful criticism and technical advice.

This book is dedicated to young Emma Louise Fenwick, who is as yet blissfully unaware of the existence of computers and, with any luck, should remain that way for a few years yet.

Graham Bland
October 1985

Syntax notes

The following notation is used to represent MSX–BASIC commands and functions throughout the book:

Keywords are indicated in upper case and must be supplied.
Items shown between the symbols ⟨ ⟩ must be supplied.
Items shown in square brackets [ ] are optional.
Items separated by ¦ symbols require only one item of the list to be supplied.
Repeated items are shown by . . .

All other punctuation must be included where shown.

1 First principles

A computer is simply a versatile piece of electronics designed to solve problems. These problems can be as diverse as the production of a game and advanced calculus. The computer can accomplish such a wide range of tasks by virtue of its ability to recognize a set of instructions and carry them out. A list of computer instructions is termed a *program.*

Contrary to popular belief, computers are not super-intelligent, nor are they stupid. They are simply machines which carry out instructions to the letter. The supplier of the instructions, the programmer, has to communicate the means of performing a given task clearly and concisely. The MSX–BASIC language has been designed to ease the task of communication between man and machine.

This chapter introduces MSX–BASIC and the elements of an MSX Computer. To begin with, there is an overview of the main components of the computer system.

The central processing unit (CPU)

The *central processing unit* is the keystone of any computer system. It can read a list of instructions and carry them out. The instructions it obeys are a series of special numbers which comprise *machine code.*

The CPU is responsible for all calculations and the control and monitoring of most of the elements of the computer. In all MSX computers, the CPU is a microprocessor called the **Zilog Z8Ø.**

Memory

Memory is used to store information. This information may be a list of instructions that constitute a program, or information that is to be processed by programs. There are two main types of memory that need to be considered.

1 **Random Access Memory (RAM)** is termed 'volatile', as it may only retain information when supplied with power.
2 **Read Only Memory (ROM)** may only be read and not written to. It stores information (generally programs) which have previously been 'burned' into it. Information is retained with or without the presence of a power source.

Input and output devices

Input devices are used to supply the CPU with information. The most important of these is undoubtedly the keyboard. Joysticks, lightpens and touchpads are further examples of this type of device.

For output, a **monitor** is virtually essential. This may be a domestic television set, or a custom-built dedicated device. A **printer** is another form of output device.

Some pieces of equipment may act as both source and recipient of information. The cheapest and most popular input/output device is the cassette recorder. A faster, but costlier alternative is a **disk** storage unit.

Bits and bytes

The most basic element of storage in a digital computer is a **bit**, which is an abbreviation for the phrase **b**inary digi**t**. A bit may have one of two values: either '0' or '1'. A group of eight bits, called a byte, is used to represent information. The Z80 is termed an 8-bit processor as it may only read or write values in groups of eight bits at a time.

Using a byte, up to 256 distinct numbers (from 0–255) may be represented. This is achieved using the binary number system. Unlike our more familiar decimal system, binary represents values in base 2. Instead of having units, tens, hundreds, etc., binary has units, twos, fours, eights, sixteens, etc. Here are a few examples of binary values:

```
100      =   4 (decimal)
11110001 = 241 (decimal)
1101     =  13 (decimal)
```

A byte is not the only collective name for a group of bits. The following names are also commonly used for different groups:

Nibble	4 bits
Byte	8 bits
Word	16 bits
Double word	32 bits

Memory addressing

Your computer will be advertised as having 64K, 48K, 32K or perhaps 16K of memory. The 'K' represents the number 2^{10} which is 1024. So if you have a 64K computer, the amount of memory will be:

$$1024 \times 64 = 65536 \text{ bytes.}$$

Each byte of memory is regarded as a unique entity which may be referenced by name. This 'name' takes the form of a number. The Nth memory location is referred to by the number N−1. So the 65536th memory location is given by the number 65535.

The MSX–BASIC interpreter

Every MSX computer contains 32K of ROM. This stores a special machine code program called the **MSX Interpreter.** As soon as the computer is switched on, the CPU reads and executes (carries out) the instructions of the interpreter.

The first thing the interpreter does is to produce the familiar 'copyright message'. It also performs a series of tests on the computer. The first visible result of these tests is a proclamation of the amount of RAM free for your programs and data. This amount varies, but for most systems (32–64K) it will be 28815 bytes.

The major function of the interpreter is to allow you to enter, modify and execute BASIC programs. BASIC programs are translated into machine code so that they may be understood and executed by the CPU.

The 'Ok' prompt shows that the interpreter is ready and waiting to accept instructions. This is known as being in **command** or **direct mode.** Instructions that may be given in this mode are numerous, but are usually concerned with the entering and modifying of programs.

Try entering the BASIC command CLS (followed by pressing RETURN). If all goes well, the screen should blank and the 'Ok' prompt appear. CLS is a direct command — the interpreter recognizes the phrase CLS (CLear Screen) and acts upon in immediately. Now try entering the phrase CLD. The interpreter should display the phrase 'Syntax error'. CLD is not a phrase that the interpreter can recognize in its rather restricted vocabulary. This syntax error message is probably the one you'll encounter most frequently while programming.

A direct command may only be carried out once without retyping the command.

Programs

Program 1.1 converts lengths supplied in inches to their equivalent length expressed in metres. Type in the program and RUN it. The command RUN tells the interpreter to start translating and executing a program.

Program 1.1

```
10 REM *
20 REM * Conversion Program
30 REM *
40 CLS
50 PRINT "Input Length in Inches"
60 INPUT INCHES
70 METRES = INCHES * .0254
80 PRINT "Length in metres is approx"
90 PRINT METRES
100 END
```

The program should respond with a request for a length in inches and then print out the metric equivalent. We can now make some general observations about programs, and introduce some of the statements and supervisory commands of BASIC.

Line numbers

Each line of a program must begin with a number. This identifies a BASIC statement as being part of a program rather than a direct command. Line numbers must be whole numbers in the range Ø–65529. Normally these are given in increments of ten.

The interpreter executes program lines in sequence. If you type in the program lines in reverse order, the interpreter will rearrange the statements into ascending order of line number. Should line numbers be duplicated, the last line with that number to be typed in will overwrite any others, and will be stored as part of the program.

Multiple statements are allowed on one line — each being separated by a colon. The simple program:

```
10 INPUT INCHES
20 PRINT METRE
```

could be written in the following way:

```
10 INPUT INCHES : PRINT METRE
```

Remarks

The statement REM indicates that the rest of the program line is not to be translated and executed. REM allows useful comments to be placed throughout your programs where appropriate. The character ' ' may be used instead of REM.

PRINT

The PRINT command is used to display information on the monitor screen. In Program 1.1 the phrase:

"Input Length in inches"

is displayed. Anything placed between double quote marks is termed a **string.** Try the following example:

```
10 PRINT "HELLO";
20 PRINT "HELLO"
```

The result should look like this after the program is run:

HELLOHELLO

Normally, after a piece of information has been printed, the next item to be printed will be displayed on the next line down. The semicolon suppresses this feature, so that following information is printed on the same line.

A comma after the item to be printed has a different effect. The statement:

PRINT "HELLO", "HELLO"

prints the two HELLOs separated by 14 spaces.

A question mark is a shorthand way of writing PRINT. The interpreter will convert the question mark to the word PRINT as soon as a program line is entered.

Stopping programs

The END statement indicates the end of a program. This statement need not be included as MSX–BASIC will run a program until it runs out of program lines to translate and execute.

The STOP statement stops program execution dead in its tracks. Program execution may be continued from the line after the STOP statement by typing CONT (continue).

Pressing the STOP key will suspend program execution until the key is pressed again. Pressing both the CTRL and STOP keys simultaneously stops program execution altogether. This is a useful fail-safe mechanism should your program get out of control for any reason.

Displaying programs

The command LIST displays the program in memory on the screen. LLIST lists the program on a printer — if you have one. The following are examples of the different ways that these commands may be used.

LIST .	List the current line.
LIST –40	List all the lines up to and including line 40.
LIST 40–	List all lines from line 40 to the end of the program.

Deleting program lines

If just a line number is typed in, when a statement with that number already exists in memory, then that statement will be deleted. The DELETE command deletes blocks of lines as follows:

DELETE 100	Delete line 100.
DELETE –100	Delete all lines up to and including line 100.
DELETE 50–100	Delete lines 50 to 100 inclusive.

The command NEW erases all the lines of the program in memory.

Renumbering programs

The command RENUM renumbers program lines. The syntax of the command is:

RENUM [[⟨new number⟩][,⟨ old number⟩][,⟨increment⟩]]

(see page vii for syntax notes). Examples of its use are given below:

RENUM	Renumber the program lines in increments of 10 starting at line 10.
RENUM 40	Renumber the program lines in increments of 10 starting at line 40.

RENUM ,,5 Renumber all lines in increments of 5 starting from line 10.

Automatic line numbering

By using the AUTO command when typing in programs, the interpreter automatically provides you with line numbers. Every time a program line is entered, a new line number is produced for the next line. The command:

AUTO 100,5

produces line numbers starting at line 100, with increments of 5.

Altering the screen display

There are two screens that you may use for text applications. The default screen allows a maximum of 24 lines with 40 characters (maximum) per line; the second screen also has 24 lines but a maximum of only 32 characters per line.

Further flexibility is provided by the WIDTH command, which clears the screen and resets the maximum number of characters per line as specified. For example, the command:

WIDTH 20

allows a maximum of 20 characters per line.

The function key display at the bottom of the screen may be turned off by using the KEY OFF command, and turned back on again using KEY ON. Disabling the function key display frees the 24th line of the display for use by programs.

The SCREEN command

There are four screens that may be used by MSX–BASIC. Two are concerned with graphics displays and are dealt with in Chapter 8. Selecting a screen mode is controlled by the command SCREEN followed by a screen mode number. The mode numbers are as follows:

0 40 × 24 text screen
1 32 × 24 text screen

2 High-resolution graphics screen
3 Low-resolution graphics

The full syntax of the command is given by:

SCREEN ⟨mode⟩ [,⟨sprite size⟩][,⟨key click⟩][,⟨baud rate⟩][,⟨printer option⟩]

The ⟨key click⟩ option may be used to turn the key click off or on.

```
SCREEN ,,Ø      (turns the click off)
SCREEN ,,1      (turns the click on)
```

Altering colour

The colour of the text and background (and the border) of the screen may be changed using the COLOR command. The range of colours available is as follows:

Ø Transparent
1 Black
2 Medium green
3 Light green
4 Dark blue
5 Light blue
6 Dark red
7 Cyan
8 Medium red
9 Light red
1Ø Dark yellow
11 Light yellow
12 Dark green
13 Magenta
14 Grey
15 White

The colour command has the following syntax:

COLOR ⟨foreground⟩,⟨background⟩,⟨border⟩

The command:

```
COLOR 15,1
```

sets the text colour to white and the background to black.

The function keys

A string of up to fifteen characters may be assigned to each of the functions keys. If we wish to set function key F1 to produce HELLO, and F10 to produce MSX, we use the following commands:

```
KEY 1, "HELLO"
KEY 10, "MSX"
```

The current contents of all the function keys may be displayed by typing KEY LIST.

Saving and loading programs

A program may be saved to tape in one of two ways. CSAVE saves the program in **tokenized** format. This is the fastest method. To see if a program has been saved correctly, the tape may be rewound and the command CLOAD? used.

CLOAD? compares the program it reads from tape with the program currently in memory to see if they match. CLOAD loads a program back from tape into memory.

SAVE saves a program in ASCII format. This is slower than CSAVE, but has advantages. A program may be loaded from tape and run automatically if it has been SAVEd.

LOAD reads a named program from tape. If the following command is used:

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

the first ASCII program encountered on the tape will be loaded and run automatically. The command:

```
RUN "CAS:"
```

has the same effect.

"CAS:" describes the cassette recorder and is an example of a **device descriptor** (see Chapter 6). "LPT:" describes the printer. If a program is SAVEd with the name "LPT:", the effect will be the same as giving the command LLIST.

Altering the baud rate

Programs are normally saved to tape at a rate of 1200 bits per second. This is called the **baud rate**. If you have a very good tape recorder, you

may double the speed at which programs are saved by setting the <baud rate> option of the SCREEN command to 2:

```
SCREEN ,,,2
```

Programs will then be saved at 2400 baud.

The CSAVE command may also be used to save a program at the higher baud rate. To save a program called "FRED" at 2400 baud, you would use the following command:

```
CSAVE "FRED",2
```

This concludes this brief look at a number of the 'bread and butter' BASIC commands, many of which have been concerned with the *manipulation* of programs rather than programming. Chapter 2 introduces some programming concepts.

Summary

CLS

COLOR ⟨foreground⟩, ⟨background⟩,⟨border⟩

WIDTH ⟨screen width⟩

SCREEN ⟨mode⟩[⟨sprite size⟩][,⟨key click⟩][,⟨baud rate⟩][,⟨printer option⟩]

REM

PRINT [⟨expression⟩]⟨separator⟩[⟨expression⟩]. . .

END

STOP

CONT

RUN [⟨line number⟩]

RUN "⟨program name⟩"

LIST [⟨first number⟩]–[⟨last number⟩]

LLIST [⟨first number⟩]–[⟨last number⟩]

In both the above, a full stop indicates the current line.

DELETE [⟨first number⟩]–⟨last number⟩

RENUMBER [[⟨new number⟩][,⟨old number⟩][,⟨increment⟩]]

AUTO [⟨line number⟩][,⟨increment⟩]

KEY ⟨function key number⟩,⟨string expression⟩

KEY ON ¦ OFF ¦ LIST

CSAVE [⟨program name⟩][,⟨ 1 ¦ 2 ⟩]

CLOAD?

CLOAD [⟨program name⟩]

SAVE ⟨program name⟩

LOAD ⟨program name⟩[,R]

2 Elementary MSX–BASIC

This chapter introduces a number of important programming concepts: data and data types, operators and conditional statements. An understanding of all the above is essential before any serious programming is undertaken.

Data and data types

Computer programs require information with which to work. The information that the computer uses is termed **data**.

Program data may originate from two main sources. Information may be supplied from the keyboard (via commands like INPUT), cassette, joystick or any one of a host of other input devices. Alternatively, data may already be present in the computer's memory, usually as part of the program text or as the result of some previous processing.

Data may be divided into distinct classes. Some information used will remain the same under any circumstances. The number of centimetres in a metre, the length of the phrase "MSX" and the acceleration due to gravity are all examples of **constants**.

Values such as the daily temperature, the length of a word taken at random from a dictionary or the Financial Times Share Index are quantities which change and cannot be assigned constant values. Such data is represented by **variables**.

An item of data may also be one of a number of different **data types**. The range of data types available varies between constants and variables. As constants have the most comprehensive selection, here is the complete list of MSX–BASIC data types.

1. Integer (whole numbers)
2. Real (numbers with a decimal point)
3. Strings (character data)
4. Binary (numbers written in $base_2$)
5. Hexadecimal (numbers written in $base_{16}$)
6. Octal (numbers written in $base_8$)

The next section details how these data types are declared for constants and, where applicable, variables, and the particular merits and drawbacks of using them.

Integers

An integer constant is a whole number written *without* a decimal point. Such constants are expressed by terminating the number with a per cent (%) sign. The permissible range of integer values is –32768 to 32767. Integer variables may also be used, and these, too, are declared using a trailing per cent sign.

Integer values are often used in applications such as counting, where fractions are unnecessary and undesirable. Their main disadvantage lies in the fact that the range of numbers they can represent is relatively small. However, storage of an integer value requires only two bytes of memory.

Real numbers

This is by far the most useful number type, mainly because fractions are possible. Also, there is a wide range of numbers that may be represented:

$9.9999999999999 \times 10^{62}$	is the largest value.
10^{-63}	is the smallest value.

There are differing levels of **precision** evident in MSX–BASIC's real numbers, which affect the amount of memory required to store the numbers:

1. **Single precision** implies that the six most significant digits of a constant or variable are to be stored. This requires four bytes of memory per value.
2. **Double precision** allows an accuracy of up to 14 significant digits. Consequently, a greater amount of memory is required for the storage of a double precision value — a total of eight bytes in fact.

There are also two different ways of representing real numbers. The first method, known as **fixed point** representation, is the one most people are used to. Examples of numbers written in this way are:

1692.34 Ø.ØØØ729 −36 −511.Ø5

An alternative is **scientific notation**. This is particularly useful when very large or very small numbers are to be used. Examples of this

notation are:

1.645E−6 9.681E−24 6.823392394D−35 −23.545D5Ø

where both E and D are equivalent to 'times 1Ø to the power of'. The D indicates that the number is stored with double precision.

Single precision numeric constants and variables are recognized by the trailing symbols ! for single precision and # for double precision.

Double precision values and variables are set by default in MSX–BASIC.

Strings

Any characters which are included within a pair of double quotation marks form a **string constant**. The characters used may be letters, numbers, spaces or any other characters which may be printed. A string constant is defined as *every character* between the quotation marks. For example:

"MSX Computers"

is not the same as:

" MSX Computers "

by virtue of the extra spaces present in the latter string.

A character that may not be included in a string constant is the double quotation mark itself. The string:

"They all shouted "Hoorah!" in unison"

will produce an error. Unfortunately, there is no way around this problem using string constants.

Another limitation in the use of string constants is their length — a maximum of 255 characters per string is allowed. MSX–BASIC imposes an initial limit of 2ØØ characters of storage space for all strings by default. The following are examples of valid string constants.

"Biggles is a Pilot" "4634.377" "2 by 2"

String variables are identified by a trailing dollar ($) symbol.

Binary constants

Numbers which are valid integers may be shown in binary format. A binary constant is composed of a series of binary digits prefixed with the characters &B. Examples of positive binary constants are:

&B11111111 $= 255_{1Ø}$

&B1111 = 15_{10}
&B0111111111111111 = 32767_{10}

The simplest way of producing a binary value from a positive decimal integer is by repeated division by 2. The method of conversion is shown for the number 137.

1. Divide 137 by 2 = 68 with remainder 1
2. Divide 68 by 2 = 34 with remainder 0
3. Divide 34 by 2 = 17 with remainder 0
4. Divide 17 by 2 = 8 with remainder 1
5. Divide 8 by 2 = 4 with remainder 0
6. Divide 4 by 2 = 2 with remainder 0
7. Divide 2 by 2 = 1 with remainder 0
8. Divide 1 by 2 = 0 with remainder 1

When the result of the last division is zero, the binary number can be assembled from all the *remainders*. The least significant (rightmost) bit being the remainder from the first division, through to the most significant bit as the final remainder. The binary equivalent of 137 (decimal) is therefore 10001001.

Negative binary numbers are not so easily included as constants. MSX–BASIC stores negative numbers in their two's complement format. To convert a negative number to its two's complement form:

1. Take the binary representation of the positive value.
2. Change all the ones to zeros and all the zeros to ones.
3. Finally, add one to this new binary number.

It is important that a full 16-bit representation of the original number is used, so remember to include *all* leading zeros. Here are the steps to creating the two's complement representation of the number –15:

1. &B0000000000001111 = 15_{10}
2. &B1111111111110000 After inverting all the bits.
3. &B1111111111110001 = -15_{10}

Here are a further three examples for you to mull over:

1. &B1000000000000001 = -32767_{10}
2. &B1111111100000001 = -255_{10}
3. &B1111111111111111 = -1_{10}

Any binary number that has the 16th bit position (2^{15}) set to a '1' is automatically assumed to be negative in MSX–BASIC. This bit is therefore termed the **sign bit**.

Binary constants are of most use when writing to hardware such as the sound chip, where the positions of specific bits have a special significance.

Hexadecimal constants

Any valid integer value can be represented as a hexadecimal (hex) constant. Numbers are written to the base$_{16}$ where the number 16_{10} would be written as 10_{16}. Decimal values of 10–15 are written using the letters A–F. Hexadecimal constants are preceded with the characters &H and valid examples are:

&HFF (255) &H20 (32) &HFFFF (−1)

Negative numbers are represented in two's complement form.

There is a simple method for converting binary values to hexadecimal values. The method is given as follows:

1. Starting from the right of the number, divide the number into groups of four bits.
2. Convert each group of four bits into its decimal equivalent. If any of these numbers is equal to or greater than 10 then convert it to the corresponding letter (A–F).

Taking the decimal number 449 as an example, once in binary format, the conversion process can be seen as follows:

1.	&B0001110000001	Original binary constant.
2.	0001 1100 0001	After dividing into 'nibbles'.
3.	1 12 1	After converting into decimal.
4.	&H1C1	Final hexadecimal constant.

Hex is mainly the province of the machine code programmer as it has the virtue of being quite a compact way of expressing binary values.

Octal constants

These are numbers to the base eight. Their inclusion in this language is a bit of an oddity as most people will find that hex and binary are all they'll ever need to use. They are included for completeness.

An octal constant is prefixed with the characters &O, and the same restrictions apply here as to hex and binary constants.

Conversion of binary numbers into octal follows much the same method as conversion into hexadecimal, but instead of dividing the binary value into groups of four bits, it is divided into groups of three bits. Here the binary value for 449 is converted to octal.

1. &BØØØ111ØØØØØ1 Original binary constant.
2. ØØØ 111 ØØØ ØØ1 After dividing into three-bit groups.
3. Ø 7 Ø 1 After converting into decimal.
4. &OØ7Ø1 Final octal constant.

Octal constants conclude the range of data types supported by MSX–BASIC.

Naming variables

There are only four types of variables in MSX–BASIC: real, integer, string and array. The array variable will be covered in some detail in Chapter 6.

A variable is simply a reference to an area of memory where a value of any of the above data types may be stored. A variable which is declared as a double precision real variable may only hold double precision real numbers, a string variable can only hold string data, and so forth.

Each variable must have a unique name. All variable names *must* start with a letter followed by up to 252 characters if desired. The last character may be one of the **type declaration characters** — #, !, %, or $. It should be noted that only the first two characters of a variable name are significant — the variable names 'VOLKSWAGEN' and 'VOLUME' would both be 'VO' to MSX–BASIC.

No variable name may contain the *reserved* words that MSX–BASIC uses itself. PRINT is not allowed as a name, nor is PREMIUM (as REM is a reserved word). Some variable names may raise objections without any apparent reason. LOCATION is one such example as it contains the reserved word LOC. LOC is not yet used by MSX–BASIC, but it is a function that anticipates the arrival of MSX disk drives. A list of MSX–BASIC reserved words which are not yet interpreted is included in Appendix 3.

The MSX–BASIC operators

As you might expect, MSX–BASIC provides a set of operators to perform arithmetic. These operators are shown in Table 2.1

Table 2.1

Operator	Example	Meaning
+	X + Y	Add X and Y
−	X − Y	Subtract Y from X
MOD	X MOD Y	Produce integer result of X divided by Y
*	X * Y	Multiply X by Y
/	X / Y	Divide X by Y
^	X ^ Y	Calculate X to the power Y

An additional operator that can affect numbers is the **unary minus** (unary as it only requires one operand). This operator negates a number: if A=10 and the expression A=−A is used, then A will assume the value of −10.

Strings have only one operator — **concatenation** (joining) which is indicated by the '+' sign. For example:

```
"Wombats " + "are go"
```

produces the string:

```
"Wombats are go"
```

An **expression** is any sequence of operators, constants, variables and parentheses (round brackets). These range from the simple:

```
A + B − C − 12
```

to a more complex expression like:

```
(A * −(B) / MOD 2) ^ (2−1)
```

Note the use of brackets as in algebra. When encountered as a list, operators are normally evaluated in this order: exponentiation, negation, multiplication and division, modulo arithmetic (MOD), addition and subtraction. If there are several appearances of the same operator, evaluation is carried out from left to right. Using brackets overrides the normal order of precedence of the operators.

Variable assignment

The simplest method of giving a variable its value is by using the **assignment statement**. This has one of two forms.

```
LET ⟨variable name⟩ = ⟨expression⟩
```

or:

```
⟨variable name⟩ = ⟨expression⟩
```

Of the two forms, the latter is shorter and more convenient. The '=' sign is termed the **assignment operator**. The variable to the left of the operator will assume the value of the expression to the right.

An expression may be a simple variable or a mathematical expression. Program 2.1 shows a series of ways in which assignment may be carried out.

Program 2.1

```
10 A = 2
20 B = 10
30 PRINT "A = ";A
40 A = A^B
50 PRINT "A = ";A
```

The INPUT statement is another means of assigning values to a variable. A simple example of the input statement is:

```
INPUT X
```

The computer prints a question mark and waits for a number to be typed in and the RETURN key to be pressed. The variable X will then contain the value of the number that was typed in. The statement:

```
INPUT X,Y,Z
```

expects three numbers to be typed in, separated by commas. If you give the computer only one number, it will print out a double question mark indicating that it expects some more information.

A string expression may also be included in the INPUT statement. It is permissible to have the following:

```
INPUT "Type in a number please ";X
```

The statement results in the string being printed followed by a question mark. The computer is then expecting a value to be supplied to put into X. The string used is known as a **prompt string**.

Assignment using READ and DATA

The READ statement is used to take constants from a DATA list and assign them to variables. As an example, the variables A, B and C will assume the values −45, 54Ø, and 1.7 when Program 2.2 is run:

Program 2.2

```
10 REM *
20 REM * READ and DATA
```

```
30 REM *
40 READ A
50 READ B
60 READ C
70 PRINT "A = ";A
80 PRINT "B = ";B
90 PRINT "C = ";C
100 DATA -45,540,1.7
```

Strings may also be included in DATA statements thus:

```
DATA Strings,may,be,included
```

Note that double quotation marks are not required for string DATA unless the strings contain spaces:

```
10 DATA "This contains spaces"
```

A command associated with READ and DATA is RESTORE. After using RESTORE, the next DATA item to be READ will be that of the very first DATA statement in the program.

Using RESTORE with a line number, e.g., RESTORE 90, sets the next item of DATA to be READ at or beyond the given line number. The use of RESTORE is demonstrated in Program 2.3.

Program 2.3

```
10 REM *
20 REM * RESTORE
30 REM *
40 RESTORE 130
50 READ A$
60 PRINT A$
70 READ A$
80 PRINT A$
90 RESTORE
100 READ A$
110 PRINT A$
120 DATA "First Data Statement"
130 DATA "Second Data Statement"
140 DATA "Third Data Statement"
```

READ and DATA statements are very useful, particularly when testing programs. A series of DATA statements can be set up to save typing in a list of test data over and over again. Data statements may occur anywhere within a program.

The SWAP command

As its name suggests, SWAP swaps two values which are stored as variables. Two variables may only be SWAPped if both have previously been assigned values. The command looks like this:

SWAP A,B

Naturally, you can only swap values of the same data type. This command is very useful. To achieve the same effect without using SWAP, requires the following sequence of instructions:

Program 2.4

```
10 A=10:B=5
20 PRINT "A= ";A
30 PRINT "B= ";B
40 C=A
50 A=B
60 B=C
70 PRINT "After swapping"
80 PRINT "A= ";A
90 PRINT "B= ";B
```

The GOTO statement

GOTO alters the order in which program statements are carried out. If the computer encounters the statement:

50 GOTO 100

the program statements numbered from line 100 will be executed next. GOTO is an example of **unconditional branching**.

One use of the GOTO is to create **loops** which allow a sequence of program statements to be repeated over and over again. Program 2.5 demonstrates an **infinite** loop. This program will run for ever unless the CONTROL–STOP key is pressed.

Program 2.5

```
10 PRINT "This Loop is Infinite"
20 GOTO 10
```

The real power of the GOTO statement is realized when combined with a decision — for example, if a sequence of program steps is to be carried out only if a value is less than 10. The next section introduces

an important set of operators and statements which allow a specific action to take place on the basis of a decision.

Conditional statements and branching

Using **conditional branching**, the order in which program statements are executed depends on some **condition**. Consider the following phrase: if a person's age is greater than, or equal to eighteen, then they are eligible to vote. The clue to an equivalent BASIC statement is given in the words IF and THEN.

The IF . . . THEN statement is given by the following:

IF ⟨condition⟩ THEN ⟨statements⟩ ¦ ⟨line number⟩
[ELSE ⟨statements⟩ ¦ ⟨line number⟩]

or:

IF ⟨condition⟩ GOTO ⟨line number⟩
[ELSE ⟨statements⟩ ¦ ⟨line number⟩]

When the ELSE option is excluded, the computer will test the condition to see if it is *true*. If it is, the sequence of instructions after the THEN keyword is carried out, or a branch to the line number occurs. If the condition tested is *false*, the next statement in sequence after the IF . . . THEN statement is executed.

With the ELSE option, if the condition tested is *false*, then the statement after the ELSE keyword will be carried out, or a branch to the appropriate line number will occur. Note that the ELSE keyword must occur on the same program line as the IF . . . THEN statement.

Relational operators

Operators which allow tests to be set up within IF . . . THEN statements are known as **relational operators**. They are shown in Table 2.2.

Returning to the voting age example, we can test for the condition 'greater than or equal to' eighteen by the following statement:

```
30 IF AGE >= 18 THEN PRINT "You may vote."
```

Compound conditional statements

When the outcome of two tests is required to determine what action is to be taken, a compound condition is used. Consider the phrase: if a

Table 2.2

Operator	Test
=	Equivalence (not to be confused with assignment)
<>	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

person's age is greater than or equal to 5, and their age is less than or equal to 16, then the person must attend school.

The person is only obliged to attend school if *both* conditions are obeyed. In BASIC, such a test could be expressed:

```
30 IF AGE>=5 AND AGE <=16 THEN PRINT "You must attend school"
```

The keyword AND is an example of a **logical operator**. If either of the conditions is *false* (e.g., AGE=56 or AGE=1), then the test would fail, and the string would not be printed. Another logical operator is OR. Only if *both* conditions tested are *false* will the test fail.

One way of showing the outcome of a logical operation is to use a **truth table**. In such a table, all the possible outcomes of tests on operands are indicated. If a condition is true then it is indicated as a '1' and if false, by a '0'.

The truth table for AND is shown below:

A	B	A AND B
0	0	0 (false)
1	0	0 (false)
0	1	0 (false)
1	1	1 (true)

NOT is a unary logical operator which sets the outcome of a test to false if it is true and vice versa. For example, in the statement:

```
30 IF AGE>=18 AND NOT (C$="RUSSIA") then print "You may vote"
```

the string will be printed only if C$ is NOT equivalent to "RUSSIA" and AGE>=18.

AND, NOT and OR are the most commonly used logical operators. The complete set is detailed in Table 2.3 and their truth tables are given

in Appendix 4. In common with other operators, they have an order of precedence which may be overridden by the use of brackets.

Table 2.3

Operator	Function
NOT	Inversion
AND	Logical AND
OR	Inclusive OR
XOR	Exclusive OR
EQV	Equivalence
IMP	Implication

Conditional branching is used in Program 2.6. The user is prompted to supply the day and month of his birthday as two integers. June 19th would be given as 6 and 19.

The program then tests that the values given are acceptable (a month of 33 would not be admissible, nor would a day of 46), and then decides what the astrological (birth) sign of the user is and prints out some relevant information as in Figure 2.1.

Program 2.6

```
10 REM ************************
20 REM *                      *
30 REM * Horoscope Program    *
40 REM *                      *
50 REM ************************
60 CLS
70 SCREEN 0 : WIDTH 38
80 REM *
90 REM * Request birth information
100 REM *
110 INPUT "Month of Birthday";M
120 IF M<1 OR M>12 THEN BEEP : GOTO 110
130 INPUT "Day";D
140 IF D<0 OR D>31 THEN BEEP : GOTO 130
150 PRINT
160 IF (M=3 AND D>=21) OR (M=4 AND
D<=19) THEN 310
170 IF (M=4 AND D>=20) OR (M=5 AND
D<=19) THEN 340
180 IF (M=5 AND D>=20) OR (M=6 AND
D<=20) THEN 370
```

```
190 IF (M=6 AND D>=21) OR (M=7 AND
D<=22) THEN 400
200 IF (M=7 AND D>=23) OR (M=8 AND
D<=21) THEN 430
210 IF (M=8 AND D>=22) OR (M=9 AND
D<=22) THEN 460
220 IF (M=9 AND D>=23) OR (M=10 AND
D<=22) THEN 490
230 IF (M=10 AND D>=23) OR (M=11 AND
D<=21) THEN 520
240 IF (M=11 AND D>=22) OR (M=12 AND
D<=21) THEN 550
250 IF (M=12 AND D>=22) OR (M=1 AND
D<=21) THEN 580
260 IF (M=1 AND D>=22) OR (M=2 AND
D<=19) THEN 610
270 IF (M=2 AND D>=20) OR (M=3 AND
D<=20) THEN 640
280 REM *
290 REM * Print out information
300 REM *
310 PRINT "ARIES"
320 S$="The Ram" : P$="Mars"
330 GOTO 660
340 PRINT "TAURUS"
350 S$="The Bull" : P$="Mars"
360 GOTO 660
370 PRINT "GEMINI"
380 S$="The Twins" : P$="Mercury"
390 GOTO 660
400 PRINT "CANCER"
410 S$="The Crab" : P$="Moon"
420 GOTO 660
430 PRINT "LEO"
440 S$="The Lion" : P$="Sun"
450 GOTO 660
460 PRINT "VIRGO"
470 S$="The Maiden" : P$="Mercury"
480 GOTO 660
490 PRINT "LIBRA"
500 S$="The Scales" : P$="Venus"
510 GOTO 660
520 PRINT "SCORPIO"
```

```
530 S$="The Scorpion" : P$="Mars"
540 GOTO 660
550 PRINT "SAGITARIUS"
560 S$="The Archer" : P$="Jupiter"
570 GOTO 660
580 PRINT "CAPRICORN"
590 S$="The Goat" : P$="Saturn"
600 GOTO 660
610 PRINT "AQUARIUS"
620 S$="The Water Bearer" : P$="Saturn"
630 GOTO 660
640 PRINT "PISCES"
650 S$="The Fish" : P$="Jupiter"
660 PRINT
670 PRINT "Symbol        : ";S$
680 PRINT "Ruling Planet: ";P$
```

```
Month of Birthday 6
Day 19

GEMINI

Symbol        : The Twins
Ruling Planet: Mercury
```

Figure 2.1 Sample output from Program 2.6

String comparisons

Relational operators can be used to compare strings. Comparisons such as less than and greater than are not as obvious for strings as they are for numbers. For example:

"ALGY" is less than "algy"

The reason for this is because each character in MSX–BASIC has a code number called its **ASCII** code. The nature of this code is dealt with in Chapter 5. It is these code numbers that are actually compared in MSX–BASIC. If we look at the code for each of the characters in both strings, we see:

"ALGY"	65	76	71	88
"algy"	97	108	103	121

(The comparison is made on the first character of the string, and only continues to the second or subsequent characters if the first characters are the same — ALGY is less than ALGy.)

Loops

Conditional statements can be used to create loops.

The while loop

The **while** loop structure may be set up using IF . . . THEN and GOTO statements. This sort of loop will carry out a sequence of operations *while* a condition is true. Program 2.7 shows a while loop in use.

Program 2.7

```
10 REM *
20 REM * While Loop
30 REM *
40 C=1
50 IF C<0 OR C>9 THEN 100
60 READ NUMBER
70 PRINT NUMBER^2
80 C=C+1
90 GOTO 50
100 END
110 DATA 1,5,12,6,23,54,34,4,8,10,20,100
```

The loop prints out items from the DATA list quite happily as long as the value of C is greater than zero and less than ten. Such a logic structure (as it is termed), may be shown by the flowchart in Figure 2.2.

The important point to notice about this sort of structure is that the test for ending the loop is carried out *before* the sequence of operations is executed. No 'Out of data' error message will be given should the initial number of items be zero.

The repeat loop

Another sort of loop that can be built from conditional statements is the **repeat** structure. Here, a sequence of operations is carried out *until* a

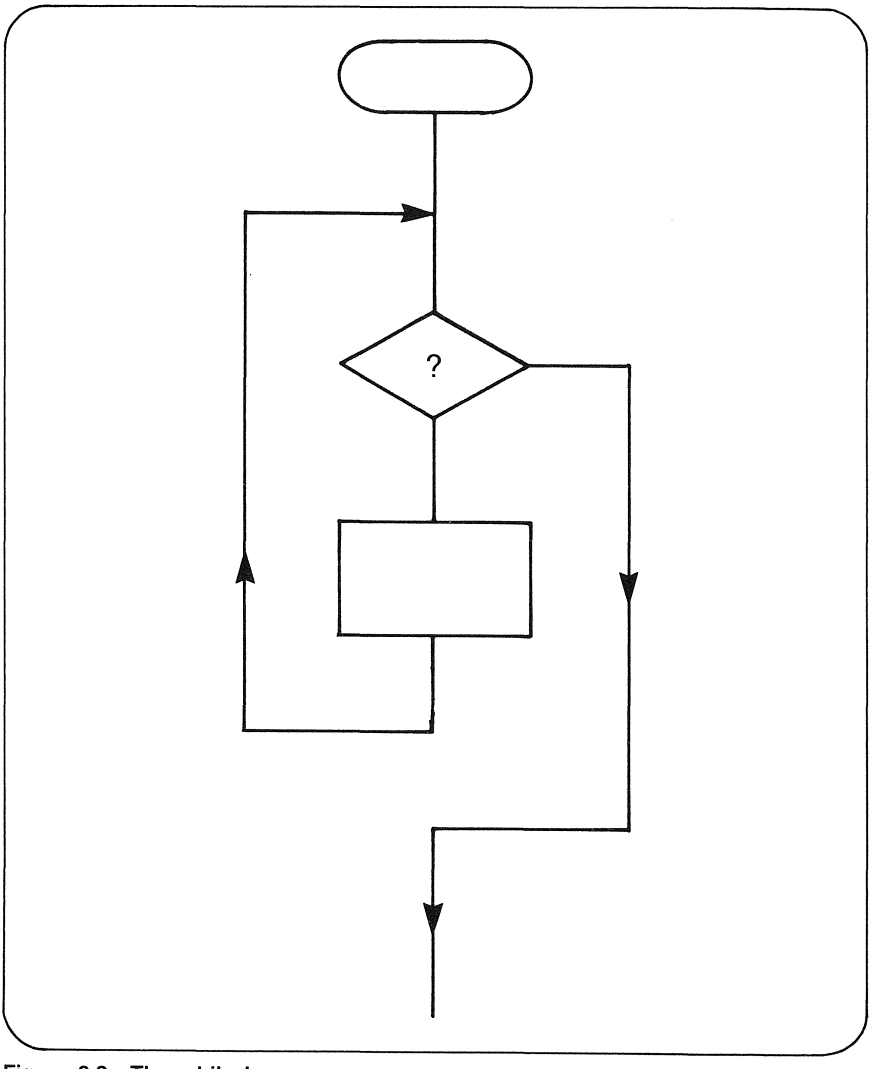


Figure 2.2 The while loop

test is found to be true. Program 2.8 requests input until a value between 0 and 10 is given.

Program 2.8

```
10 REM *
20 REM * Repeat Loop
30 REM *
40 INPUT "Value Between 0 and 10";A
```

```
50 IF A<0 OR A>10 THEN 40
60 PRINT "Correct"
70 END
```

The flowchart for this type of loop is shown in Figure 2.3.

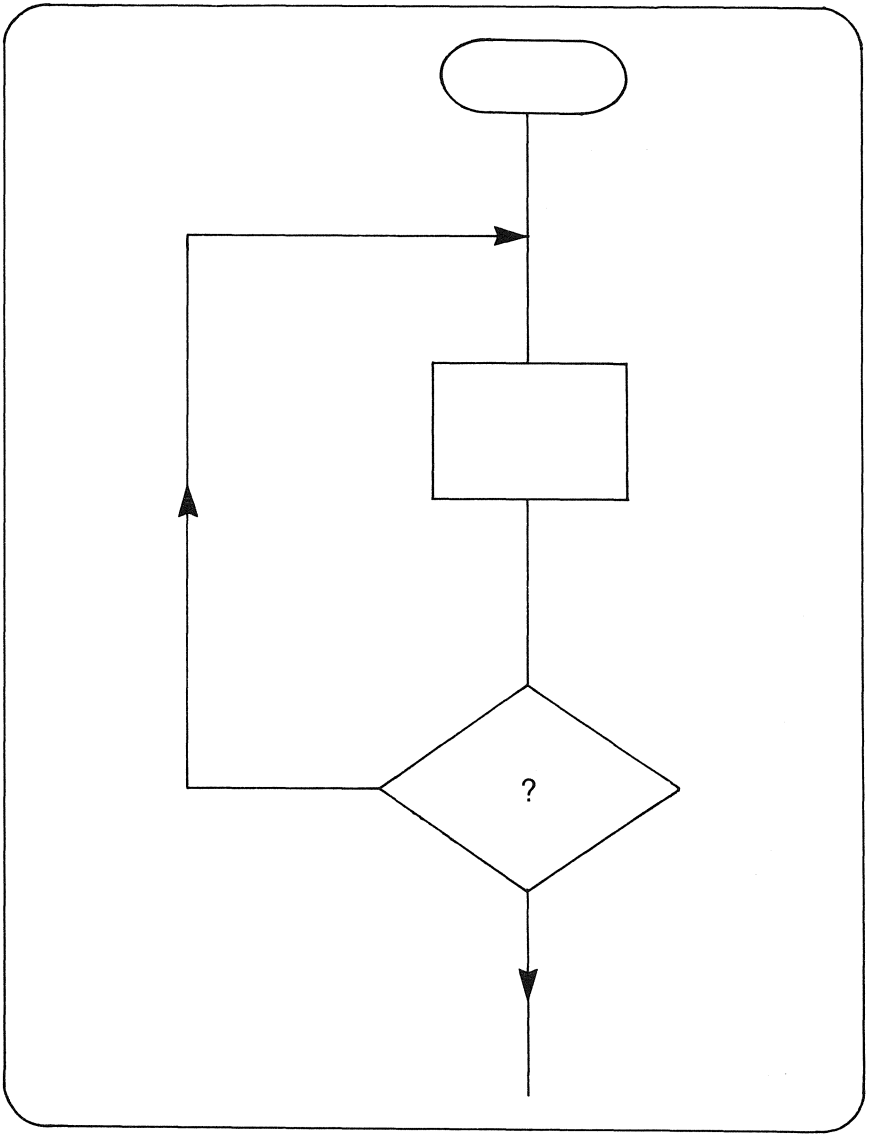


Figure 2.3 The repeat loop

Multi-way branching

The ON . . . GOTO statement causes a branch to any of a number of line numbers, depending on the value of a variable. The format of the instruction is:

ON ⟨variable⟩ GOTO ⟨line number⟩[,⟨line number⟩][,⟨line number⟩]. . .

If the value of the variable is 1, then a branch to the first line number in the list will occur. If the value is 2, then the program will branch to the second line number in the list and so on.

Should the value of the variable being tested exceed the number of line numbers in the list, or should the value be less than one, execution will continue from the next line in sequence.

Program 2.9 is used to calculate areas. One of three shapes may be used by giving a number from 1–3. The ON. . .GOTO statement transfers control to the appropriate section of the program.

Program 2.9

```
10 REM *
20 REM * Area Calculations
30 REM * Using ON GOTO
40 REM *
50 CLS
60 SCREEN 0 : WIDTH 38 : KEY OFF
70 PRINT
80 PRINT "Area Calculations"
90 PRINT
100 PRINT "1. Rectangles"
110 PRINT "2. Rt. Angled Triangles"
120 PRINT "3. Circles"
130 PRINT
140 PRINT "Input Option Number:";
150 INPUT X
160 ON X GOTO 190,280,370
170 BEEP
180 GOTO 50
190 REM *
200 REM * Rectangles
210 REM *
220 CLS
230 INPUT "Input Length of side A";A
240 INPUT "Input Length of side B";B
```

```
250 PRINT
260 AREA = A * B
270  GOTO 480
280 REM *
290 REM * Triangles
300 REM *
310 CLS
320 INPUT "Input Length of Base";B
330 INPUT "Input Height";H
340 PRINT
350 AREA = (B * H)/2
360 GOTO 480
370 REM *
380 REM * Circles
390 REM *
400 CLS
410 INPUT "Radius";R
420 AREA = 3.142 * (R*R)
430 GOTO 480
440 REM *
450 REM * Print Out Results
460 REM *
470 PRINT
480 PRINT "Area = ";AREA;" Sq. Units"
490 PRINT
500 PRINT "Type 1. for another area"
510 INPUT "Any other number to exit";B
520 ON B GOTO 50
530 END
```

Typical output from Program 2.9 is shown in Figure 2.4.

Variable housekeeping

Before a variable is assigned a value in a program, it is initially set to zero in the case of numbers, or to an **empty** or **null** string ("") in the case of strings.

The BASIC command CLEAR has a number of purposes. It sets all numeric variables to zero and all string variables to null strings. In addition, it is used to increase the amount of storage to be allocated to strings. By default, there is enough space for 200 characters. If enough space for 1000 characters is to be set aside, the command:

```
CLEAR 1000
```

```
Area Calculations

1. Rectangles
2. Rt. Angled Triangles
3. Circles

Input Option Number:? 2


Input Length of Base? 13
Input Height? 45

Area =  292.5  Sq. Units
Type 1. for another area
Any other number to exit? 2
Ok
```

Figure 2.4 Sample output from Program 2.9

may be given. A third function of this command is considered in Appendix 6.

You can find out how much of this character space you have left by typing:

```
FRE("")
```

Similarly, the amount of overall memory space left for programs, variables and data may be given by typing:

```
FRE(Ø)
```

which returns the current number of bytes available.

These two commands are examples of **functions** which are detailed in Chapter 3.

Global variable typing

MSX–BASIC initially assumes that all variables are real, double precision numbers, until a type declaration character alters the situation. Four commands allow *every* variable, or a range of variables in a program, to be set to a specific data type. For example, all variables with 'I' as the first letter in their name could be defined as integer variables using the command:

```
DEFINT I
```

Variable names which start with a range of letters can also be defined as one type. For example:

```
DEFINT A–Z
```

sets *all* variables (because variable names must start with a letter) to integer type.

The commands for each data type are:

DEFINT	Set to integers.
DEFSNG	Set to single precision real numbers.
DEFDBL	Set to double precision real numbers.
DEFSTR	Set to strings.

After using any of these commands, the data type may still be set for characters on an individual basis using type declaration characters.

To conclude, this is rather a large, but essential chunk of information to wade through. However, with the commands and statements detailed above, quite complex programming projects can be attempted.

Summary

CLEAR ⟨no. of characters⟩[,⟨end of memory address⟩]

DATA ⟨constant list⟩

DEF/INT/SNG/DBL/STR ⟨range of letters⟩

FRE(0)

FRE("")

GOTO ⟨line number⟩

IF ⟨expression⟩ THEN ⟨statement⟩ ¦ ⟨line number⟩
[ELSE ⟨statements⟩ ¦ ⟨line number⟩]

IF ⟨expression⟩ GOTO ⟨line number⟩
[ELSE ⟨statements⟩ ¦ ⟨line number⟩]

INPUT [⟨prompt string⟩;]⟨list of variables⟩

LET ⟨variable⟩ = ⟨expression⟩

ON ⟨expression⟩ GOTO ⟨list of line numbers⟩

READ ⟨list of variables⟩

RESTORE [⟨line number⟩]

SWAP ⟨variable⟩,⟨variable⟩

3 Functions and subroutines

A great number of programming applications require a specific mathematical expression or sequence of steps to be executed a number of times throughout the program. BASIC provides two convenient shorthand methods by which a sequence of instructions or an expression need be written only once, yet can be used as often as required thereafter. This chapter looks at these two important language features — the **function** and the **subroutine.**

Functions

With functions, a simple expression is evaluated to give a single result. Commonly encountered problems, such as the calculation of square roots, sines, cosines, etc. are provided as part of the MSX–BASIC language. These are known as the **intrinsic** functions, and there are over 4Ø of them available to the programmer. The **user-defined** functions have to be defined by the programmer, and may provide any result desired.

Before going on to use some of the intrinsic mathematical functions, some very basic terminology is required. The data given to a function is known as a **parameter** or **argument**. Functions cannot normally be assigned values like ordinary variables. A parameter may be a variable, constant or a BASIC expression. The value produced by a function is said to be *returned* to a program. BASIC functions will be referred to using the function name followed by brackets; e.g., SIN ( ).

I'll present some of the commonly used mathematical functions here.

Trigonometric functions

MSX–BASIC provides four functions for trigonometry: SIN( ), COS( ), TAN( ), ATN( ) (arctangent). Normally, we would take the expression: X = SIN(45) to mean 'let X become equal to the sine of 45°'. What MSX–BASIC returns is a value of about Ø.85Ø9, which is not quite the value you would expect (Ø.7Ø71). This is because the trigonometric

functions work with *radians* and not degrees. The number of radians in a circle is given as 2 × PI, where PI is approximately 3.142. So one degree is given as 2 × PI / 360 radians.

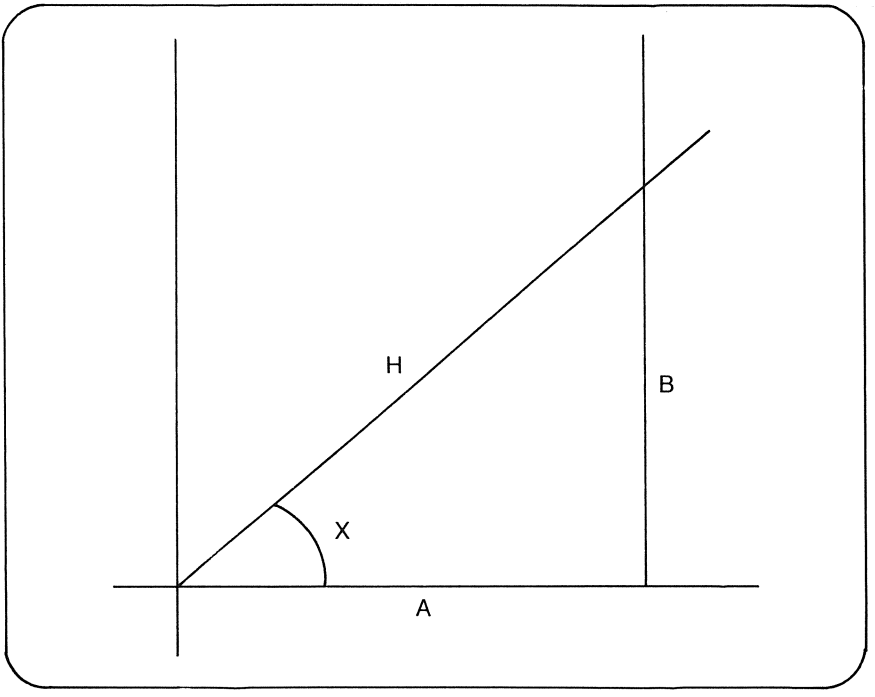


Figure 3.1 Trigonometric relationships

We can define all of these trigonometric rules in terms of the diagram shown in Figure 3.1

Sine of X = B / H
Cosine of X = A / H
Tangent of X = (B / H) / (A / H)

The arctangent is the complementary function of TAN( ). Program 3.1 uses these functions to print out the values of all the trigonometric functions for an angle given either in radians or degrees. Figure 3.2 shows typical results.

Program 3.1

```
10 REM *
20 REM * Trigonometric Functions
30 REM *
```

```
40 SCREEN 0 : KEY OFF
50 PRINT "Degrees (1) or Radians (2) ";
60 INPUT SWITCH%
70 IF SWITCH% < 1 OR SWITCH% > 2 THEN
GOTO 50
80 PRINT
90 PRINT "Input angle ";
100 INPUT A
110 IF SWITCH% = 2 THEN GOTO 160
120 REM *
130 REM * Convert to Radians
140 REM *
150 A = A * 3.14159/180
160 REM *
170 REM * Print Out Trig Functions
180 REM *
190 PRINT
200 PRINT "Sin of angle:";SIN(A)
210 PRINT "Cos of angle:";COS(A)
220 PRINT "Tan of angle:";TAN(A)
230 PRINT "Atn of angle:";ATN(A)
240 PRINT
250 PRINT "Angle =";A;"Radians."
```

```
Degrees (1) or Radians (2)  1

Input angle  36

Sin of angle:  .58778482293256
Cos of angle:  .809017306323
Tan of angle:  .7265417171408Ø
Atn of angle:  .5609817356067

Angle =  .628318  Radians.
Ok
```

Figure 3.2 Trigonometric results from Program 3.1

INT( ) and FIX( )

These two functions are both used to round real numbers to integers, and can be applied in innumerable situations. They both return the

integer part of a real number, but they differ in the way the values are rounded.

INT( ) rounds the number *down* to the next lowest integer. FIX( ) works differently depending on the sign of the number given as a parameter. Negative numbers are rounded up to the next *highest* integer; positive values are rounded down to the next lowest integer. Program 3.2 should help clarify the difference between the two functions.

Program 3.2

```
10 REM *
20 REM *
30 REM * INT() and FIX()
40 SCREEN 0 : KEY OFF
50 C=5
60 READ A
70 PRINT "Initial Value =";A
80 PRINT "FIX()         =";FIX(A)
90 PRINT "INT()         =";INT(A)
100 PRINT
110 C=C-1
120 IF C<>0 THEN GOTO 60
130 DATA 12.453
140 DATA -1932.555
150 DATA -3.21
160 DATA 234.11
170 DATA -0.55
```

Type conversion

Numeric values may change their data type. This conversion alters the amount of memory required to store the value, and may effect some rounding in the process. The major numeric conversion functions are detailed below.

CINT( ) This function converts a real number into an integer by truncating the fractional part. Unlike FIX( ) or INT( ), CINT( ) can only work with real numbers which fall within the range of acceptable integers.

CSNG( ) This produces a single precision number from an integer or real argument. The number is automatically allocated four bytes of storage. Double precision values will be rounded up or down to the nearest sixth significant digit.

CDBL( ) Values are converted to double precision numbers, consequently the storage allocated to the number is increased to eight bytes.

BIN$( ) A string *representation* of a number is produced — the type of the argument is not actually changed. Only integer values may be used with this function.

HEX$( ) As with BIN$( ), a string is returned, this time being a hexadecimal representation of an integer value.

OCT$( ) This works with exactly the same contraints as HEX$( ) and BIN$( ), producing an octal representation of an integer value.

Program 3.3 gives examples of these functions at work.

Program 3.3

```
10 REM *
20 REM * Conversion Functions
30 REM *
40 SCREEN 0 : KEY OFF
50 C=2
60 READ A
70 PRINT "Original Value = ";A
80 PRINT "CINT =";CINT(A)
90 PRINT "CSNG =";CSNG(A)
100 PRINT "CDBL =";CDBL(A)
110 C = C-1
120 IF C <> 0 THEN GOTO 60
130 PRINT
140 C=3
150 READ A
160 PRINT "Original Value = ";A
170 PRINT "Bin$ = ";BIN$(A)
180 PRINT "Hex$ = ";HEX$(A)
190 PRINT "OCT$ = ";OCT$(A)
200 C = C - 1
210 IF C <> 0 THEN GOTO 150
220 DATA 0.434235
230 DATA 152.32331221231
240 DATA 1000
250 DATA 231
260 DATA -12
```

SGN( ) and ABS( )

SGN( ) determines the sign of a number. The function will return one of three values:

```
 Ø  if the argument = Ø
-1  if the argument < Ø.
 1  if the argument > Ø.
```

ABS( ) returns the absolute value of an expression. This may be used to convert negative numbers to positive, or to find the absolute difference between two numbers. For example:

```
ABS(-5.87) = 5.87
ABS(1223-124Ø) = 17
```

Random numbers

Adding an element of chance to programs can be helpful in a number of situations. One prime use of random values is when simulating events in the real world. For example, the number of staff required in a shop at a certain time of day could be predicted by simulating the pattern of customer arrival. Another use for these values is in games, for example, randomizing the arrival of bombs and aliens, or dealing a pack of cards.

The RND( ) function generates a sequence of (pseudo) random values which are dependent on the argument supplied. The way the function performs on given arguments is summarized below.

If the argument is *zero*, the last random number generated is given.
If the argument is *positive*, the next random number in sequence is given.
If the argument is *negative*, a new sequence is given, depending on the value of the argument.

Programs which use positive arguments will always generate the same sequence of numbers each time the program is run. A more random sequence can be generated by **seeding** the RND( ) function. This is carried out by supplying the function with a negative value on a random basis. There are two main ways of doing this. The first is to use a negative value supplied by the program user. You can never be sure what is going to be INPUT each time the program is run, so the sequence of numbers is going to be less predictable. The second method involves the use of a special MSX–BASIC variable.

Every 50th of a second, the video chip generates a signal which is

counted by MSX–BASIC. The current count value is stored in the variable TIME. At any time, this variable has a value between Ø and 65535. As a program may be run at any time the user wishes, the value of TIME cannot be predicated, this making it ideal for use in random seeding.

RND( ) always returns numbers between Ø and 1. To obtain a range of numbers, say between Ø and 1ØØ, we can multiply the value from RND( ) by 1ØØ.

We can apply RND( ) to simulate the roll of two dice. A value between 1 and 6 needs to be produced for each die. This is achieved by multiplying a value returned from RND( ) by 6, rounding it down using INT( ) then adding 1. This neatly gets rid of the value of zero to produce the acceptable range of values for the simulation.

Program 3.4

```
10 REM *
20 REM * Dice
30 REM *
40 REM * Seed RND() Function
50 REM *
60 CLS
70 X = RND(-TIME)
80 D1 = INT(RND(1)*6)+1
90 D2 = INT(RND(1)*6)+1
100 PRINT "Die  1 :";D1;" Die  2 :";D2
110 PRINT
120 PRINT "Press RETURN for next";
130 INPUT X
140 PRINT
150 GOTO 80
```

The possible uses of the TIME variable will be discussed further in Chapter 4.

User-defined functions

MSX–BASIC allows functions with up to 9 parameters to be defined. User-defined functions may always be distinguished from the intrinsic functions by the letters FN which precede their name. Taking a simple example, suppose that a function is to be written which finds the circumference of a circle. The equation that has to be expressed in BASIC is:

$$C = 2 \times PI \times R$$

where C is the circumference and R is the radius of the circle. The function would be written thus (with PI approximated):

```
DEF FNCIRC(R) = 2 * 3.141593 * R
```

The function name is given as FNCIRC. The value in brackets is the parameter to be given to the argument. All items in brackets declare **local variables** to be used in the expression. Local variables only have meaning within a function. A variable named R in the main program will not be affected by the value of R in the function FNCIRC.

There is a one-to-one relationship between the parameters in brackets and their values when the function is used. If a function defined as FNA(A,B,C) = A * B * C is called using the expression X = FNA(1,2,4), the values taken by the local variables are: A = 1, B = 2 and C = 4. Program 3.5 demonstrates the use of FNCIRC, a typical 'answer' is shown in Figure 3.3.

Program 3.5

```
10 REM *
20 REM * Circumference Calculation
30 REM *
40 DEF FNCIRC(R) = 2 * 3.141593# * R
50 CLS
60 INPUT "Radius ";R
70 C=FNCIRC(R)
80 PRINT Circumference = ";C
90 END
```

```
Radius ? 145
Circumference =  911.06197
```

Figure 3.3 Calculation of circumference using Program 3.5

A more complex use of functions is demonstrated in Program 3.6. Two functions are to be applied which will rotate the coordinates of a point by a certain number of degrees. For example, assume that the point given by the coordinates (0,1) is to be rotated about the origin (0,0) by 45°. After rotation, the new coordinates would be given by (0.707,0.707). Similarly, the same point rotated by 90° would give final coordinates of (1,0). For this program, the user is asked to input the X

and Y values for the initial coordinates, and the angle of rotation in degrees. A user-defined function then converts the value supplied in degrees to radians. Another two functions carry out the rotation for the X and Y points. The complete program is shown below. This rotation example will be used as the basis of a future graphics program in Chapter 8.

Program 3.6

```
10 REM *
20 REM * Point Rotation
30 REM *
40 DEF FNC(D)=D*3.141599#/180
50 DEF FNXC(X,Y,R)=X*COS(R)+Y*SIN(R)
60 DEF FNYC(X,Y,R)=-(X*SIN(R)-Y*COS(R))
70 SCREEN 0 : KEY OFF
80 INPUT "Coordinate Pair ";X,Y
90 INPUT "Rotate Angle ";A
100 PRINT
110 R=FNC(A)
120 NX=FNXC(X,Y,R)
130 NY=FNYC(X,Y,R)
140 PRINT "X =";NX
150 PRINT "Y =";NY
160 END
```

It is worth noting that function definitions may include user-defined functions. For example, the sequence:

```
DEF FNA(A) = A/100 * 3
DEF FNB(B) = B ^ (1/2)
DEF FNC(X,Y,Z) = FNA(X) + FNB(Y) * Z
```

although it doesn't achieve much, shows that the inclusion of previously defined functions is perfectly acceptable in MSX–BASIC.

Subroutines

Functions are fine when there is only one value that needs to be calculated. When a more complex sequence of operations is required the function cannot cope, and you need to use subroutines.

Just as the function allows a single expression to be written only once, the subroutine allows a group of program statements to be written

once. Aside from freeing the programmer from needless repetition of program sections, the subroutine has the advantage of better program structuring. A subroutine can be written for each of the logical steps towards a problem solution.

Subroutines are not declared as such, but they may be *referenced.* This reference takes the form of the GOSUB command. It operates similarly to the GOTO command, with one important difference. A GOTO causes a branch to a program line, and execution continues in sequence from that point on; a GOSUB branches to a line number and executes program statements until the command RETURN is encountered. RETURN causes a branch to the program statement immediately following the line containing the GOSUB command. Program 3.7 demonstrates this process.

Program 3.7

```
10 REM *
20 REM * GOSUB Demo
30 REM *
40 SCREEN 0 : KEY OFF
50 GOSUB 130
60 PRINT "Back at Line 60"
70 GOSUB 150
80 PRINT "And back at Line 80"
90 END
100 REM *
110 REM * Subroutines
120 REM *
130 PRINT "Hello from Line 100"
140 RETURN
150 PRINT "Hello from Line 150"
160 PRINT "and 160"
170 PRINT "and 170"
180 PRINT "and 180"
190 RETURN
```

Multiple RETURN statements are permitted in subroutines. These allow a return to the main program to be made depending on a conditional evaluation.

RETURN may be used to return control to any line number in a BASIC program, e.g., RETURN 50. This is not a particularly attractive feature of the language, and really should be avoided unless absolutely necessary. If you have to use RETURN with a line number, it is possibly better to use two GOTO commands to simulate a subroutine.

Subroutines may also be nested, i.e., a subroutine may call another subroutine. Another interesting feature of subroutines is that they may be **recursive** — a subroutine may contain a branch to itself. Consider the following problem. A subroutine is written to check the range of data input to a program. If the data is out of range, then an error signal is given, and more data is requested. An algorithm for this subroutine is presented here. Assume that it is called VALID.

Subroutine VALID
Ask for the user's Age.
If the Age < 16 or the Age > 65 then BEEP and GOSUB VALID.
ELSE the data is valid so RETURN from the subroutine.

This particular use of subroutines is very concise and can be very powerful if used with care.

Stepwise refinement

A complex problem can be analyzed, and gradually broken down into smaller problems, for which program solutions may then be found. This process is known as *stepwise refinement* or *top-down programming.*

In this section, the problem to be broken down is that of a simple payroll. For a set number of employees, the following data is required to be input:

1. The employee's name.
2. The number of hours they have worked in the previous week.
3. Their hourly rate of pay.

The program has to calculate their wages for the week based on the following information:

1. No employee is allowed to work more than 5Ø hours per week.
2. Tax deducted is 3Ø% of the gross pay if more than 4Ø pounds was earned in one week.
3. If over 37 hours of work was carried out, the extra hours are paid at 15Ø% of the employees normal hourly rate.

A basic sequence of steps towards solving this problem could be:

1. Ask how many workers are to be processed.
2. Input one worker's details.

3. Calculate the number of hours of overtime worked (if any).
4. Calculate the pay at overtime rates (if any).
5. Calculate the pay for normal hours worked.
6. Calculate gross pay.
7. Calculate and deduct tax if necessary.
8. Print out all the details.
9. If there are still workers to process, repeat steps 2 through 8.

This is still quite primitive. It is clear that some of the processes can be handled by subroutines. Instead of resolving the problem completely, I'll take one area of the problem and refine it further. It is known that the numbers of hours worked cannot be negative nor more than 50. There is no information on the rate of pay, but it cannot be less than £0.00 per hour. So the algorithm for the input routine can be given as:

> Input a worker's name.
> Input the number of hours worked.
> If the hours worked < 0 or > 50 then signal an error and ask for new data.
> Input rate of pay.
> If the rate of pay < 0 then signal an error and ask for new data.
> Exit the subroutine.

Programming in stages makes program writing and testing far easier. The completed program is shown as Program 3.8. Most of the program is highly modular, making it easy to modify should it be necessary. The subroutines may also be used in other programs at a later date.

Program 3.8

```
10 REM ******************************
20 REM *                            *
30 REM * Simple Payroll Program     *
40 REM *                            *
50 REM ******************************
60 REM *
70 REM * Main Program
80 REM *
90 SCREEN 0 : KEY OFF
100 WIDTH 38
110 PRINT "Type in the number of"
120 PRINT "employees to be processed."
130 INPUT ":";N%
140 IF N% < 0 THEN BEEP : GOTO 130
```

```
150 REM *
160 REM * Main Loop
170 REM *
180 GOSUB 320 : REM * Input Data
190 GOSUB 420 : REM * Overtime
Calculation
200 GOSUB 500 : REM * Tax Calculation
210 GOSUB 590 : REM * Printout Routine
220 N% = N%-1
230 REM *
240 REM * If all workers processed,
finish
250 REM *
260 IF N%=0 THEN PRINT "Finished": END
270 INPUT "Press Return For Next Item";
A$
280 GOTO 180
290 REM *
300 REM * Input Subroutine
310 REM *
320 CLS
330 INPUT "Name                    : ";N$
340 INPUT "Hours Worked            : ";HW
350 IF (HW<0) OR (HW>50) THEN BEEP :
GOTO 340
360 INPUT "Rate of Pay/Hour : ";RP
370 IF (RP<0) THEN BEEP : GOTO 360
380 RETURN
390 REM *
400 REM * Overtime Calculation
410 REM *
420 IF (HW<=37) THEN OH=0:OP=0 : RETURN
430 OH = HW-37
440 HW = 37
450 OP = (1.5 * RP) * OH
460 RETURN
470 REM *
480 REM * Pay and Tax Calculation
490 REM *
500 BP = (HW * RP)
510 GP = BP + OP
520 IF GP<=40 THEN TAX=0:NP=GP:RETURN
530 TAX = GP*.3
540 NP = GP - TAX
```

```
550 RETURN
560 REM *
570 REM * Output All Workers Details
580 REM *
590 CLS
600 PRINT "Final Details"
610 PRINT
620 PRINT "Name                 : ";N$
630 PRINT
640 PRINT "Hours Worked         :";HW
650 PRINT "Rate of Pay          :";RP
660 PRINT
670 PRINT "Basic Pay            :";BP
680 PRINT "Hours at Overtime :";OH
690 PRINT "Overtime Pay         :";OP
700 PRINT
710 PRINT "Gross Pay            :";GP
720 PRINT "Tax due              :";TAX
730 PRINT
740 PRINT "NET PAY              :";NP
750 PRINT
760 RETURN
```

Creating subroutine libraries

Subroutines which occur in a number of programs may be saved on tape and loaded when required. The subroutines should be saved using SAVE rather than CSAVE. When needed, they may be incorporated into programs using the MERGE command. Some caution has to be exercised however. If the program in memory has statements with line numbers that match those of the subroutine to be MERGEd, these lines will be replaced by the corresponding subroutine statements.

Summary

ABS(X), ATN(X), BIN$(X), CDBL(X), CINT(X), COS(X), CSNG(X), FIX(X), HEX$(X), INT(X), OCT$(X), RND(X), SGN(X), SIN(X), TAN(X).

DEF FN ⟨Name⟩[(⟨Parameter list⟩)] = ⟨Function definition⟩

GOSUB ⟨line number⟩

RETURN [⟨line number⟩]

MERGE ⟨file name⟩

4 Loops, interrupts and event monitoring

There are a number of input devices that may be monitored using special MSX–BASIC functions. This chapter introduces some of these commands, but first, a little more about loops . . .

The FOR . . . NEXT loop

Chapter 2 explained how conditional statements could be used to create program loops. MSX–BASIC provides a shorthand method of representing the repeat loop which is made up of two statements:

FOR ⟨loop variable⟩ = ⟨start value⟩ TO ⟨end value⟩ [STEP ⟨increment⟩]

and:

NEXT [⟨variable⟩],[,⟨variable⟩][,⟨variable⟩] . . .

The use of this structure is best shown by example.

Assume we wish to print out the maximum and minimum values of a series of ten lines read from a data statement. The solution is coded first using the IF . . . THEN style repeat loop (Program 4.1) and then by the FOR . . . NEXT equivalent (Program 4.2).

Program 4.1

```
10 REM *
20 REM * MAX and MIN Determination
30 REM * IF ... THEN Loop
40 REM *
50 CLS
60 MX=0 : MN=10^61
70 I = 1
80 REM * Main Loop
90 READ N
100 PRINT I;".",N
110 IF N>MX THEN MX=N
```

```
120 IF N<MN THEN MN=N
130 I = I+1
140 IF I<=10 THEN 90
150 REM *
160 REM * Print Results
170 REM *
180 PRINT
190 PRINT "Maximum Value = ";MX
200 PRINT "Minimum Value = ";MN
210 END
220 DATA 675,43,-4.24,49241,2
230 DATA 12,-1032,999,89.65,34
```

Program 4.2

```
10 REM *
20 REM * MAX and MIN Determination
30 REM * FOR ... NEXT Loop
40 REM *
50 CLS
60 MX=0 : MN=10^61
70 REM * Main Loop
80 FOR I=1 TO 10
90 READ N
100 PRINT I;".",N
110 IF N>MX THEN MX=N
120 IF N<MN THEN MN=N
130 NEXT I
140 REM *
150 REM * Print Results
160 REM *
170 PRINT
180 PRINT "Maximum Value = ";MX
190 PRINT "Minimum Value = ";MN
200 END
210 DATA 675,43,-4.24,49241,2
220 DATA 12,-1032,999,89.65,34
```

In both programs the variable I is used to count how many times the sequence of steps has been performed. In Program 4.2 the statement:

NEXT I

is equivalent to the statement:

I = I + 1

in Program 4.1. The FOR statement sets the initial value of I and also tests the value of I to see if its current value has exceeded the value given by ⟨end value⟩, in this case 10.

This loop notation is much more compact and easier to read than an IF . . . THEN structured loop. The variable name in the NEXT statement can be omitted, but for the sake of clarity it is usually better to include it.

Program 4.2 uses the FOR loop as a simpler counter. The loop variable may also be used *within* the loop.

A factorial is given by the following expression:

$$n! \text{ (n factorial)} = 1 \times 2 \times 3 \cdots \times n$$

Examples are:

$$3! = 1 \times 2 \times 3$$
$$= 6$$

$$8! = 1 \times 2 \times 3 \times 4 \cdots \times 8$$
$$= 40320$$

Using a FOR . . . NEXT loop, a factorial for any number may be produced by keeping a running total which is multiplied by the current value of I at each pass through the loop. Program 4.3 and Figure 4.1 illustrate this process. The program soon reaches the maximum numeric value permitted in MSX–BASIC, and selection of a large value for N results in the 'Overflow' error message.

Program 4.3

```
10 REM *
20 REM * Factorials
30 REM *
40 CLS
50 SUM = 1
60 INPUT "Value ";N
70 FOR I = 1 TO N
80 SUM = SUM * I
90 NEXT
100 PRINT
110 PRINT N;"! = ";SUM
120 END
```

```
Value ? 12

 12 ! =  479001600
```

Figure 4.1 Program 4.3 calculates factorials

Use of the STEP option

When the interpreter encounters a NEXT statement, the loop variable is normally incremented by 1. By using the keyword STEP in a FOR statement, the loop variable may be incremented (or decremented) by any value you choose. Program 4.4 uses the STEP option to print the radian measure through 360° at 45° intervals — the results are shown in Figure 4.2.

Program 4.4

```
10 REM *
20 REM * The STEP Option
30 REM *
40 A=0
50 PRINT "Degrees","Radians"
60 PRINT
70 FOR I = 0 TO 360 STEP 45
80 R=I*(3.142/180)
90 PRINT I,R
100 NEXT
```

```
Degrees         Radians

 0               0
 45              .7855000000000002
 90              1.571
 135             2.3565000000000001
 180             3.1420000000000001
 225             3.9275000000000001
 270             4.7130000000000001
 315             5.4985000000000001
 360             6.2840000000000002
```

Figure 4.2 Degree/radian conversion using Program 4.4

A negative STEP value will, of course, decrement a loop variable. Like the repeat loop, the sequence of statements between FOR and NEXT statements will *always* be carried out at least once. If the following improbable FOR statement is used:

```
FOR I = 12 TO 15 STEP -1
```

one pass through the loop will be carried out, even though the end condition can never be reached.

One use for these loops is to add delays into programs. These are termed **idle loops**, as they do little else but waste time. An idle loop is shown here. The delay time will vary depending on the type and precision of the loop variable.

```
10 FOR D = 1 TO 1000 : NEXT D
```

Nested loops

One FOR . . . NEXT loop may be included within another FOR . . . NEXT loop. This is called **loop nesting**. The arrangement of nested loops looks like this:

```
10  FOR I = 1 TO 10
20  FOR J = 1 TO 10
    (Program statements)
50  NEXT J
60  NEXT I
```

Both the FOR and NEXT statements of the inner loop must lie *totally* within the FOR and NEXT statements of the outer loop. You must watch out for **crossed loops** which are not permitted. An illegal loop structure may be shown thus:

```
10  FOR I = 1 TO 10
20  FOR J = 1 TO 10
    (Program statements)
50  NEXT I
60  NEXT J
```

In loop nesting, only one NEXT statement need be used, for example:

```
NEXT K,J,I
```

handles three loops. This notation is not as clear as using a series of NEXT statements, and may make program error correction unnecessarily difficult.

A simple method of making nested loops easier to see is to **indent** loops in your program:

```
10  FOR I = 1 TO 10
20    FOR J = 1 TO 10
30      FOR K = 1 TO 10
          (statements)
70      NEXT K
80    NEXT J
90  NEXT I
```

Matching FOR and NEXT statements are much more obvious using this layout, but the extra spacing used means that more memory is required to store your program.

Nested loops are used in Program 4.5 to provide a print out of the song *One Man Went To Mow*. The computer 'sings' (if that is appropriate) the song until it completes the verse 'ten men went to mow'. Although seemingly abstract, this example is a good demonstration of how loop nesting is used.

Program 4.5

```
10 REM *
20 REM * Men Mowing Meadows
30 REM *
40 CLS
50 A$ = " One man and his dog
60 B$ = " went to mow a meadow"
70 PRINT " One man went to mow
80 PRINT " went to mow a meadow"
90 PRINT A$ : PRINT B$
100 FOR I=2 TO 10
110 REM * Delay Loop
120 FOR D = 1 TO 800:NEXT D
130 CLS
140 BEEP
150 PRINT I;" men went to mow
160 PRINT " went to mow a meadow"
170 FOR J=I TO 2 STEP -1
180 PRINT J;" men"
190 NEXT J
200 PRINT A$ : PRINT B$
210 NEXT
220 PRINT "The End"
230 END
```

```
10  men went to mow
went to mow a meadow
10  men
9  men
8  men
7  men
6  men
5  men
4  men
3  men
2  men
One man and his dog
went to mow a meadow

The End
```

Figure 4.3 The final verse of Program 4.5!

Monitoring devices

All MSX computers have at least one Input/Output (I/O) port. More often than not, this port is used for a joystick, but other, less common devices may be plugged in. MSX–BASIC provides a set of functions which read these ports. The most important of these is undoubtedly the STICK( ) function.

Any one of three values may be given as a parameter to STICK( ) as follows:

- Ø Read the cursor keys.
- 1 Read the joystick attached to Port A.
- 2 Read the joystick attached to Port B.

STICK( ) returns the current direction of a given joystick. The values returned are described thus:

- Ø Neutral (centred)
- 1 Up
- 2 Up and right
- 3 Right
- 4 Down and right
- 5 Down
- 6 Down and left

7 Left
8 Up and left

All the program examples in this book use STICK(Ø) — the cursor keys. This function is used extensively in later chapters, so there is little point in dwelling on it here.

Joysticks generally have a single trigger button (some have two) which can be monitored using BASIC. The STRIG( ) function returns the status of a given trigger button (whether pressed or not). The parameters that may be used with STRIG( ) are as follows:

Ø The spacebar
1, 3 Trigger buttons for joystick A
2, 4 Trigger buttons for joystick B

STRIG( ) always returns either −1 (trigger pressed) or Ø (not pressed). These two values are examples of MSX **Boolean** values. *True* is given by the value −1, while Ø or any other value is taken to be *false*. The following expression, although it may look odd, is perfectly valid:

```
IF STRIG(Ø) THEN BEEP
```

It produces a BEEP if the trigger button is pressed (i.e., STRIG(Ø) returns −1).

If this function is used in a game, to fire missiles and the like, STRIG( ) must be monitored many times a second. Frequent examination of a value is termed **polling**.

The polling method is used in Program 4.6. The routine starts by asking the user whether the cursor keys or joystick are to be used. The status of all the trigger buttons is polled, and the joystick selected depends on the trigger button pressed.

Program 4.6

```
10 REM *
20 REM * STRIG() Function
30 REM *
40 CLS
50 WIDTH 40
60 PRINT "Press The Trigger Button or"
70 PRINT "Space bar to select Joystick"
80 PRINT "or the cursor keys.."
90 PRINT : PRINT
100 PRINT "<PRESS NOW>"
110 PRINT : PRINT
```

```
120 REM *
130 REM * Poll all trigger buttons
140 REM *
150 F=-1
160 FOR I=0 TO 4
170 IF STRIG(I) THEN F=I
180 NEXT I
190 IF F THEN GOTO 160
200 BEEP
210 REM *
220 REM * Print the stick to be used
230 REM *
240 IF F=0 THEN PRINT "Cursor Keys";
250 IF F=1 OR F=3 THEN PRINT "Stick A";
260 IF F=2 OR F=4 THEN PRINT "Stick B";
270 PRINT " Selected":END
```

There are other devices that may be plugged-in to the joystick ports, and which may be monitored from BASIC — namely touchpads and paddles. These are less common. Although examples of their use are not given here, for completeness, their associated functions are detailed in the Appendix 1.

Monitoring the keyboard

MSX–BASIC can be made to examine the keyboard. The function that does this is called INKEY$. There are no parameters for this function (there is only one keyboard!). INKEY$ returns the value of the key currently being pressed (if any).

This function also needs to be polled to be of any use. Program 4.7 shows INKEY$ in use.

Program 4.7

```
10 REM *
20 REM * INKEY$ Function
30 REM *
40 CLS
50 PRINT "Press Any Key.."
60 PRINT
70 A$ = INKEY$
80 IF A$="" THEN 70
90 BEEP
```

```
100 PRINT "Key pressed was '";A$;"'"
110 END
```

Further functions associated with the keyboard are detailed in Chapter 5.

Error conditions

There are four principle types of error that may be encountered while programming.

1. **Logic errors** These occur when the program *runs*, but does not work in the way you expected.
2. **Syntax errors** These are normally encountered while developing a program and mean that the interpreter does not understand your program statement.
3. **Hardware errors** These are the rarest of all. Malfunction of your computer or associated equipment is not something that can be prevented by software means.
4. **Run-time errors** Although syntax errors are often detected when a program is run, I'll limit this classification to conditions like *overflow* or *type conversion* errors.

Syntax and logic errors can be rectified during the development of a program. Run-time errors are not so easy to correct. If a user tries to put the value 10^{999} into a integer variable, then the program will grind to a halt with the 'Overflow' error message. The same thing will happen if the intermediate value of a calculation results in numeric overflow.

If you write programs that may be used by somebody else on a regular basis, it is better that such errors do not stop or **crash** the program. MSX–BASIC allows errors to be detected and handled under program control rather than by the interpreter as is the norm.

Each error that may occur has an associated **error code** — for example, error number 2 is the familiar 'Syntax error'. When an error occurs, the error code is stored in the special variable ERR. The line number of the statement where the error occurred is conveniently stored in the reserved variable ERL.

Interrupts

These are means of finding out what happened where, but what a programmer really needs to know is *when* an error arises, so that appropriate steps may be taken to deal with the problem. Both the

special error variables could be polled, but this is impractical for all but the shortest programs. A special type of branch instruction called an **interrupt** is used instead.

If an interrupt is set (enabled) for an event, MSX–BASIC will automatically check for the occurrence of the event at the beginning of each program line it encounters. This is far more convenient and faster than polling.

When the occurrence of the particular event is noted (trapped), a branch to a program section that handles that particular event occurs automatically. That section of code is generally termed the **interrupt servicing routine**.

Error trapping in this way is turned on by use of the ON ERROR GOTO statement. This statement must also give the line number of the start of the interrupt servicing routine. It need only be given once in a program, usually at the beginning.

The interrupt servicing routine has to end with one of two statements: END or RESUME. The latter statement allows processing to continue after an error condition has been dealt with and has much in common with the RETURN statement. It has one of these forms:

RESUME or RESUME 0	Continue execution from the line where the error occurred.
RESUME NEXT	Continue execution from the line immediately after the line where the error was encountered.
RESUME <line number>	Continue execution from the given line number.

Program 4.8 is a short example demonstrating how error trapping may be incorporated into programs. The error condition monitored is that of 'numeric overflow'.

Program 4.8

```
10 REM *
20 REM * Error Handling
30 REM *
40 ON ERROR GOTO 140
50 CLS
60 WIDTH 38
70 INPUT "Number Please";A%
80 PRINT "Number";A%;"is within the"
90 PRINT "acceptable Integer range."
```

```
100 END
110 REM *
120 REM * Error Handling
130 REM *
140 IF ERR=6 THEN PRINT "!!OVERFLOW!!":P
RINT "Outside range for an Integer"
150 RESUME
```

Defining error conditions

You are not restricted to setting traps for only MSX–BASIC error messages. If your program regards numeric inputs outside the range 0–1000 as invalid they can be assigned an error code. MSX–BASIC does not *as yet* use all the possible codes — codes 26–49 and 60–255 are currently available for programmer use. (Available ranges may change in future versions of MSX–BASIC.)

The code for the input validation routine should be in the upper range of error codes. An sample sequence of statements could be:

```
10 ON ERROR GOTO 1000
  . . .
50 IF X<0 OR X>1000 THEN ERROR 255
  . . .
1000 IF ERR=255 THEN PRINT "Out of range"
1010 RESUME
```

Line 50 causes an automatic branch to the error handling routine once the error condition is detected.

There is very little virtue in defining your own error messages. The method is untidy — two IF . . . THEN statements are needed when only one is necessary. Its main advantage lies in the fact the error messages can be grouped together in one routine, so making the process of updating and modifying error handling a little easier.

Timer interrupts

The pulse which is counted by the TIME variable may also be used as a reference for another MSX–BASIC interrupt. Once this interrupt is set, a branch to a specified interrupt servicing routine will occur at a given time interval. Trapping is turned on with the INTERVAL ON statement. The branch to the servicing routine is indicated with the ON INTERVAL command. The time period to be monitored is given in 1/50ths of a

second. For example:

ON INTERVAL = 100

causes a branch to the subroutine every two seconds — provided an INTERVAL ON statement has already been issued. INTERVAL trapping may be turned off or suspended by the use of two statements: INTERVAL OFF explicitly turns the trapping off; INTERVAL STOP causes MSX–BASIC to maintain monitoring of the time interval, but the effect of the branch instruction is suppressed. As soon as an INTERVAL ON statement is issued, MSX–BASIC 'remembers' if an interrupt condition occurred and a branch to the servicing routine takes place immediately.

Normally, an INTERVAL STOP is issued by the *interpreter* when a branch to the servicing routine occurs, followed by an automatic INTERVAL ON when returning from the subroutine.

Program 4.9 produces a BEEP every 10 seconds using this interrupt timer.

Program 4.9

```
10 REM *
20 REM * Interval Interrupt
30 REM *
40 ON INTERVAL=500 GOSUB 110
50 INTERVAL ON
60 CLS : PRINT "10s Timer"
70 GOTO 70
80 REM *
90 REM * Interval Routine
100 REM *
110 BEEP : PRINT "10 Second Interval"
120 RETURN
```

The ON/STOP/OFF structure is common to all the remaining interrupt commands.

The trigger button interrupt

Trapping of the joystick trigger buttons is turned on by use of STRIG(⟨N⟩) ON, where N is a trigger button number. The branch to the interrupt service routine is controlled by the following statement:

ON STRIG GOSUB ⟨line number⟩[,⟨line number⟩]. .

This operates in a similar manner to the ON GOSUB statement. If

trapping for trigger 0 (the spacebar) and trigger 1 is enabled; if trigger 0 is pressed, a branch to the first line number in the list will occur and if Trigger 1 is pressed, the second line number will be used as the branch reference, and so forth. Up to five line numbers may be given here.

There are also associated STRIG(⟨N⟩) STOP and STRIG(⟨N⟩) OFF commands. Both the STRIG and INTERVAL interrupts are used in Program 4.10 to provide a simple reaction timer.

Program 4.10

```
10 REM *
20 REM * INTERVAL Interrupt:
30 REM * Reaction Timer
40 REM *
50 ON STRIG GOSUB 270
60 CLS
70 R=RND(-TIME)
80 PRINT "Reaction Timer"
90 PRINT
100 PRINT "When you hear a beep"
110 PRINT "Press the Space Bar"
120 PRINT
130 T=INT(RND(1)*500)+10
140 ON INTERVAL = T GOSUB 200
150 INTERVAL ON
160 GOTO 160
170 REM *
180 REM * Interval Handling Routine
190 REM *
200 INTERVAL OFF
210 BEEP:TIME=0
220 STRIG(0) ON
230 GOTO 230
240 REM *
250 REM * Trigger Button Routine
260 REM *
270 STRIG(0) OFF
280 PRINT "Reaction Time"
290 PRINT "=";TIME/50;"seconds"
300 END
```

Monitoring the function keys

Again much the same structure is used as for the previous interrupts.

KEY(⟨N⟩) turns on trapping for a specific function key, while ON KEY GOSUB provides the branch instruction. Note that KEY ON and KEY OFF commands are not associated with function key trapping: these commands merely turn the function key *display* on or off. Up to ten line numbers may be specified in the ON KEY GOSUB statement.

CTRL–STOP monitoring

When the CTRL and STOP keys are pressed, program execution will normally be aborted. This can be prevented using the ON STOP interrupt. The main use of this interrupt is to make program break-proof. It must always be the last item added to any program under development. If not used with care, a program may be created which can never be stopped except by turning the computer off. A sequence showing its use is:

```
10 ON STOP GOTO 1000
   . . .
1000 RETURN
```

Points to note

When both function key 1 and trigger button 0 have interrupts set, some anomalous behaviour may be seen when monitoring STICK(0). If the spacebar and the cursor keys are pressed simultaneously, MSX–BASIC will sometimes interpret this as though function key 1 has been pressed.

The ON ERROR GOTO statement will *always* reset any interrupts currently set. Ideally, some provision must be made to reset interrupts in the error handling routine.

As interrupts are seemingly invisible in operation they must always be used with a good deal of forethought, particularly in large programs and where numerous interrupt conditions may be set.

Summary

FOR ⟨count variable⟩ = ⟨initial value⟩ TO ⟨end value⟩ [STEP ⟨increment/decrement⟩]

Functions

INKEY$, STICK(⟨N⟩), STRIG(⟨N⟩)

For details of the following see Appendix 1:

PAD(⟨N⟩), PDL(⟨N⟩)

Interrupts

Error handling
ON ERROR GOTO ⟨line number⟩

ERROR = ⟨error number⟩

RESUME [Ø]

RESUME NEXT

RESUME ⟨line number⟩

END

ERR, ERL

Timer interrupts
ON INTERVAL = ⟨time interval × 1/5Øs⟩ GOSUB ⟨line number⟩

INTERVAL ON ¦ STOP ¦ OFF

Trigger buttons
ON STRIG GOSUB ⟨line number⟩[,⟨line number⟩][. . .]

STRIG (⟨N⟩) ON ¦ STOP ¦ OFF

Function keys
ON KEY GOSUB ⟨line number⟩[,⟨line number⟩][. . .]

KEY (⟨N⟩) ON ¦ STOP ¦ OFF

CTRL–STOP
ON STOP GOSUB ⟨line number⟩

STOP ON ¦ OFF ¦ STOP

5 Input, output and string handling

INPUT and PRINT are two of the most commonly used BASIC statements. Although they are very convenient to use, they are not particularly elegant. This chapter looks at the different ways of taking data from the outside world, and producing a more streamlined output.

There are a large number of MSX–BASIC functions which are relevant to processing string data. These functions and their usage will be described in some detail.

The MSX–BASIC character set

Each character that may be used in MSX–BASIC has a unique code number known as its **ASCII** code. For example, the character 'A' has the code 65 and the plus sign '+' has the code 43. The ASCII character code was designed to allow character exchange between different computer systems: 65 will always produce the letter 'A' on any computer that uses the ASCII code. ASCII code defines a total of 128 characters. Codes Ø–31 have a special significance and are known as the control codes — for example, code 13 is the code for the RETURN character.

MSX–BASIC has more than 128 characters, and these have non-standard ASCII codes. They have the code numbers 129 to 255, and include those characters that you will not find on every ASCII computer, such as special graphics characters.

If you want to find the code number for a particular character, then you can use the ASC( ) function. ASC returns the ASCII code for the first character in a string. For example:

```
PRINT ASC("ZAPHOD")     gives 9Ø (the code for 'Z).
PRINT ASC("*")          gives 42.
```

If ASC( ) is given an empty string as an argument it causes an error.

A complementary function to ASC( ) is CHR$( ). CHR$( ) returns the character for a given code. PRINTing CHR$(7) (a control character)

produces a beep, and CHR$(33) produces an exclamation mark. Some of the MSX–BASIC characters require a double code. To produce these codes, CHR$(1) is added to the CHR$( ) value of a number from 65 to 95; e.g.:

CHR$(1)+CHR$(65)	produces a face.
CHR$(1)+CHR$(78)	produces a musical note symbol.

The ASC function will always return 1 if the character argument has a double code.

If you do not have a special MSX printer, it is better to include the non-standard ASCII characters in programs using the CHR$( ) function rather than using a string constant. Program 5.1 displays all the MSX–BASIC characters produced using CHR$( ) — the results are shown in Figure 5.1

Program 5.1

```
10 REM *
20 REM * Character Set
30 REM *
40 KEY OFF
50 COLOR 15,4,4
60 SCREEN 1
70 PRINT "Double Codes"
80 PRINT
90 FOR I=65 TO 95
100 PRINT CHR$(1)+CHR$(I);
110 NEXT
120 PRINT : PRINT
130 PRINT "Single Codes"
140 PRINT
150 FOR I = 32 TO 255
160 PRINT CHR$(I);
170 NEXT
```

String manipulation

The only string manipulation operation that has been seen so far is concatenation — adding two strings together. With the string functions available in MSX–BASIC, programs may be written to truncate strings, extract and replace sections of strings, and so forth. The first piece of information we can find out about a string is its length. The function

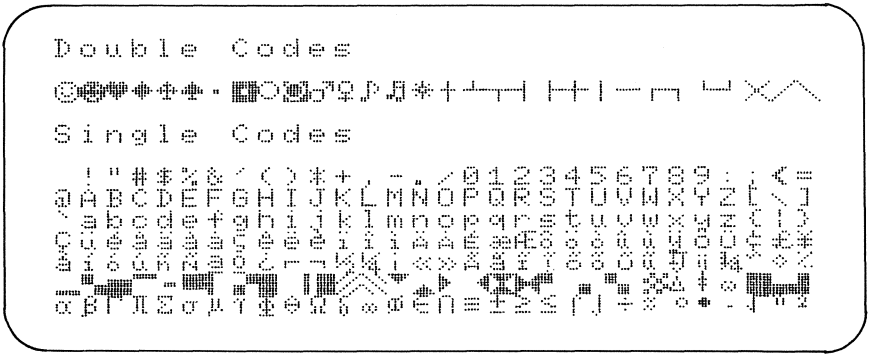


Figure 5.1 The MSX–BASIC character set from Program 5.1

that does this is LEN( ). It counts all the characters in a string, including the control characters which you cannot see, such as the backspace and beep characters. If LEN( ) is used with an empty string, Ø will be returned.

Program 5.2

```
10 REM *
20 REM * The LEN() Function
30 REM *
40 KEY OFF
50 CLS
70 PRINT "Line of text please:"
80 INPUT T$
90 PRINT
100 PRINT "The string is composed of";
LEN(T$)
110 PRINT "characters."
```

LEFT$( ) and RIGHT$( )

These functions return a number of characters from the leftmost or rightmost end of a string. They both have the same syntax:

LEFT$(⟨String Name⟩,⟨no. of characters to be returned from the left⟩
RIGHT$(⟨String Name⟩,⟨no. of characters to be returned from the right⟩

If the number of characters to be returned exceeds the length of the

string, then the whole string will be returned. Here are four examples of these functions in use:

```
LEFT$("MSX Computers",3)      gives "MSX"
RIGHT$("MSX Computers",9)     gives "Computers"
RIGHT$("Hiccup",2Ø0)          gives "Hiccup"
LEFT$("Yes",1)                gives "Y"
```

Program 5.3 takes a number, produces a binary string using BIN$( ) and swaps the lower 4 bits of the number with the upper four bits. Typical output is shown in Figure 5.2

Program 5.3

```
10 REM *
20 REM * Nibble Swapping
30 REM *
40 KEY OFF
50 SCREEN 0
60 INPUT "A number from 0-255";N
70 IF N<0 OR N>255 THEN GOTO 60
80 BYTE$ = BIN$(N)
90  REM *
100 REM * If the no. bits is less than
110 REM * eight, add extra zeros
120 REM *
130 IF LEN(BYTE$)<8 THEN BYTE$="0"+BYTE$
:GOTO 130
140 REM *
150 REM * Reverse Nibbles
160 REM *
170 H$ = LEFT$(BYTE$,4)
180 L$ = RIGHT$(BYTE$,4)
190 NBYTE$ = L$ + H$
200 PRINT
210 PRINT "Original Value:"
220 PRINT
230 PRINT "Binary:    ";BYTE$
240 PRINT
250 PRINT "New Value:"
260 PRINT
270 PRINT "Binary:    ";NBYTE$
280 END
```

```
A number from 0-255? 97

Original Value:

Binary:    01100001

New Value:

Binary:    00010110
```

Figure 5.2 Nibble swapping from Program 5.3

String search and replace

A string can be searched for the occurrence of another string using the INSTR( ) function. For example, we can see if the string "Egg" is found in a given string. INSTR( ) returns the position of the first character where a match for the search string is found. If one string is not present in the other, INSTR( ) returns 0:

```
PRINT INSTR("The World is an Egg", "Egg")   gives 17
PRINT INSTR("Elves are Green", "Hedgehogs")   gives 0
```

The position where the string search is to start may also be specified. This must be between 1 and 255.

```
PRINT INSTR(7,"Hello Mouse", "Hello")   gives 0
PRINT INSTR(7,"Hello Mouse", "us")   gives 9
```

By respecifying the start point for the search, the positions of all occurrences of one string within another may be found, and so counted. Program 5.4 and Figure 5.3 demonstrate word counting.

Program 5.4

```
10 REM *
20 REM * Word Counting
30 REM *
40 KEY OFF
50 SCREEN 0
60 WIDTH 38
70 PRINT "Line of Text Please:"
80 PRINT
```

```
90 INPUT T$
100 IF LEN(T$)=0 THEN GOTO 70
110 PRINT "Word to Count:"
120 INPUT CWD$
130 REM *
140 REM * Search and Count Words
150 REM *
160 I = 1 : C=0
170 X = INSTR(I,T$,CWD$)
180 IF X=0 THEN 230
190 C = C+1
200 I = X+1
210 GOTO 170
220 PRINT
230 PRINT
240 PRINT "Word : ";CWD$
250 PRINT "Occurrences :";C
260 END
```

```
Line of Text Please:

? Humpty Dumpty sat on a wall. Humpty
Dumpty had a great fall.
Word to Count:
? Humpty

Word : Humpty
Occurrences : 2
```

Figure 5.3 Word counting using Program 5.4

A portion of a string may be extracted using the MID$( ) function. The syntax of this function is given as:

MID$(A$,X [,Y])

A$ is the string to be worked on, X is the start position in the string, and Y is the length of the string to be returned. If the last argument is omitted, the rightmost characters from the start position will be returned. The way it works can be seen in these examples:

PRINT MID$("Large snakes are horrible",7) gives "snakes are horrible"
PRINT MID$("Fred is a brain surgeon", 11,5) gives "brain"

One possible use for MID$( ) is to scan a string, so allowing all lower case letters to be converted to upper case as in Program 5.5. Typical results are shown in Figure 5.4

Program 5.5

```
10 REM *
20 REM * Lower>Upper Case Conversion
30 REM *
40 KEY OFF
50 SCREEN 0
60 WIDTH 38
70 PRINT "Text Please:"
80 PRINT
90 INPUT T$
100 L = LEN(T$)
110 IF L=0 THEN GOTO 90
120 N$ = ""
130 FOR P = 1 TO L
140 S$ = MID$(T$,P,1)
150 IF S$<"a" OR  S$>="z" GOTO 240
160 REM *
170 REM * Actual Conversion
180 REM *
190 S = ASC(S$)-32
200 S$ = CHR$(S)
210 REM *
220 REM * Build-up New String
230 REM *
240 N$ = N$ + S$
250 NEXT P
260 PRINT
270 PRINT "Old Text : " : PRINT
280 PRINT T$
290 PRINT
300 PRINT "New Text : " : PRINT
310 PRINT N$
320 END
```

MID$ may also be used as a statement. One portion of a string replaces another when used in this way. The syntax for the MID$ statement is:

MID$(⟨String Variable⟩,⟨Start position⟩[,⟨String length⟩] = ⟨Replace String⟩

```
Text Please:

? Some of this TEXT is lower case.

Old Text :
Some of this TEXT is lower case.

New Text :
SOME OF THIS TEXT IS LOWER CASE.
```

Figure 5.4 Lower/upper case conversion using Program 5.5

If we want to replace the word "trees" with the word "frogs" in the phrase "All trees are green", we would use the expression as in Program 5.6.

Program 5.6

```
10 REM *
20 REM * Simple String Substitution
30 REM *
40 A$ = "All trees are green"
50 B$ = "Frogs"
60 PRINT A$
70 MID$(A$,5,5)=B$
80 PRINT A$
```

Note that the length of the original string variable is not changed in any way. If we were to replace "trees" with the string "great big elephants", the result would be:

"All great big eleph"

MID$ used as a statement is quite inconvenient for normal string replacement operations: only strings of identical length can be replaced. By using the other string functions, we can get around this limitation quite easily. Program 5.7 allows insertion, deletion and replacement of sections of text — Figure 5.5 illustrates its use.

Program 5.7

```
10 REM *
20 REM * Insert, Delete, Replace
30 REM *
```

```
40 KEY OFF
50 SCREEN 0
60 WIDTH 38
70 CLEAR 400
80 PRINT "String Editing"
90 PRINT "---------------"
100 PRINT
110 PRINT "Type in text :"
120 INPUT TEXT$
130 L=LEN(TEXT$)
140 IF L=0 THEN GOTO 120
150 REM *
160 REM * Main Loop
170 REM *
180 PRINT
190 PRINT "--------------"
200 PRINT "1- Insert":PRINT "2- Delete
":PRINT "3 - Replace"
210 PRINT "--------------"
220 PRINT
230 PRINT "Text: ";TEXT$ : PRINT
240 INPUT "Option";O
250 IF O<1 OR O>3 THEN GOTO 200
260 PRINT
270 ON O GOSUB 320,470,600
280 GOTO 190
290 REM *
300 REM * Insert Routine
310 REM *
320 INPUT "Insert Text:";I$
330 IL=LEN(I$) : IF IL=0 THEN 320
340 IF IL+L>255 THEN PRINT "* No Room"
: RETURN
350 PRINT
360 INPUT "Insert at position ";P
370 IF P<1 OR P>LEN(TEXT$) THEN 360
380 PRINT
390 IF P>L+1 THEN GOTO 360
400 A$ = LEFT$(TEXT$,P-1)
410 B$ = RIGHT$(TEXT$,L-P+1)
420 TEXT$=A$ + I$ + B$
430 L = LEN(TEXT$)
440 RETURN
450 REM *
```

```
460 REM * Delete Routine
470 REM *
480 INPUT "Delete Text:";D$
490 PRINT
500 DL=LEN(D$) : IF DL=0 THEN 480
510 P = INSTR(TEXT$,D$)
520 IF P=0 THEN PRINT "* Not found"
:PRINT : RETURN
530 A$=LEFT$(TEXT$,P-1)
540 B$=RIGHT$(TEXT$,L-P-DL+1)
550 TEXT$=A$+B$
560 L=LEN(TEXT$)
570 RETURN
580 REM *
590 REM * Replace Routine
600 REM *
610 INPUT "Search Text:";S$
620 SL=LEN(S$) : IF SL=0 THEN 610
630 P = INSTR(TEXT$,S$)
640 IF P=0 THEN PRINT "* Not found" : PR
INT : RETURN
650 PRINT
660 INPUT "Replace Text:";R$
670 RL=LEN(R$) : IF RL=0 THEN 670
680 PRINT
690 REM *
700 REM * Replace String=in length
710 REM *
720 IF RL=SL THEN MID$(TEXT$,P,RL)=R$ :
RETURN
730 IF RL>SL GOTO 870
740 REM *
750 REM * Replace Long With Short
760 REM * String
770 REM *
780 MID$(TEXT$,P,RL)=R$
790 P=P+RL : DL=SL-RL
800 GOSUB 530 : RETURN
810 REM *
820 REM * Replace Short With Long
830 REM * String
840 REM *
850 B = RL-SL
860 IF B+L>255 THEN PRINT "* No room"
:RETURN
```

```
String Editing
--------------

Type in text :
? This is a spamule piece of text.

-------------
1- Insert
2- Delete
3 - Replace
-------------

Text: This is a spamule piece of text.

Option?  3

Search Text:? spamule
Replace Text:? sample

-------------
1- Insert
2- Delete
3 - Replace
-------------

Text: This is a sample piece of text.

Option?  2

Delete Text:? penguin

* Not found

-------------
1- Insert
2- Delete
3 - Replace
-------------

Text: This is a sample piece of text.
```

Figure 5.5 Text editing with Program 5.7

```
870 MID$(TEXT$,P,SL)=R$
880 P = P+SL
890 A$ = LEFT$(TEXT$,P-1)
900 B$=RIGHT$(TEXT$,L-P+1)
910 TEXT$=A$+RIGHT$(R$,B)+B$
920 L = LEN(TEXT$)
930 RETURN
```

Variations on INPUT

One of the most inconvenient features of INPUT is the question mark which invariably appears as a prompt. An alternative to INPUT is the LINE INPUT statement, which does not automatically output a question mark. LINE INPUT has a syntax similar to that of INPUT, and allows up to 254 characters to be input from the keyboard. Data is displayed on the screen as it is typed, and input is terminated by typing RETURN. The main disadvantage of this statement is that it may only accept string data. If numeric input data is required LINE INPUT on its own is not sufficient.

The MSX–BASIC function VAL( ) converts string representations of numbers to actual numeric values. So VAL("121.62") produces the number 121.62. An empty string argument will always cause VAL( ) to return 0. Any leading blanks and control characters in the string will be ignored in the conversion process.

A prime use of VAL( ) is to convert strings representing binary, hexadecimal and octal values into decimal numbers. VAL("&B10000001") returns the number 129 for example.

The complementary function to VAL( ) is STR$( ), which converts numeric values to strings. When numbers are printed out, a space is left in front of the first digit. If a number is converted to a string before printing, this space can be removed by the use of RIGHT$( ) or MID$( ). This method allows text and numeric output to be more uniform. Program 5.8 demonstrates the combined use of VAL( ), STR$, and LINE INPUT.

Program 5.8

```
10 REM *
20 REM * VAL and LINE INPUT
30 REM *
40 LINE INPUT "X value = ";A$
50 LINE INPUT "Y value = ";B$
60 PRINT
```

```
70 X = VAL(A$)
80 Y = VAL(B$)
90 P = X*Y
100 PRINT A$;CHR$(42);B$;" =";P
110 PRINT
120 P$=STR$(P)
130 PRINT "Most sig. digit of product is
: ";
140 PRINT MID$(P$,2,1)
150 END
```

Another keyboard input feature is the INPUT$( ) function, which reads a number of characters and assigns them to a string variable. In this case, the characters are not displayed on the screen as they are typed, and the obligatory RETURN is not needed to signal the end of input. The statement A$ = INPUT$(4) will assign the first four characters typed to the variable A$. Program 5.9, a simple password program, shows one use of this function — although as a security system it leaves a lot to be desired . . .

Program 5.9

```
10 REM *
20 REM * Password
30 REM *
40 CODE$ = "SECRET" : COUNT=0
50 PRINT "Name: ";
60 INPUT N$
70 PRINT "Password: "
80  REM *
90  REM * Read Six Character String
100 REM *
110 P$=INPUT$(6)
120 IF P$=CODE$ THEN GOTO 150
130 COUNT=COUNT+1
140 IF COUNT < 3 THEN GOTO 70 ELSE PRINT
"Access Denied": END
150 PRINT
160 PRINT N$
170 PRINT "You may pass friend"
180 END
```

A far better use of INPUT$( ) is in accepting data from the keyboard one character at a time. If a program is required to accept a number, INPUT$( ) can be used to check each character of the number as it is

typed in. Basically, we can say that all numbers typed in are to be made up of following characters:

1. A '−' or '+' sign, if, and only if it is the first character of the number string.
2. A single decimal point.
3. The characters '0' to '9'.

It is assumed that leading blanks are to be ignored (VAL ( ) does this for us anyway), and blanks cannot occur anywhere in a number. The number is deemed complete when the RETURN key is pressed. An input routine which carries out number input in this manner is shown in Program 5.10.

Program 5.10

```
10 REM *
20 REM * Number Verifying
30 REM *
40 CLS
50 N$ = "" : P=0
60 PRINT "Number Please: ";
70 A$ = INPUT$(1)
80 IF A$= CHR$(13) THEN 180
90 IF A$= "+" AND N$="" THEN 150
100 IF A$="-" AND N$="" THEN 150
110 IF A$= "." THEN P=P+1
120 IF P= 1 THEN 150
130 IF A$>="0" AND A$<="9" THEN 150
140 BEEP : GOTO 70
150 N$=N$+A$
160 PRINT A$;
170 GOTO 70
180 PRINT : PRINT
190 PRINT "Number is: ";VAL(N$)
200 END
```

This routine could be increased in scope to allow numbers to be given in floating point representation and also to check that the magnitude of a number is within an acceptable range. In addition, the use of the INS and DEL keys could be catered for, to allow editing of the input line.

INPUT$( ) is much neater to use than the INKEY$ function. The value of INKEY$ has to be checked continuously until it has a value other than null, whereas INPUT$ polls the keyboard automatically until it has read a set number of values.

Cursor positioning

The familiar block on the screen, the cursor, can be put anywhere on the screen using the LOCATE command. The text screens have a maximum of 40×24 positions for SCREEN 0 and 32×24 positions for SCREEN 1. By giving a pair of coordinates to the LOCATE command, we can change the cursor position on the screen and print out data wherever is desired.

LOCATE requires the coordinates for horizontal and vertical position on the screen. The convention for naming these coordinates is based on the fact that position 0,0 is in the top left-hand corner of the screen. If a WIDTH command has been given previously then this will affect the range of coordinates that may be used by LOCATE. For example, if WIDTH 20 has been specified, an attempt to place the cursor at position 25,20 will cause an error. If no WIDTH command has been specified the default width for each screen is assumed. Program 5.11 prints out a text string in different places on the screen.

Program 5.11

```
10 REM *
20 REM * LOCATE
30 REM *
40 X = RND(-TIME)
50 SCREEN 0
60 KEY OFF : WIDTH 40
70 FOR I = 1 TO 20
80 X=INT(RND(1)*35)
90 Y=INT(RND(1)*23)
100 LOCATE X,Y,1
110 PRINT "HELLO";
120 NEXT
```

The cursor itself can be turned off by setting a switch at the end of a LOCATE command:

0 Turns the cursor off.
1 Turns the cursor on.

Two functions allow us to determine the position of the cursor on the screen. CSRLIN returns the vertical position of the cursor, POS(0) returns the horizontal position — POS( ) always uses a dummy argument. Program 5.12 allows text to be typed in and updates a clock on the top right of the screen. The values of CSRLIN and POS( ) are read before updating the time display, so allowing the original cursor

position to be restored. Note that the keyboard is polled using INKEY$. INPUT$( ) would be unsuitable here as it waits for a key to be pressed, so the keyboard and the clock processing cannot be carried out equally.

Program 5.12

```
10 REM *
20 REM * Located Clock
30 REM *
40 CLS : WIDTH 40
50 ON INTERVAL=50 GOSUB 230
60 TIME = 0
70 INTERVAL ON
80 LOCATE 16,0,0 : PRINT "Time:"
90 LOCATE 24,0 : PRINT H;":";M;":";S
100 INTERVAL ON
110 A$=INKEY$ : IF A$="" THEN GOTO 110
120 PRINT A$;
130 REM *
140 REM * Clear Screen when full
150 REM *
160 IF CSRLIN=23 AND POS(0)=37 THEN
PRINT A$:CLS:LOCATE 0,1,0:GOTO 80
170 GOTO 110
180 REM *
190 REM * Interval Interrupt Routine
200 REM *
210 REM * Save X,Y Cursor Coordinates
220 REM *
230 X=POS(0) : Y = CSRLIN
240 S=S+1
250 IF S = 60 THEN M = M+1 : S=0
260 IF M=60 THEN H=H+1 : M=0
270 IF H=24 THEN H=0 : LOCATE 24,0,0
:PRINT"                    "
280 LOCATE 24,0,0 : PRINT H;":";M;":";S
290 LOCATE X,Y,1
300 RETURN
```

Miscellaneous string functions

The PRINT command allows a number of special functions to be included in the expression to be printed. The first of these is TAB( ). TAB moves the cursor along by a number of horizontal positions from

the left-hand side of the screen. The number of positions moved is specified by an integer argument, which must be between 0 and 255. If the argument given is greater than the current screen width, then the cursor will move down a line.

Another function which can be used with a PRINT command is SPC( ). This prints out a given number of spaces. (Note that with TAB( ) only the cursor *position* is moved, whereas SPC( ) actually generates a number of characters and moves the cursor along correspondingly.) These points are illustrated in Program 5.13 and Figure 5.6.

Program 5.13

```
10 REM *
20 REM * SPC and TAB
30 REM *
40 SCREEN 0
50 KEY OFF : WIDTH 38
60 FOR I = 0 TO 5
70 PRINT TAB(I);"*";
80 NEXT
90 PRINT
100 FOR I = 0 TO 5
110 PRINT SPC(I);"*";
120 NEXT
```

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

Figure 5.6 Program 5.13 – using SPC and TAB

STRING$ allows a string to be created in which every character is the same. STRING$( ) may be supplied with an ASCII code, or a string argument. PRINT STRING$(35,"*") prints out a string of 35 asterisks, as would the command PRINT STRING$(35,42). If a character string longer than 1 is given as an argument in the second use of STRING$( ), only the first character in the string will be used. If the first character of a given variable is defined with a double code, STRING$ returns a string with each element of the string defined by ASCII code 1.

SPACE$( ) produces a string of a given length which is made up entirely of spaces. Either of these functions may be used to produce 'fillers' while outputting data. Program 5.14 prints out a series of strings such that the rightmost characters of each string lie above each other. This is known as right justification.

Program 5.14

```
10 REM *
20 REM * Right Justification
30 REM *
40 SCREEN 0
50 KEY OFF : WIDTH 40
60 PRINT TAB(13);"Right Justified"
70 PRINT : PRINT
80 FOR I = 0 TO 9
90 READ A$
100 FILL = 35-LEN(A$)
110 PRINT I;STRING$(FILL,"_");A$
120 NEXT
130 DATA "The Quick Brown Fox"
140 DATA "Hellzapoppin"
150 DATA "Queen Victoria"
160 DATA "Reginald Bosanquet"
170 DATA "The Crown Jewels"
180 DATA "Mr Pye"
190 DATA "Christopher Columbus"
200 DATA "ABC"
210 DATA "Quo Vadis"
220 DATA "ELstree Studios"
```

Program 5.15 uses STRING$( ) with TAB( ) to produce underlined, centred text on the screen. The variable W should be set to the current screen width.

Program 5.15

```
10 REM *
20 REM * Centre and Underline
30 REM *
40 W = 38 : REM * Set Screen Width
50 WIDTH W
60 SCREEN 0
70 LINE INPUT "Text: ";TEXT$
80 L = LEN(TEXT$)
90 CLS
100 REM *
110 REM * Centre
120 REM *
130 IDT = (W-L)/2
```

```
140 PRINT TAB(IDT);TEXT$
150 REM *
160 REM * Underline
170 REM *
180 PRINT TAB(IDT);STRING$(L,195)
```

Formatted output

One of the most attractive output features of MSX–BASIC is the powerful PRINT USING command, which surprisingly few people get around to using. PRINT USING provides a way of obtaining a neat and uniform output of text and characters. Its general syntax is:

PRINT USING ⟨format string expression⟩;⟨list of print items⟩

The format string expression controls how the items in the following list are to be printed. A number of format characters may be placed in this string, some of which control text output, with the remainder responsible for numeric output. The characters formatters are:

1. ! Prints the first character of a string.
2. & Allows one string to be printed embedded into another output string.
3. \⟨n spaces⟩\ This formatter prints out at least 2 characters from a string, plus as many characters as there are spaces between the two backslash symbols. Where there are more spaces allocated than there are elements in the string to be printed, remaining spaces will be filled with spaces.

Program 5.16 and Figure 5.7 show how these formatters may be used for strings.

Program 5.16

```
10 REM *
20 REM * Character Formatters
30 REM *
40 SCREEN 0
50 KEY OFF : WIDTH 38
60 PRINT
70 FOR I = 1 TO 3
80 READ A$
90 PRINT USING "!";A$
100 PRINT USING "The & Situation";A$
110 PRINT USING "\\";A$
```

```
120 PRINT USING "\   \";A$
130 PRINT USING "\       \";A$
140 PRINT STRING$(37,"-")
150 NEXT I
160 PRINT
170 DATA "International"
180 DATA "World"
190 DATA "Awkward"
```

```
I
The International Situation
In
Inte
Internat
-------------------------------------
W
The World Situation
Wo
Worl
World
-------------------------------------
A
The Awkward Situation
Aw
Awkw
Awkward
-------------------------------------
```

Figure 5.7 Formatted string output from Program 5.16

There are several numeric formatters — each will now be illustrated with an example.

1. # Used to print out digits. Each digit of the number to be printed requires one hash symbol, and if the value to be printed is likely to be negative it is wise to include an additional symbol. A decimal point may be used in the formatting string. The number will be printed in the following manner:

   (a) If, on the left-hand side of the decimal point, there are fewer digits than allowed for in the formatting string, the extra digit places will be filled with blanks; i.e., right justified. If this is the case for digits to the right of the decimal point, the trailing

places will be filled with zeros.

(b) Should the magnitude of the number to be formatted be greater than allowed for in the format string, a '%' sign will be displayed to indicate overflow.

(c) Numbers to the right of the decimal point will be rounded up where necessary.

Program 5.17 and Figure 5.8 should help to make this slightly clearer.

Program 5.17

```
10 REM *
20 REM * Numeric Formatters 1
30 REM *
40 CLS
50 PRINT TAB(1);"Raw","Formatted"
60 PRINT TAB(1);STRING$(3,195),STRING$
(9,195)
70 PRINT
80 FOR I = 1 TO 6
90 READ A
100 PRINT A,
110 PRINT USING "###.###";A
120 NEXT
130 DATA 142.854,93.281,1.62399
140 DATA -43,-112.45,1000.83
```

```
Raw             Formatted

 142.854         142.854
 93.281           93.281
 1.62399            1.624
-43              -43.000
-112.45          %-112.450
 1000.83         %1000.830
```

Figure 5.8 Program 5.17 – format using '#'

2. **– and +** Used to indicate the sign of the number printed. '–' is used at the end of the digit formatting string, and will print out a minus sign at the end of a number if it is negative. '+' at the beginning or end of a number indicates the sign of the number either as positive or negative — useful in that an extra digit symbol

need not be included in the format string to allow for negative values. Program 5.18 and Figure 5.9 illustrate this.

Program 5.18

```
10 REM *
20 REM * Numeric Formatters 2
30 REM *
40 CLS
50 PRINT "Formatted"
60 PRINT STRING$(9,195)
70 PRINT
80 PRINT USING "+###.#";42.34
90 PRINT USING "+###.#";-3.357
100 PRINT USING "###.##-";434.3
110 PRINT USING "###.##-";-324.48
```

```
Formatted

 +42.3
  -3.4
434.30
324.48-
```

Figure 5.9 Program 5.18 – format using '+' and '–'

3. ** Right justify a number using asterisks instead of spaces.
4. ££ Print a pound sterling sign at the beginning of a number. This symbol represents one digit place.
5. **£ Right justifies a number with asterisks if necessary, and adds a pound sterling sign.
6. , Print out numbers to the left of the decimal point using the old convention of separating 1000s, 100000s, etc. with a comma. (A comma will be placed every three digits to the left of the decimal point.)
7. ^^^^ Print the number in scientific (exponential) format.

Program 5.19 is a final roundup of these formatters in use.

Program 5.19

```
10 REM *
20 REM * Numeric Formatters 3
30 REM *
```

```
40 CLS
50 PRINT USING "**####.##";1.453
60 PRINT
70 PRINT USING "££####.##";321.63
80 PRINT
90 PRINT USING "**£##.##";-10.3
100 PRINT
110 PRINT USING "#########,";5411284#
120 PRINT
130 PRINT USING "##.#^^^^";141223!
140 END
```

```
****1.45

 £321.63

*-£10.30

5,411,284

 1.4E+05
```

Figure 5.10 Program 5.19 – mixed formatting

Normal text data may be included before the string formatting characters in any of the above PRINT USING examples. It is perfectly permissible to have them used as in Program 5.20.

Program 5.20

```
10 REM *
20 REM * Text and Formatters
30 REM *
40 PRINT USING "Tax p.a : ###.##";92.53
50 PRINT USING "State   : \    \.";
"California"
60 PRINT USING "Initials: !.!.";"Bill";"
Bloggs
```

Using a printer for output

If you are fortunate enough to have a printer attached to your system you are free to use all the MSX–BASIC variants of PRINT. Whatever

```
Tax p.a :  92.53
State   : Calif.
Initials: B.B.
```

Figure 5.11 Program 5.20 – mixing normal and formatted text

you can do with print, you can also do with LPRINT — there is even a version of PRINT USING called, naturally enough, LPRINT USING.

There is only one function related to printers that needs to be discussed here — the LPOS( ) function. Because the MSX computer can send out data faster than it is mechanically possible to print it (for most printers that is), an area of RAM, known as the **print buffer**, is set aside while printing is carried out. This buffer is filled up with print data by the CPU ready for the printer to fetch at its leisure. LPOS(Ø) returns the position of the print head in this buffer. (It is unlikely that you will use this particular function often, if at all.)

Printers are quite sophisticated devices, with processors and memory of their own. Using LPRINT, codes known as *escape sequences* can be sent to the printer to produce such things as condensed, enlarged, and emboldened text. How a printer responds to these different codes varies from manufacturer to manufacturer — your printer manual will give you full details.

The most common set of codes used by printers are those created by Epson — now also used by a large number of other manufacturers. Very few examples in this book use a printer, but those that do assume the Epson printer codes.

Summary

ASC(A$), CHR$(X), CSRLIN, INPUT$(X), INSTR([X,]A$,B$), LEFT$(A$,X), LEN(A$), LPOS(Ø), MID$(A$,X[,Y]), POS(Ø), RIGHT$(A$,X), SPACE$(X), SPC(X), STR$(X), STRING$(X,A$), STRING$(X,Y), TAB(X), VAL(A$)

LOCATE ⟨X⟩,⟨Y⟩[,⟨1 ¦ Ø⟩]

LPRINT [⟨expression⟩][separator][,⟨expression⟩] . . .

MID$(⟨string variable⟩,X [,Y]) = ⟨string expression⟩

PRINT USING ⟨string format expression⟩;⟨expression⟩ [;⟨expression⟩] . . .

PRINT USING ⟨string format expression⟩;⟨expression⟩ [;⟨expression⟩] . . .

6 Data structures

This chapter looks at the way in which a set of data items may be logically grouped together under one name.

All the variables discussed so far are classed as **simple variables** in which one variable name may only reference one single value.

There are many cases where this is undesirable. Assume that a program was required to store and process a list of 5Ø names. Such a program would require 5Ø independant variables; for example: N1$, N2$, N3$, . . ., N5Ø$. In this case, the variable N5Ø$ would not be distinguished from the variable N5$. A table of data merely increases the number of variables required, and the problem of variable naming.

It would be far more convenient if lists and tables of data could be referenced using a single variable name. The **array** data structure permits this.

Data lists

A list of data may be declared as an **array variable** using the DIM statement. This statement has two purposes:

1. It defines the number of elements in the list.
2. It defines the data type for each element of the list.

To declare a list of 5Ø double precision numbers the following statement could be used:

```
1Ø DIM A! (5Ø)
```

As shown, array variables may be set to a specified type by use of type declaration characters. Mixed data types are not permitted — an array may contain data of only a single type.

Each element of the list is referenced using a **subscript** (or **array index**). The array variable A! may be viewed as a series of memory locations as shown in Figure 6.1. The first element of the array is given by the subscript '1', or 'Ø', so we would refer to that element with the

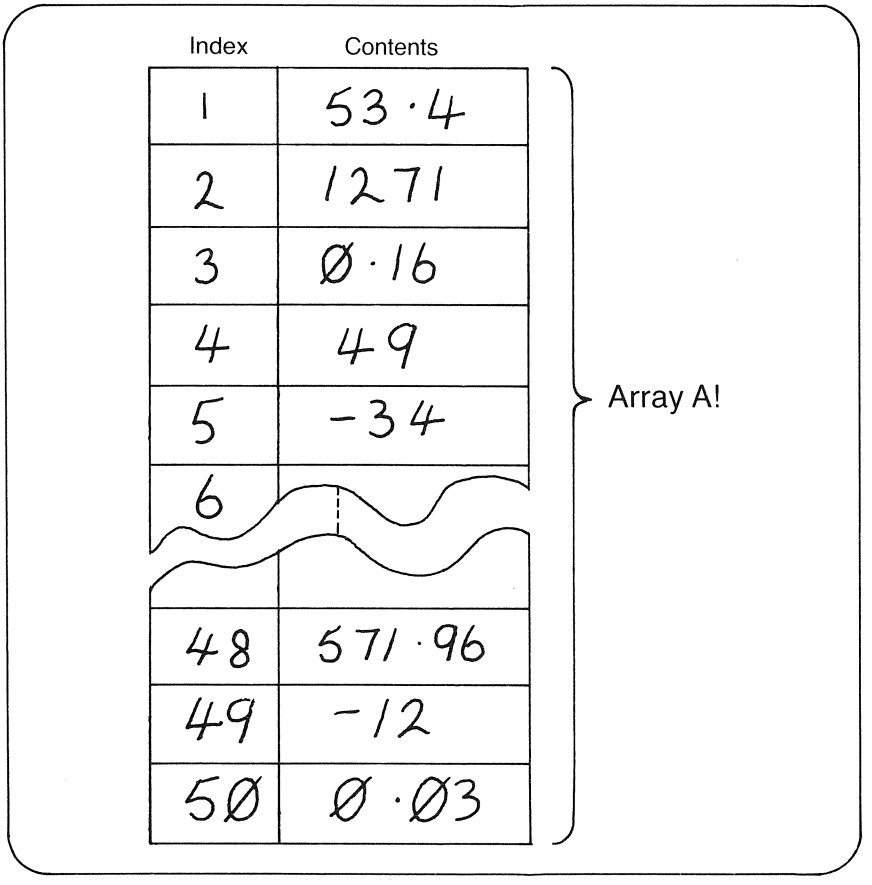


Figure 6.1 The elements of the array A!

variable name A!(1) or A!(Ø). The last element of the list would be given by A!(5Ø) or A!(49).

The lowest value permitted for an array subscript is Ø, and the highest is equal to the number of elements in the array. Any subscript outside this range will generate an error.

Array processing examples

Assume that we wish to select one of 1Ø names at random. This would not be too difficult to achieve without the use of arrays, but the result is likely to be rather unwieldy. Program 6.1 calculates a random value which is then used as a subscript to the array containing the list of names.

Program 6.1

```
10 REM *
20 REM * Array Handling
30 REM *
40 CLEAR 1000
50 DIM A$(10)
60 R=RND(-TIME)
70 CLS
80 REM *
90 REM * Read Names
100 REM *
110 FOR I = 0 TO 9
120 PRINT I;
130 INPUT "Name: ";A$(I)
140 NEXT
150 REM *
160 REM * Select Name at Random
170 REM *
180 R = INT(RND(1)*10)
190 N$ = A$(R)
200 PRINT
210 PRINT "Random Name is: ";N$
220 END
```

Program 6.2 also uses randomizing, this time to simulate dice-throwing. The results from a number of 'throws' are stored in an array. This information is then used to provide the percentage of throws which produced each number as illustrated in Figure 6.2.

Program 6.2

```
10 REM *
20 REM * Throwing a Die
30 REM *
40 DIM V(6)
50 X=RND(-TIME)
60 INPUT "How Many throws (max 100)";N
70 IF N<1 OR N>100 THEN 60
80 REM *
90 REM * Throw die N times
100 REM *
110 FOR I=1 TO N
120 T=INT(RND(1)*6+1):V(T)=V(T)+1
130 NEXT
```

```
140 REM *
150 REM * Calculate Percentages
160 REM *
170 CLS
180 PRINT "Number of Throws:";
190 PRINT USING "###";N
200 PRINT
210 PRINT "Value";TAB(8);"Total Thrown";
TAB(26);"Percentage"
220 PRINT
230 FOR I=1 TO 6
240 PRINT TAB(1);I;
250 PRINT TAB(12); : PRINT USING "###";V
(I);
260 PRINT TAB(27);
270 PRINT USING "###.## %";(V(I)/N)*100
280 NEXT
```

```
How Many throws (max 100)?  85


Number of Throws: 85

Value    Total Thrown       Percentage

  1           19              22.35 %
  2           12              14.12 %
  3           13              15.29 %
  4           11              12.94 %
  5           11              12.94 %
  6           19              22.35 %
```

Figure 6.2 Typical output from Program 6.2

Secondary indexing

The contents of one array may be used to index another array. This technique is known as **secondary indexing**. One possible use for the method is to allow an array of strings to be linked with a numeric array.

The index array would contain the subscripts used to reference values in separate numeric and a string arrays. Program 6.3 deals with the searching and sorting of such a payroll system. Searching is a

simple process. Each element of the array is compared against the search string until a match is found.

Sorting is carried out by a process of successive scans through the array data, exchanging values until the list of data is in the correct order. An A–Z or Z–A sorted sequence may be produced.

In order to ensure that strings such as "BELL" and "BROWN" are put into correct order, the leftmost two characters of the strings are compared. All the lower case characters are converted to upper case before comparison.

It is the indexing array that is actually sorted in this case, and the exchanges that take place are in the contents of this array. The advantage of this method is obvious if you are processing a considerable number of linked arrays. With our example, the data sorting requires only one exchange at a time on the index, instead of the two exchanges for sorting both the string and the numeric arrays.

Program 6.3

```
10 REM *
20 REM * Secondary Indexing
30 REM *
40 CLEAR 1000
50 DIM IDX(10)
60 DIM NM$(10)
70 DIM SLY(10)
80 REM *
90 REM * Set up arrays
100 REM *
110 FOR I=1 TO 10
120 IDX(I)=I
130 READ NM$(I),SLY(I)
140 NEXT
150 CLS
160 PRINT "1. Search
170 PRINT "2. Sort and list data"
180 PRINT
190 INPUT "Input Option ";A
200 IF A<1 OR A>2 THEN 190
210 ON A GOSUB 260,430
220 END
230 REM *
240 REM * Search
250 REM *
260 CLS
```

```
270 PRINT "Search for which name:"
280 INPUT S$
290 IF S$="" THEN 270
300 F=0
310 FOR I=1 TO 10
320 IF NM$(IDX(I))=S$ THEN S=SLY(IDX(I))
:F=-1
330 NEXT
340 IF NOT(F) THEN PRINT "Not Found":GOT
O 380
350 PRINT "Name   : ";S$
360 PRINT "Salary:";S
370 RETURN
380 BEEP
390 RETURN
400 REM *
410 REM * Sort
420 REM *
430 CLS
440 PRINT "1. A-Z"
450 PRINT "2. Z-A"
460 PRINT
470 INPUT "Input Option";A
480 IF A<0 OR A>2 THEN 470
490 CLS
500 PRINT "Sorting.." : PRINT
510 REM *
520 REM * Sort by first 2 letters
530 REM * of the name
540 REM *
550 FOR I = 2 TO 10
560 FOR J = 10 TO I STEP -1
570 A$ = LEFT$(NM$(IDX(J-1)),2)
580 B$ = LEFT$(NM$(IDX(J)),2)
590 IF A=2 THEN SWAP A$,B$
600 FOR P=1 TO 2
610 X=ASC(MID$(A$,P,I))
620 REM *
630 REM * Convert letter to Upper Case
640 REM * if needs be
650 REM *
660 IF X>90THEN MID$(B$,P,1)=CHR$(X-32)
670 X=ASC(MID$(B$,P,I))
680 IF X>90THEN MID$(A$,P,1)=CHR$(X-32)
```

```
690 NEXT P
700 IF A$>B$ THEN SWAP IDX(J-1),IDX(J)
710 NEXT J
720 NEXT I
730 FOR I=1 TO 10
740 PRINT NM$(IDX(I)),:PRINT USING "####
#";SLY(IDX(I))
750 NEXT I
760 RETURN
770 DATA "Tigger",5940
780 DATA "Bryant",7800
790 DATA "Cunliffe",15940
800 DATA "POTTS",10010
810 DATA "Major",8080
820 DATA "Tremayne",7100
830 DATA "Wirth",9500
840 DATA "Bartram",11340
850 DATA "Kernighan",6200
860 DATA "Sparks",7400
```

Tables

Lists are known as **one-dimensional arrays**. **Two-dimensional** arrays (matrices) provide the means of creating tables. Again, the DIM statement is used to create the array. For example, the statement:

```
10 DIM A(3,3)
```

creates a 3×3 table. Tables are used in the game of battleships illustrated in Program 6.4. In this version, the player places six ships in a 20×20 grid. The computer then places its own ships randomly on its own grid.

Both player and computer take it in turns to guess the location of the opponent's ships. The game is over when one player 'destroys' all the other's vessels.

After each guess, the location of the guess is indicated in the array by placing the value 99 in the opponent's grid. This allows duplicate guesses to be signalled.

Program 6.4

```
10 REM *
20 REM * Battleships
30 REM *
40 DIM CG(20,20),PG(20,20)
50 CS=0 : PS=0
60 SCREEN 0:WIDTH 36:KEY OFF
```

```
70 PRINT TAB(2);
80 FOR J=65 TO 84
90 PRINT CHR$(J);
100 NEXT
110 PRINT
120 FOR J=65 TO 84
130 PRINT CHR$(J)
140 NEXT
150 REM *
160 REM * Set up player Grid
170 REM *
180 LOCATE 0,24:PRINT "Place Ships - Let
ter:   Letter:";
190 FOR S=1 TO 6
200 LOCATE 22,24,1
210 A$=INPUT$(1)
220 IF A$<"A" OR A$>"T" THEN BEEP:GOTO
200
230 PRINT A$;
240 X=ASC(A$)-64
250 LOCATE 32,24,1
260 A$=INPUT$(1)
270 IF A$<"A" OR A$>"T" THEN BEEP:GOTO
200
280 PRINT A$;
290 Y=ASC(A$)-64
300 IF PG(X,Y)=0 THEN PG(X,Y)=1 ELSE
BEEP:GOTO 200
310 NEXT S
320 REM *
330 REM * Set Up Computers Grid
340 REM *
350 FOR S = 1 TO 6
360 X=INT(RND(1)*20)+1
370 Y=INT(RND(1)*20)+1
380 IF CG(X,Y)=0 THEN CG(X,Y)=1 ELSE
GOTO 360
390 NEXT S
400 REM *
410 REM * Start Play
420 REM *
430 LOCATE 28,0:PRINT "GUESS    ";
440 LOCATE 22,24,0
450 A$=INPUT$(1)
460 IF A$<"A" OR A$>"T" THEN BEEP:GOTO
```

```
440
470 PRINT A$;
480 X=ASC(A$)-64
490 LOCATE 32,24,1
500 A$=INPUT$(1)
510 IF A$<"A" OR A$>"T" THEN BEEP:GOTO
490
520 PRINT A$;
530 Y=ASC(A$)-64
540 IF CG(X,Y)=99 THEN BEEP:GOTO 450
550 LOCATE X+1,Y:PRINT "X";
560 IF CG(X,Y)=1 THEN PS=PS+1:LOCATE X+1
,Y:PRINT "S"
570 CG(X,Y)=99
580 IF PS=6 THEN CLS:PRINT "YOU WON!!!!"
:END
590 REM *
600 REM * Computers Guess
610 REM *
620 LOCATE 28,0:PRINT "Computer"
630 FOR I=1 TO 500:NEXT I
640 X=INT(RND(1)*20)+1
650 Y=INT(RND(1)*20)+1
660 IF PG(X,Y)=99 THEN 640
670 IF PG(X,Y)=1 THEN CS=CS+1
680 IF CS=6 THEN CLS:PRINT "I WON!":END
690 PG(X,Y)=99
700 GOTO 430
```

You can have more than two array dimensions, for example:

```
10 DIM A(3,3,3)
```

declares a three-dimensional array with a total of 27 elements. You do not usually need to declare arrays of more than three dimensions.

Array housekeeping and general details

Arrays may be wiped from memory by using the ERASE command. All the space allocated to the named array is released for general usage. The statement:

```
80 ERASE A!
```

destroys the array variable A!.

The only limitation on the array size is memory. This should be taken into account, particularly when using multi-dimensional and string arrays.

Use of files

The tape recorder may be used to record not only programs, but also program data. Cassette tape stores data serially. Data may be only retrieved in the sequence in which it was written to tape. This form of storage is very inflexible, and quite unsuitable for applications which require frequent updating of data. (It is, however, useful for long-term storage of data.)

Cassette tape may be read from, or written to in programs by treating the cassette recorder as a **file**. In normal parlance, the term file is used to indicate a collection of data. In MSX–BASIC, a file may be taken to mean any peripheral device.

There are three main processes that may be identified in file processing:

1. Preparation of the device to be treated as a file.
   MSX–BASIC needs to set aside areas of memory as **buffers** before file processing can begin. MSX–BASIC is said to **open** the file for further processing.
2. Input or output to the file.
3. Signalling the termination of a file's processing — this is termed **closing** the file.

In the case of the cassette recorder, the file may be opened in one of two modes: **input** or **output**. When opened, the file is given a **file number** which is used to reference the file in later processing. Up to sixteen files may be opened at once in MSX–BASIC. Two files are opened at once by default, and this number is increased by use of the MAXFILES statement. The statement:

```
10 MAXFILES = 4
```

sets aside enough buffer space for five files. If MAXFILES = 0 (for one file), then only program saving or loading operations may be carried out.

The file may also be named as you would name a program to be saved. This name can be a string of up to six characters, in which case the file would be assumed to be a cassette file. The name may also include the name of a **device descriptor** (see Chapter 1).

There are variants of the INPUT commands which support input from files, and also complementary output instructions which are variants of PRINT. Everything is written to cassette as ASCII data.

The statement CLOSE when used with a file number ends the processing of that particular file; if used alone it terminates processing of *all* open files. The END statement also closes all open files.

A cassette file is closed by writing the character given by a CTRL-Z sequence to tape. Any input operations must monitor for the occurrence of this special character which marks the end of the file. The EOF( ) function checks for this character for you — returning the value −1 when the end-of-file marker is encountered. The argument required is the file number of the open file you wish to check.

The cassette motor may be turned on or off using the MOTOR command. MOTOR ON and MOTOR OFF do exactly what their names suggest. If MOTOR is given as a command, the status of the cassette motor is reversed: if on, it is turned off and vice versa.

Simple examples

Program 6.5 inputs a series of names, writes them to tape, and allows the same data to be read back and printed on the screen.

Program 6.5

```
10 REM *
20 REM * File I/O
30 REM *
40 CLS
50 PRINT "FILES":PRINT
60 PRINT "Opening File.."
70 PRINT "PRESS PLAY AND RECORD"
80 OPEN "CAS:TEST" FOR OUTPUT AS #1
90 PRINT "Input '*' to Finish"
100 PRINT
110 LINE INPUT "name ";A$
120 IF A$="*" THEN GOTO 150
130 PRINT#1,A$
140 GOTO 110
150 PRINT "Closing File.."
160 CLOSE#1
170 PRINT "Press Rewind and Press any Ke
y"
180 A$=INPUT$(1)
```

```
190 MOTOR
200 PRINT "Press a key to stop Rewind"
210 A$=INPUT$(1)
220 MOTOR
230 PRINT "Press PLAY then a key"
240 A$=INPUT$(1)
250 PRINT
260 OPEN "cas:TEST" FOR INPUT AS #1
270 IF EOF(1) THEN GOTO 310
280 INPUT#1,A$
290 PRINT A$
300 GOTO 270
310 PRINT "All data Read"
320 CLOSE
330 END
```

Remember that programs may also be SAVEd to tape in ASCII format, and so read into programs as file data. There is also scope to write data to tape which may later be LOADed as a BASIC program. This technique is used to produce graphics programs in Chapter 9. Program 6.6 simply reads in and prints an ASCII-saved program from tape using the LINE INPUT# statement.

Program 6.6

```
10 REM *
20 REM * Program Read and Echo
30 REM * Must be saved in ASCII
40 REM * format; i.e with SAVE
50 REM *
60 SCREEN 0
70 INPUT "Program name: ";N$
80 OPEN "cas:"+N$ FOR INPUT AS #1
90 IF EOF(1) THEN 130
100 LINE INPUT#1,A$
110 PRINT A$
120 GOTO 90
130 PRINT
140 PRINT "End of Program File"
150 CLOSE#1
160 END
```

Writing records

If we create a file which contains a list of people's names and the cities where they live, we could write these **records** to tape. Each record would contain two **fields**: a name field and a city field.

The data is written to tape separated by commas, so allowing the data to be retrieved using a single INPUT# statement at a later date. Program 6.7 shows the technique at work.

Program 6.7

```
10 REM *
20 REM * File Output as a Record
30 REM *
40 CLS
50 REM *
60 REM * Output Routine
70 REM *
80 PRINT "Opening NAMES File..."
90 OPEN "cas:NAMES" FOR OUTPUT AS #1
100 PRINT "Name: (* to terminate):"
110 INPUT N$
120 IF N$="*" THEN 200
130 PRINT "City:"
140 REM *
150 REM * Input Routine
160 REM *
170 INPUT C$
180 PRINT#1,N$;",";C$
190 GOTO 100
200 CLOSE#1
210 CLS
220 PRINT "To retrieve, Rewind tape"
230 PRINT "Press a key when ready"
240 PRINT
250 A$=INPUT$(1)
260 OPEN "cas:NAMES" FOR INPUT AS #1
270 IF EOF(1) THEN 310
280 INPUT#1,N$,C$
290 PRINT N$,C$
300 GOTO 270
310 PRINT "End of File"
320 CLOSE#1
330 END
```

File processing with other devices

The other devices that may be treated as files are of the output only type. (The graphics screen is dealt with in Chapter 8.) These files do not have to be OPENed in input or output mode, they need only be given a file number. The following statement opens the printer as a file:

```
10 OPEN "LPT:" AS # 1
```

As they are output-only, it follows that INPUT commands will not work with these devices. Program 6.8 writes data to one of three devices. This technique is used in the program generator of Chapter 9.

Program 6.8

```
10 REM *
20 REM * Devices as Files
30 REM *
40 CLS
50 MAXFILES=3
60 OPEN "CAS:" FOR OUTPUT AS #1
70 OPEN "LPT:" FOR OUTPUT AS #2
80 OPEN "CRT:" FOR OUTPUT AS #3
90 PRINT "1. Save Data to cassette"
100 PRINT "2. List Data to Printer"
110 PRINT "3. List Data to Screen"
120 PRINT
130 INPUT "Option";N
140 FOR I=1 TO 10
150 READ A$
160 PRINT #N,A$
170 NEXT
180 CLOSE
190 DATA Nut,Bolt,Sprocket,Spanner,Wrenc
h
200 DATA Hammer,Saw,Axe,Nails,Grips
```

File handling is useful when data is to be stored on a long-term basis, or when the memory available to arrays is limited. The sequential nature of cassette files makes them inconvenient for use in applications like telephone and address books. The storage needed for such an application is likely to be vast, thus excluding the use of an array, and is one example where a good address book is certainly more convenient and faster than a cassette-based computer equivalent could ever be. Updating a single record in such an address book would require the processing and rewriting of the entire data file.

Summary

Device descriptors

CAS: Cassette recorder
LPT: Printer (output)
CRT: Text screen (output)

Array statements

DIM ⟨variable name⟩(Max subscript [,⟨max subscript⟩]. . .) [, ⟨variable name⟩(. . .), . . .]

ERASE [⟨array variable name⟩,[⟨array variable name⟩]. . .]

File handling commands

OPEN "⟨device descriptor⟩ [⟨file name⟩]"
[FOR ⟨INPUT ¦ OUTPUT⟩ AS # ⟨file number⟩]

CLOSE [#⟨file number⟩]

INPUT #⟨file number⟩,⟨variable name⟩ [separator ⟨variable name⟩]. . .

LINE INPUT #⟨file number⟩,⟨variable name⟩ [separator ⟨variable name⟩]. . .

PRINT #⟨file number⟩, [⟨print expression⟩]

PRINT #⟨file number⟩, USING ⟨format string⟩;[⟨print expression⟩]

Associated statements and functions

MAXFILES = ⟨maximum number of open files permitted⟩

EOF (⟨file number⟩)

MOTOR [⟨ON ¦ OFF⟩]

7 MSX sound features

The information I have given on MSX–BASIC so far is, for the greater part, applicable to most dialects of BASIC. When moving into areas of programming such as sound and graphics, implementations of BASIC differ widely. The features of the language depend on the sophistication of the specialist hardware present. In this chapter and those that follow, I'll be looking more closely at these dedicated devices.

The use of sound

A judicious use of sound can spice up many a program. At its simplest, sound generation is a good way of indicating that a computer program is alive and responsive. A beep can alert a user to the presence of an input error, or provide confirmation of a keypress.

To games afficionados, there are explosions and laser zaps, and often a piece of music playing mercilessly in the background. Surprisingly, microcomputers are not in themselves very good musical instruments, being rather limited in the range and quality of the sounds they produce. It is also much easier to compose using conventional instruments. However, for those prepared to devote a reasonable amount of time and patience, MSX–BASIC has a rewarding repertoire of musical features.

The MSX sound chip is the General Instruments AY–3–8912 Programmable Sound Generator (PSG for convenience). It is a dual function device in that it also handles input and output for the joystick ports. The chip can produce a range of notes over 8 octaves, with three independent channels of sound.

In addition, a sound effect in the form of noise may be added to any of these sound channels. MSX–BASIC provides two means of controlling sound, one of which is clearly music oriented; the other is more primitive, but allows a greater variety of sound production.

The PLAY command

PLAY is the music-oriented BASIC feature. Note that in all the program examples in this section, you are advised to use the command BEEP

before running a program. BEEP effectively initializes the sound chip by resetting its default values.

The sound generated by PLAY depends on a list of subcommands which are supplied within a character string. These subcommands are collectively termed the **Music Macro Language** (or MML). Pitches, note lengths, tempo, and volume are a few of the options which may be defined. All the language features available are covered in the sections that follow.

Setting pitch

As mentioned earlier, the note range spans 8 octaves, a total of 96 notes. The simplest way of playing a note is to use PLAY with the name of a note. The MML names notes after standard musical notation — C, D, E, F, etc. Sharps and flats (the **semitones**, or if you prefer, the black notes on a piano) are given differently: C# or C+ represents C sharp, and B− is B flat. The following example may be entered in direct mode to show you how this works.

```
PLAY "CC+DD+EFF+GG+AA+B"
```

The subcommand string could also have been given as a string variable as shown in this routine:

```
10 A$ = "CC+DD+EFF+GG+AA+B"
20 PLAY A$
```

Notice that only 'legal' notes are allowed. For example B# and E# do not exist in normal musical notation and will generate error messages.

We haven't yet specified the octave within which the string of notes is to play. The default setting is the fourth octave and octaves range from C to B inclusively. To change the octave value we use the O subcommand. Along with the O is an octave number ranging from 1–8, e.g., O5 or O3. The next example plays a simple sequence using three octaves.

```
PLAY "O1CDEFGO4CDEFGO8CDEFG
```

The simple note sequences given so far have tumbled out continuously, but we can also set **rests** — periods of silence. A rest is entered by giving the letter R in a string as below:

```
PLAY "O4CRO5CRO6C"
```

The final means of specifying pitch is by using the N subcommand. Instead of a note name, a note number between 0 and 96 is used, with 0

representing a rest, 1 equivalent to 'O1C', 2 being 'O1C+', etc. Again, another simple example.

```
PLAY "N1N48NØN96"
```

Altering note length and tempo

All the notes (and rests) in the previous examples have been of the same length. If we want to alter the length some basic musical knowledge is required. The MML uses standard musical terminology for note length, but the notation differs. Note length is set by the L command followed by a value from 1 to 64. MSX–BASIC assumes an initial note length of L4, which is a crochet. Figure 7.1 illustrates the relationship between MML note lengths and musical symbols for quavers, crotchets, minims, etc.

This is not the only means of designating note and rest length. You may also set it for each note, on an individual basis, by adding the value

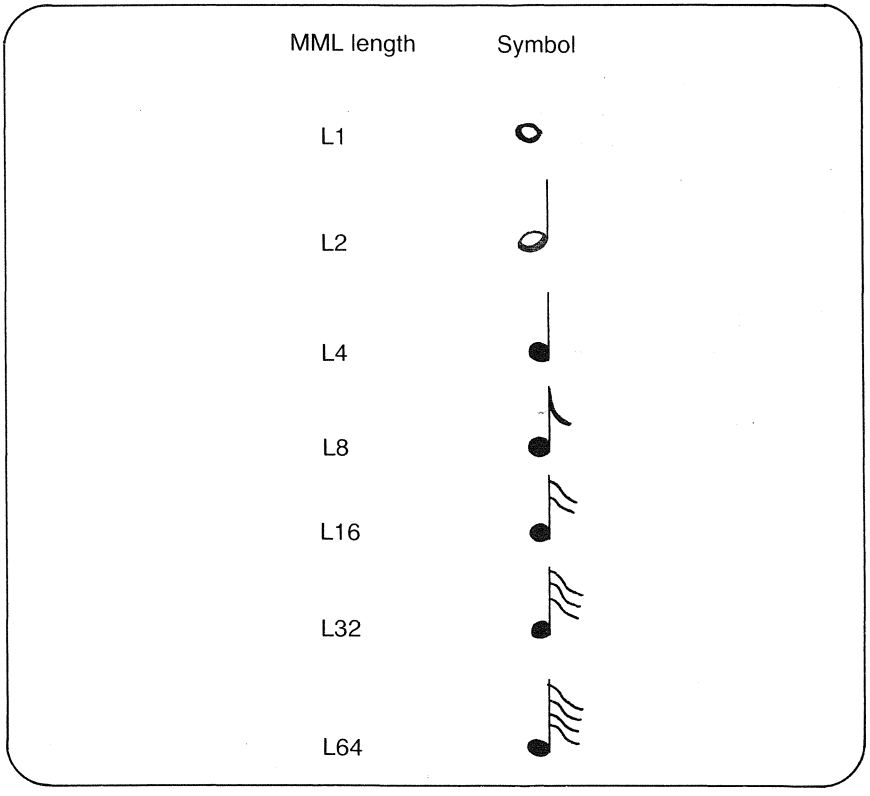


Figure 7.1 MML note lengths and musical symbols

of a note length to a note name. For example:

```
PLAY "C4R4D4C8R16D16"
```

is equivalent to:

```
PLAY "L4CRDL8CL16RD"
```

Another feature of MML is the ability to play what are known as dotted notes — where the duration of a note is increased by half of the value set by the L command. For example:

```
PLAY "CRC.RC. . "
```

What we have been doing with the L command is to set the length of the notes in relation to each other. For example, a semibreve (or whole note) given by L1 is twice as long as a minim, which is given by L2, and four times as long as a crotchet (L4). Dotted rests may also be played using this notation.

We can alter the actual length of all the notes (without changing their length in relation to each other) by altering the **tempo** of the music. This subcommand is given as T followed by a value between 32 and 255, and expresses the number of crotchets that may be played in one minute. As with other MML commands, there is a default value, in this case it is T12Ø — 12Ø crotchets per minute (or if you prefer, 3Ø whole notes per minute). So, the higher the value of the T subcommand, the faster the sequence is played. Program 7.1 illustrates the effect of varying the tempo.

Program 7.1

```
10 REM *
20 REM * Tempo Alteration
30 REM *
40 A$ = "O4L8DF#ARAF#D"
50 FOR I = 1 TO 4
60 READ T$ : PLAY T$
70 PLAY A$
80 NEXT
90 DATA T32,T64,T128,T255
```

Volume variation and sound effects

To add more expression to a piece of music, we can alter the volume of the sounds produced. The crudest means of achieving a variation in volume is to use the MML's V (volume) command. The volume may be

set at any value from Ø to 15. This range encompasses VØ — very quiet, to V15 which is positively raucous. I'll not demonstrate it here as it works in exactly the same way as other MML commands. In this case the default value is V8.

A much more interesting feature offered by the MML is the range of eight **sound envelopes**. An envelope dictates the shape of a sound in terms of volume (amplitude) variation. For example, a note which rises in volume then abruptly cuts off sounds like a note played in reverse. Warbling sounds, blips and pings are a few of the interesting sounds that may be created. Only one of the eight envelopes may be used at any one time, however, and it is impossible to have a different envelope selected for each of the sound channels.

To set up an envelope, two pieces of information may be supplied. The first of these is the *shape* of the envelope, which is specified by the S command. The second is the **modulation cycle** — the frequency of the modulation effect on a note — given by the M command. Figure 7.2 illustrates the sound envelopes that may be used. Note that although the S command may have a value between Ø and 15, some of these values produce identical envelopes (the reason for this is examined more closely in the section of SOUND).

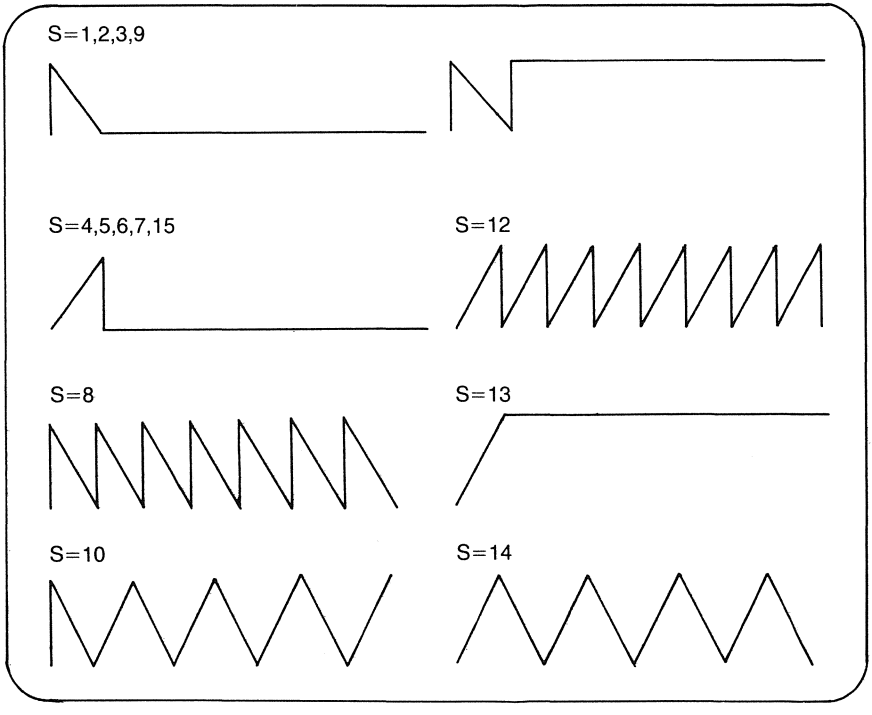


Figure 7.2 Sound envelopes

Program 7.2 produces examples of each of the eight envelopes.

Program 7.2

```
10 REM *
20 REM * Eight Sound Envelopes
30 REM *
40 CLS
50 A$ = "M5000O4L4DF#ARAF#D"
60 FOR I = 1 TO 8
70 READ S$ : PRINT "Sound Envelope ";S$
80 PLAY S$
90 PLAY A$
100 FOR J = 1 TO 2000 : NEXT J
110 NEXT  I
120 END
130 DATA S1,S4,S8,S10,S11,S12,S13,S14
```

Now let's see the effect of altering the modulation cycle. Using the S8 envelope, we can see how three diffferent values of M produces totally different sounding notes in Program 7.3.

Program 7.3

```
10 REM *
20 REM * Modulation Effects
30 REM *
40 A$ = "S8O4L12CGAO5CDGA"
50 PLAY "M200" : PLAY A$
60 PLAY "M1000" : PLAY A$
70 PLAY "M5000" : PLAY A$
80 END
```

The smaller the value of the M command, the greater the modulation frequency. Of the three sounds, the lower value for M produces a rapid pulsing of the notes, with the higher values giving rise to a more leisurely 'beating' effect. When using the sound shaping commands, you'll find that the V subcommand turns the effect of an envelope off. The S command has exactly the same effect on a previously issued V command. Programs 7.4 and 7.5 offer more sound shaping examples.

Program 7.4

```
10 REM *
20 REM * Sounds One
30 REM *
```

```
40 PLAY "T250L4"
50 FOR I = 1 TO 4
60 PLAY "S14M40004CAB"
70 PLAY "S8M60006C03C02C"
80 PLAY "S14M40004C05BA04CA"
90 PLAY "S8M60004C02C03C"
100 NEXT I
110 END
```

Program 7.5

```
10 REM *
20 REM * Sounds Two
30 REM *
40 PLAY "T120L2"
50 PLAY "S13M900003F+RGACD"
60 PLAY "S13M900003F+RGAC02A"
70 FOR I = 1 TO 4
80 PLAY "S1M1500L6405F+GAC04A"
90 PLAY "S11M900L6403F+GAC02A"
100 NEXT I
110 END
```

Sound shaping and modulation complete the major subcommands in the Music Macro Language, which are all summarized at the end of this chapter. This is enought material covered so far to allow simple tunes to be written, but there are still a few more features of PLAY and the MML to be discovered.

Using variables and substrings

All the MML commands have used constant values to set modulation, pitch and so forth. A nice feature is the ability to include variables into MML substrings. This is done by giving a subcommand followed by '=', a variable name and a semicolon. For example, V=I; (the semicolon is vital to avoid errors). In Program 7.6 the volume of a channel is turned up using a FOR . . . NEXT loop. This technique may be used with all the MML subcommands discussed so far.

Program 7.6

```
10 REM *
20 REM * Altering the Volume
30 REM *
40 FOR I = 15 TO 1 STEP -2
```

```
50 PLAY "V=I;T255L6403D02D05CDEGFE03ADEC
D"
60 NEXT I
70 END
```

Another possibility is to include a substring. Suppose that a musical phrase occurs many times within a piece that you are transcribing. Instead of writing out this phrase many times over, it may be declared a string variable, and executed as a substring when required. The final MML command is X — execute substring. It is followed by a string variable name and a semicolon (e.g. XA$;) and is used in Program 7.7.

Program 7.7

```
10 REM *
20 REM * Substring Example
40 REM * A$ is the substring here
50 REM *
60 A$ = "T18003F32F#32G4"
70 PLAY "T180XA$;XA$;F#4F4D#4C2.02G2"
80 PLAY "RXA$;XA$;F#4F4D#4C2."
90 END
```

Multiple voices and buffering

All the music strings played in previous examples have used only one channel of sound. By using all three sound channels, we can produce chords (all three channels played simultaneously) or write three-part music, with melody and accompaniment lines. To play all the voices at once, three string expressions are given to PLAY, with a comma to separate each string. We'll start by playing a chord.

```
PLAY "T120O4D4", "T120O4F#","T120OA4"
```

This is a relatively straightforward example as the notes for all three channels are of the same length. When using more complex strings of commands, a problem may arise. This is principally concerned with the way the PLAY command works. The BASIC interpreter has to analyse the strings for each channel. If one channel has a longer list of MML commands than another, it will take longer to interpret, and this gives rise to a slight delay. Eventually, the different channels will drift hopelessly out of synchronization. Shorter strings are not only easier to work with, but go a long way to minimising this problem.

Writing music for three channels requires a great deal of care taken with timing. At first, it is often better to have all the notes for each

channel of a PLAY command add up to the same length, e.g., a semibreve in one, two minims in another an eight quavers in the third. When composing or transcribing, a great deal of forethought is required when setting up MML strings.

You may notice that the 'Ok' prompt appears, even though a piece of music is still playing. Part of the MSX RAM is set aside as a music queue. Once an MML string has been translated, the sound chip instructions are stored in this queue, and periodically executed. This systems has its advantages. Music can be played continuously in the background, while your program gets on with output to the screen, maths processing or other such tasks. Its disadvantage is seen when trying to get the music to play in step with other events in a program, such as a response from the keyboard, or items displayed on the screen.

The program 7.8 highlights this effect of the music queue. It attempts to play ten notes in sequence, displaying each note name as it is played.

Program 7.8

```
10 REM *
20 REM * Out of Step
30 REM *
40 CLS
50 PLAY "T25505L16"
55 FOR I = 1 TO 10
60 READ A$
65 PRINT "Note Name ";A$
70 PLAY A$
80 NEXT I
90 END
100 DATA C,A,E,F,G,B,D,G,E,C
```

As you can see, note names appear in a batch, totally out of step with the note sequence being played. A simple solution is to add a delay loop into the major FOR loop. The length of the delay will need to be 'tuned' to the values of tempo and note length selected.

MSX–BASIC does allow the programmer a limited view of what is happening in the music queue. The only intrinsic function offered for music use is the rather confusingly named PLAY. This function returns a value which indicates either that the music queue is empty (no music playing) or that it still has some data to play. One of four arguments may be supplied to PLAY. If values 1 to 3 are given, the status of the corresponding channel's music queue is given. Supplying Ø checks all the channels. PLAY returns −1 (true) if a channel is playing, and Ø (false) otherwise.

We can use this function to set up a piece of background music. By

periodically checking the status of a music channel, we can see when it is empty and add more music data. The rate at which PLAY is polled needs to be selected with some care if gaps in the music are to be avoided.

A background music piece is set up in Program 7.9 while the major task in the foreground is just producing random character strings. The results are not quite perfect, but they give some idea of what may be done.

Program 7.9

```
10 REM *
20 REM * Background Music
30 REM *
40 CLS : KEY OFF
50 R = RND(-TIME)
60 REM *
70 REM * Define Music Strings
80 REM *
90 A$ = "O5DF#GA"
100 B$ = "O5GBO6CD"
110 C$ = "O5AO6C#DE"
120 D$ = "XA$;XC$;XB$;XA$;XB$;XC$;"
130 PLAY "V10T120L16"
140 PLAY D$
150 IF NOT(PLAY(1)) THEN  140
160 REM *
170 REM * Generate Random Strings
180 REM *
190 W$ = ""
200 FOR I = 1 TO 5
210 X=INT(RND(1)*26)
220 W$ = W$ + CHR$(65+X)
230 NEXT I
240 PRINT W$,
250 GOTO 150
```

Finally, Program 7.10 offers an example of three voice music which brings together most of the features of the Music Macro Language, including sound shaping, variables and substrings.

Program 7.10

```
10 REM *
20 REM * Three Voice Example
```

```
30 REM *
40 REM * Music String Definition
50 REM *
60 A$ = "O2G4F8C8R8D8C8D8"
70 B$ = "O6D4R8O4D4A8O5D8O6A8"
80 C$ = "O5B8G8D4A4O5C4"
90 D$ = "O5F16G16A16B16O6C16"
100 E$ = "O3G4F8C8R8D8C8D8"
110 F$ = "O5F16G16B16C16O5D4"
120 G$ = "O2D4R8D8R8D8R4F8F8R8F8R8F8R8F8
G4R8"
130 H$ = "O3D4R8D8R8D8R4F8F8R8F8R8F8R8F8
G4R8"
140 I$ = "O3G4R8O2G4E8O2G8O3E8"
150 J$ = "O2G4F8C8R8D8O4C8O3B8"
160 K$ = "G8R8G8R4G4R8G8R8G8R4"
170 REM *
180 REM * Chord Introduction
190 REM *
200 PLAY "O4T100L1S13M23000G","O5T100L1S
13M23000F","O3T100L1S13M23000D"
210 PLAY "G","B","O5D"
220 PLAY "E","G","O4A"
230 PLAY "O3E","O5G","O4B"
240 PLAY "O3D","O5A","O4F#"
250 PLAY "O5G","O4E","O3D"
260 PLAY "M30000O4D","M30000O5D","M30000
O3D"
270 REM *
280 REM * Set Volume and Turn
290 REM * Modulation Off
300 REM *
310 PLAY "V10T150","V10T150","V11T150"
320 FOR I = 4 TO 6
330 J = I+1
340 L$ = "L24O=I;DEFGO=J;CO=I;BAG"
350 PLAY L$,L$,L$
360 NEXT
370 FOR I = 1 TO 2
380 PLAY B$,B$,B$
390 NEXT I
400 FOR I = 1 TO 2
410 PLAY B$
420 PLAY A$,E$,B$
```

```
430 NEXT
440 FOR I = 1 TO 2
450 PLAY A$,B$,C$
460 PLAY A$,B$,D$
470 PLAY A$,B$,C$
480 PLAY J$,B$,F$
490 NEXT I
500 PLAY G$,H$,C$
510 PLAY K$,K$,F$
520 PLAY G$,H$,"XB$;XC$;"
530 PLAY K$,K$,F$
540 PLAY G$,H$,"XB$;XC$;"
550 PLAY "R4","R4","R4"
560 REM *
570 REM * Fade Away to Finish
580 REM *
590 FOR I = 11 TO 0 STEP -2
600 PLAY "R","R","T255V=I;XF$;"
610 PLAY "R","R","XC$;"
620 NEXT I
630 END
```

The SOUND command

The Music Macro Language is fine if all you want to do is produce notes which lie in the standard (Western) musical scale. To create sound effects, such as the sound of surf and sirens, a more exacting control over the PSG is required. The SOUND command allows fairly precise selection of frequencies, addition of a noise sound effect, to all or individual channels, as well as sound shaping. As with a number of things in life, the more you are offered, the harder you have to work for it. SOUND is no exception in this respect.

Command format and register set

After the wealth of options available from PLAY, the austerity of SOUND may come as a bit of a shock. The format of the command is:

SOUND ⟨PSG register number⟩,⟨value to be written⟩

MSX–BASIC allows values to be written to fourteen of the chip's registers — there are more, but they are concerned with joystick I/O and not sound production. The contents of all (or a combination) of these registers determines the nature of the sound generated. Table 7.1

describes the function of these registers and the maximum significant value that may be written to each register.

Table 7.1

Register	Purpose	Values
Ø, 2, 4	Fine tune A, B, C	Ø–255
1, 3, 5	Coarse tune A, B, C	Ø–16
6	Noise frequency	Ø–31
7	Mixer register	Ø–63
8, 9, 1Ø	Volume of A, B, C	Ø–16
11	Modulation fine tune	Ø–255
12	Modulation coarse tune	Ø–255

The way each of these registers is used will now be discussed, with a number of examples. SOUND is a BASIC command which invites experimentation. It is worth remembering: should you feel you've lost control at any time, BEEP resets everything.

The mixer/channel select register

Register 7 is possibly the most important of the PSG's registers. Its main purpose is to select which channels are to produce sound. It also allows a noise effect to be added to one or all of the channels. The noise effect is a hissing sound, which is useful for producing effects like steam engines, the sound of surf, etc. To see how the register operates, it is best viewed as a binary memory location. The lower six bit positions are significant in controlling the sound channels. The format of the register is given in Figure 7.3.

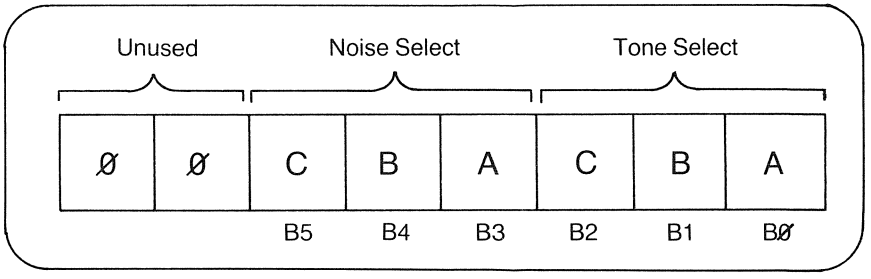


Figure 7.3 Register 7 – mixer/channel select register

Contrary to the binary numbers seen so far, in this case, a 'Ø' in a bit position represents ON (low true) and a '1' denotes OFF. Referring to Figure 7.3, if the bit positions BØ to B2 are all zero, then all three channels will produce a tone. If all six bits (BØ–B5) are zero, tone and noise effect are selected for all three channels. Here are some examples

of binary values and the effect they have on the mixer register:

&B00111000	(56)	Select tone only from all channels.
&B00000111	(07)	Noise only from all channels.
&B00110110	(54)	Noise and tone from channel 1.
&B00001100	(12)	Noise from channel 3, noise and tone from channel 2, tone only from channel 1.

Setting frequency

Registers 0 to 5 control the frequency of the three sound channels. For example, the combination of the values in registers 0 and 1 determines the frequency of the tone from channel 1. All frequencies are produced by division of the SOUND generator's clock frequency — 1789800 Hertz (cycles per second) — approximately half that of the Z80's (which is 3579545 Hertz).

To produce a given frequency, we need to use a conversion function. The general equation for the sound chip is given by:

$$\frac{1789800}{16 \times (\text{Hertz})} = 256 \times (\text{FT} + \text{CT})$$

Where FT is the fine tune register value (registers 0, 2, or 4), and CT the coarse tune register value (registers 1, 3, or 5). By juggling the equation in a BASIC program, the values for the tuning registers for any given frequency can be calculated — as shown in Program 7.11

Program 7.11

```
10 REM *
20 REM * Play Note of a Given Freq.
30 REM *
40 SCREEN 0 : KEY OFF
50 CLS
60 PRINT
70 PRINT "Frequency Calculation"
80 PRINT
90 PRINT
100 INPUT "Input Frequency (Hz) ";HZ
110 IF HZ<28 OR HZ>55932! THEN 50
120 PRINT
130 TMP = 1789800#/(16*HZ)
140 CT = INT(TMP/256)
150 FT = TMP MOD 256
```

```
160 PRINT "Frequency";TAB(17);":";HZ;
"Hz"
170 PRINT "Fine Tune Value  :";FT
180 PRINT "Coarse Tune Value:";CT
190 PRINT
200 PRINT "Another Frequency (Y/N)";
210 A$=INKEY$ : IF A$="" THEN 210
220 IF A$="Y" OR A$="y" THEN 50
230 END
```

We can also convert the values of a fine tune/coarse tune register pair to find the frequency in Hertz, as in Program 7.12.

Program 7.12

```
10 REM *
20 REM * Convert Register Values
30 REM * To Frequencies
40 REM *
50 DEF FNFC(A,B)=INT((1789800#/((256*A)+
B))/16)
60 CLS
70 PRINT
80 PRINT "Register to Freq. Conversion"
90 PRINT
100 INPUT "Fine Tune Value  : ";FT
110 IF FT < 0 OR FT > 255 THEN BEEP
:GOTO 100
120 INPUT "Coarse Tune Value: ";CT
130 IF CT < 0 OR CT > 15 THEN BEEP
:GOTO 120
140 REM *
150 REM * Protect against Zero Division
160 REM *
170 IF CT = 0 AND FT=0 THEN BEEP : GOTO
100
180 PRINT
190 PRINT "Frequency ";CHR$(247);FNFC(CT
,FT);" Hz"
200 PRINT
210 PRINT "Another Set of Values (Y/N)";
220 A$ = INKEY$ : IF A$ = "" THEN 220
230 IF A$ = "y" OR A$ = "Y" THEN 60
240 END
```

Appendix 5 gives the frequency and register values required to produce the range of notes available from the MML.

Varying the noise pitch

The frequency of the noise sound effect may be changed by writing a value to register 6. The general equation that determines the value of register 6 is:

$$\frac{1789800}{16 \times (\text{Hertz})} = \text{Value of register 6}$$

Only register 6 values between 0 and 31 are significant here.

Volume variation and modulation effects

Registers 8, 9 and 10 control the volume of channels 1 to 3 respectively. Writing values to these registers has the same effect as the MML V command. However, when the value 16 is written to a volume register, any sound envelope that has been set up may be brought into use. Figure 7.4 shows a binary representation of a volume register.

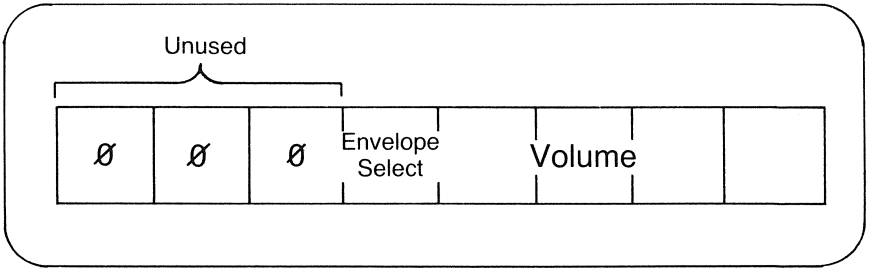


Figure 7.4 The volume control registers

It is now easy to see why the MML's S and V commands seemed antagonistic. When as S command is used, 16 is written to the appropriate volume register. This sets B4 to '1' and all lower bits to '0' — so cancelling out any volume setting. V can only use values between 0 and 15, so B4 will always be reset.

The envelope shapes are selected by writing a value to register 13. Only the lower four bits are significant. The reason why some values produce identical envelope shapes is due to the fact that the lower two bits of register 13 are not significant in some cases. All values over 7 are sure to give unique sound shapes.

The modulation frequency for an evelope is determined in the same way as for sound frequency. In this case, the fine tune is register 11, and the coarse tune is register 12. Program 7.11 for deriving register values from frequencies may be modified quite easily.

There are two distinct classes of sound envelope. The 'continuous' type will, as its names suggests, modulate the sound continuously. The continuous envelopes are given by the values, 8, 10, 12 and 14. The 'one-shot' type will modulate the sound for only one cycle. To trigger a one-shot, the envelope's number must be written to register 13 each time the sound is required.

SOUND programs

In this section, I'll put all the above theory into practice with six different sound programs. A description and explanation will accompany each.

Program 7.13 Simple siren

In this program, the aim is to produce the sound of a 'wailing' type of siren, with a steady rise and fall in pitch. This is achieved by playing a tone, and altering the pitch using a fine tune register. In this case, the pitch rises steeply, then falls back gradually to its original level.

```
10 REM
20 REM * Siren
30 REM
40 SOUND 0,255
50 SOUND 1,0
60 SOUND 8,8
70 SOUND 7,&B00111110
80 REM *
90 REM * Rising Tone
100 REM *
110 FOR I = 252 TO 170 STEP -.2
120 SOUND 0,I
130 NEXT
140 REM *
150 REM * Falling Tone
160 REM *
170 FOR I = 170 TO 252  STEP .1
180 SOUND 0,I
190 NEXT
200 GOTO 110
```

Program 7.14 Using noise

This program produces noise from one channel, and continually sweeps the frequency from high to low by altering the contents of register 6.

```
10 REM *
20 REM * The Noise Effect
30 REM *
40 SOUND 6,0
50 SOUND 7,&B00000111
60 REM *
70 REM * Decrease Noise Frequency
80 REM *
90 FOR I = 0 TO 31
100 SOUND 8,10 : SOUND 9,10 : SOUND
10,10
110 SOUND 6,I
120 FOR J = 1 TO 50 : NEXT J
130 SOUND 8,0 : SOUND 9,0 : SOUND 10,0
140 FOR J = 1 TO 25 : NEXT J
150 NEXT
160 GOTO 90
```

Program 7.15 Sound effects

A modulation effect is added to a basic tone. When a key is pressed, the modulation cycle is increased via register 12, and random frequency values alter the fine tune register. This is a good example of the startling sorts of sounds that may be produced.

```
10 REM *
20 REM * Alien Noises
30 REM *
40 SOUND 0,252
50 SOUND 1,0
60 SOUND 8,16
70 SOUND 11,200
80 SOUND 12,2
90 SOUND 13,10
100 SOUND 7,&B00111110
110 PRINT "Press Any Key.."
120 A$ = INPUT$(1)
130 REM *
```

```
140 REM * Increase Mod. Cycle
150 REM * Send Random Freq. Values
160 REM *
170 SOUND 12,1
180 SOUND 0,RND(1)*255
190 GOTO 180
```

Program 7.16 Reversed sounds

Another envelope shape is selected, which slowly rises in volume then cuts off abruptly. All three sound channels are used, with the frequency of one 'detuned' slightly. This detuning introduces a phasing effect into the sound produced.

```
10 REM *
20 REM * Reversed Sounds
30 REM *
40 R = RND(-TIME)
50 FOR I = 0 TO 13
60 READ A : SOUND I,A
70 NEXT I
80 FOR I=1 TO 1000:NEXT
90 X=RND(1)*250
100 IF X<30 THEN 90
110 Y=RND(1)*10+1
120 SOUND 0,X : SOUND 1,Y
130 SOUND 2,X : SOUND 3,Y
140 SOUND 4,X+5 : SOUND 5,Y
150 SOUND 13,13
160 GOTO 80
170 DATA 62,2,60,2,60,2,0,56,16,16,16,
120,30,13
```

Program 7.17 Percussive effects

The simulation of a snare drum is created by using the noise effect and a one-shot envelope. A bass note is produced by using the same envelope to modulate a tone instead of noise. Depending on which key is pressed, the program will produce either a bass sound or the snare. Register 7 provides the means of selecting the sound produced.

```
10 REM *
20 REM * Snare Drum and Bass
30 REM *
```

```
40 SCREEN 0,0,0
50 FOR I = 0 TO 12
60 READ S
70 SOUND I,S
80 NEXT I
90 PRINT "Press 'n' to Sound Drum"
100 PRINT "Press 'b' to Sound Bass"
110 A$ = INPUT$(1)
120 IF A$ = "b" THEN SOUND 7,&B00111000:
SOUND 13,1
130 IF A$ = "n" THEN SOUND 7,&B00000111:
SOUND 13,1
140 GOTO 110
150 DATA 140,7,140,7,140,7,15,255,16,16,
16,190,7
```

Program 7.18 Bell chimes

The final program of this chapter simulates the characteristic ring of a bell or chime. Here, the bell sound is produced by using a range of three frequencies which are not quite in tune with each other. A one-shot envelope creates the impression of a striking action on the bell, and the ringing decay is achieved by using quite a long modulation cycle. By altering the pitch frequencies used, other bell-like sounds can be created, such as a triangle or even the beating of a metal pipe.

```
10 REM *
20 REM * Bell Chimes
30 REM * Initialize Sound Chip
40 REM *
50 FOR I = 0 TO 12
60 READ S : SOUND I,S
70 NEXT I
80 WIDTH 40 : CLS : KEY OFF
90 PRINT "BELL CHIMES" : PRINT
100 SOUND 7,56
110 REM *
120 REM * Trigger Chimes
130 REM *
140 FOR I = 1 TO 12
150 SOUND 13,0
160 PRINT I;TAB(1)
170 FOR J = 1 TO 1200 : NEXT J
180 NEXT I
190 PRINT : PRINT
```

```
200 PRINT TAB(1);"Twelve O'clock":PRINT
" and all's Well!
210 END
220 DATA 229,0,97,0,115,0,0,63,16,16,16,
190,120
```

Summary

Sound commands

BEEP

SOUND ⟨register number⟩,⟨value to be written⟩

PLAY ⟨string⟩[,⟨string⟩[,⟨string⟩]]

PLAY (⟨music channel⟩)

Play subcommands

C, D, E, F, G, A, B	Play note given by name.
N⟨n⟩	Play note of number ⟨n⟩.
R	Play a rest.
O	Set current octave.

Note and rest postfixes

+ or #	Play the note sharp.
−	Play the note flat.
.	Play note as a dotted note.
⟨n⟩	Play note with length ⟨n⟩.

Sound shaping

S⟨n⟩	Set envelope.
M⟨n⟩	Set modulation frequency.

Others

V⟨n⟩	Set current volume.
L⟨n⟩	Set current note length.
T⟨n⟩	Set tempo.
X⟨string variable⟩	Execute substring.

8 Introduction to graphics

MSX–BASIC provides the programmer with a healthy array of functions and commands for the production and manipulation of graphics data. These tools may be used for a number of applications, some of which I have summarized below.

Graphing When studying a large set of data, a *qualitative* rather than a *quantitative* assessment may be all that is needed to pick out a trend. Data can be presented in graphical formats such as histograms, pie charts, and pictograms. The computer's ability to marry mathematical calculation with graph drawing at reasonably high speed can make it a valuable tool in data analysis.

User interfaces A program can be made much easier to use by communicating with a user via graphics images. It is possibly easier to interpret a picture of a filing cabinet, than the phrase 'relational database'. Modern business computers are beginning to use this technique quite extensively. Although these are all 16-bit computers, some of their ideas can be incorporated in MSX–BASIC programs.

Games Producing scenarios for 'Zap-em!' type games must constitute the largest single application of computer graphics. Often, the backdrops for the action change, which may be seen to dramatic effect in games where the 'world' is a three-dimensional simulation. The computer's virtuosity in the animation of spaceships, missiles, monsters and other such antagonists is well known.

In this chapter and Chapter 9 we shall explore some of these graphics techniques.

The MSX–graphics screens

There are two graphics screens in MSX–BASIC. Both screens obey the convention for coordinate naming used by the text screens; i.e. (Ø,Ø) is in the top left-hand corner of the screen. The screens are dimensioned

256 × 192 — a total of 49152 addressable points, but they differ in **resolution**. The resolution of the screen defines the size of the smallest point (**pixel**) that may be displayed on the screen.

The low-resolution screen has a pixel size of 4 × 4 points. So a maximum of 64 × 48 individual pixels may appear on the screen at any one time. Therefore, setting a point at coordinate (128,96) produces the same effect as setting a point at coordinate (130,98).

With the high-resolution screen, the number of pixels matches the number of addressable points on the screen. Most of the example programs use the high-resolution mode.

A screen mode is selected by using the SCREEN command. The high-resolution mode is given by SCREEN 2, and low-resolution by SCREEN 3. Unlike the text modes, which once selected remain in that mode until another SCREEN command is issued, the graphics modes are volatile. If SCREEN 2 is given in command mode, all that will be seen is a flash, followed by the 'Ok' prompt. To maintain a graphics mode, some form of infinite loop is required to prevent a return to text mode.

Setting points

The BASIC command for putting a pixel on a screen is PSET (Point SET). There are two ways of specifying a coordinate of a point to be set:

Absolute A coordinate is given which directly references a point on the screen. This is by far the most common type of point addressing. The command has the syntax:

PSET (⟨X coord⟩,⟨Y coord⟩)[⟨colour⟩]

If negative numbers are given as absolute coordinates they are assumed to be zero; e.g., PSET (−45,20) is the same as PSET (0,20).

Relative The word STEP is given followed by a pair of coordinates. In this case, the coordinates specify a position relative to *the last point referenced.* So an offset can be given as a negative or positive integer. This form of PSET is expressed as:

PSET STEP (⟨X offset⟩,⟨Y offset⟩)[,colour]

These two ways of specifying coordinates are common to all MSX graphics commands.

Any valid integers may be used as coordinate values, however, only those in the range 0–255 for the X coordinate and 0–191 for the Y coordinate will have any effect. If real numbers are given, they will be rounded down to the nearest whole number.

As you can see from the syntax of PSET, colour may be specified as an option. If colour is not specified the point will be plotted in the current foreground colour. This is true of all the graphic commands except PRESET. The command PRESET (Point RESET) has exactly the same syntax as PSET, but, if no colour is specified PRESET plots the point in the current *background* colour. Programs 8.1, 8.2, 8.3 and Figure 8.1 illustrate these commands at work.

Program 8.1

```
10 REM *
20 REM * PSET and PRESET
30 REM *
40 COLOR 15,1,1
50 SCREEN 2
60 REM *
70 REM * Plot White Points
80 REM *
90 FOR X = 16 TO 240 STEP 4
100 FOR Y = 16 TO 176 STEP 4
110 PSET(X,Y)
120 NEXT Y
130 NEXT X
140 REM *
150 REM * Erase Them Using PRESET
160 REM *
170 FOR Y = 176 TO 16 STEP -4
180 FOR X = 240 TO 16 STEP -4
190 PRESET(X,Y)
200 NEXT X
210 NEXT Y
220 REM *
230 REM * Maintain Graphics Mode
240 REM *
250 GOTO 250
```

Program 8.2

```
10 REM *
20 REM * Relative Drawing with PSET
30 REM *
40 COLOR 15,4,4
50 SCREEN 2
60 REM *
70 REM * Set Start point for Drawing
```

```
80 REM *
90 PRESET (0,0)
100 REM *
110 REM * Draw Boxes Using Relative
120 REM * Point Setting
130 REM *
140 FOR I=1 TO 255:PSET STEP(1,0):NEXT
150 FOR I=1 TO 191:PSET STEP(0,1):NEXT
160 FOR I=1 TO 255:PSET STEP(-1,0):NEXT
170 FOR I=1 TO 191:PSET STEP(0,-1):NEXT
180 REM *
190 REM * Establish new start position
200 REM *
210 PRESET (60,30)
220 FOR I = 1 TO 4
230 FOR J = 50 TO 10 STEP -10
240 FOR K = 1 TO J
250 PSET STEP(1,0)
260 NEXT K
270 FOR K = 1 TO J
280 PSET STEP(0,1)
290 NEXT K
300 FOR K = 1 TO J
310 PSET STEP(-1,0)
320 NEXT K
330 FOR K = 1 TO J
340 PSET STEP(0,-1)
350 NEXT K
360 PRESET STEP (5,5)
370 NEXT J
380 NEXT I
390 GOTO 390
```

Program 8.3

```
10 REM *
20 REM * Low Resolution PSET
30 REM * Random Colour Mosaic
40 REM *
50 COLOR ,,15
60 SCREEN 3
70 FOR I = 0 TO 255 STEP 4
80 FOR J = 0 TO 191 STEP 4
90 PSET(I,J),RND(1)*16
```

```
100 NEXT J
110 NEXT I
120 GOTO 120
```

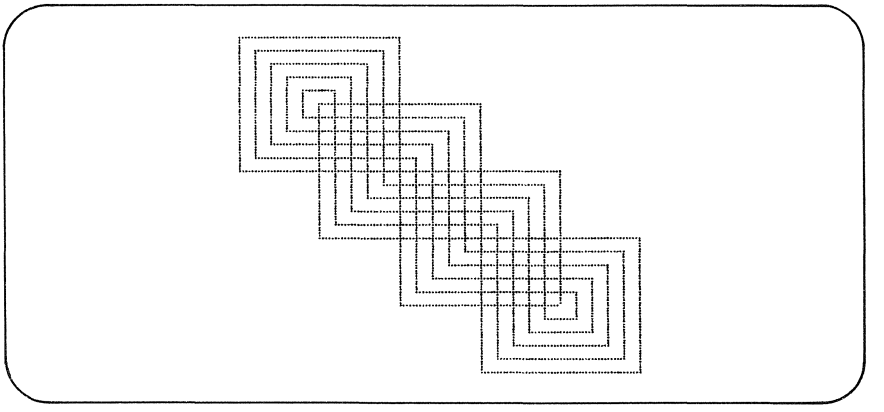


Figure 8.1 Typical output from Program 8.2

Line and box drawing

MSX–BASIC provides a fast line drawing command, with options that produce boxes. The command syntax is given by:

LINE [⟨coordinate specifier⟩]–⟨coordinate specifier⟩[,⟨colour⟩] [,⟨B ¦ BF⟩]

The two coordinates indicate the start and end points of the line to be drawn. Both these coordinates may be given in the relative form, i.e. with the coordinates prefixed with the keyword STEP. If the first coordinate pair is absent, the start point for the line will be taken as the last point referenced. Note that the '–' symbol must always be included, e.g.:

LINE –(128,96)

To draw boxes, the options B or BF are given at the end of the command. The B option produces a simple box, while BF draws a box and paints it. The start and end points given in the line command must be diagonally opposite each other if a box is to be drawn. All the possible forms of LINE are used in Programs 8.4 and 8.5; Figure 8.2 shows some typical results.

Program 8.4

```
10 REM *
20 REM * The LINE Statement
30 REM *
40 COLOR 15,4,4
50 SCREEN 2
60 REM *
70 REM * Draw Simple Lines
80 REM *
90 FOR I = 120 TO 170 STEP 5
100 LINE (5,I)-(I,I)
110 NEXT I
120 REM *
130 REM * Draw Boxes
140 REM *
150 FOR I = 10 TO 60 STEP 10
160 LINE (I,I)-(I+50,I+50),,B
170 NEXT
180 REM *
190 REM * Draw Filled Boxes
200 REM *
210 FOR I = 10 TO 125 STEP 35
220 LINE (170,I)-(220,I+25),,BF
230 NEXT
240 GOTO 240
```

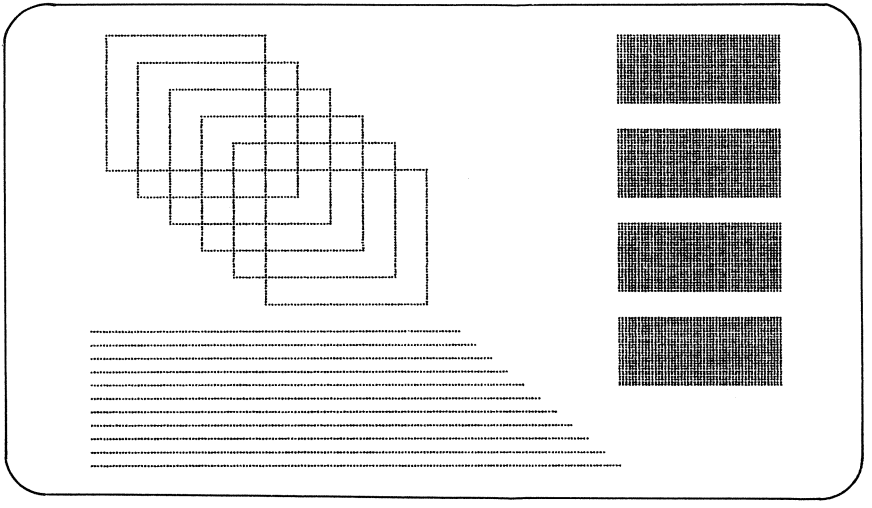


Figure 8.2 Typical output from Program 8.4

Program 8.5

```
10 REM *
20 REM * Relative LINE commands
30 REM *
40 COLOR 1,15,15
50 SCREEN 2
60 PSET(170,60)
70 FOR I = 5 TO 55  STEP 5
80 LINE -STEP(I,0)
90 LINE -STEP(I,I)
100 LINE -STEP(0,I)
110 LINE -STEP(-I,I)
120 LINE -STEP(-I,0)
130 LINE -STEP(-I,-I)
140 LINE -STEP(0,-I)
150 LINE -STEP(I,-I)
160 PRESET STEP(-7,-5)
170 NEXT
180 FOR I = 1 TO 500 : NEXT I
190 CLS
200 PRESET (70,30)
210 FOR I = 1 TO 30
220 LINE STEP(-45,2)-STEP(50,0)
230 NEXT
240 FOR I = 1 TO 30
250 LINE STEP(-55,2)-STEP(50,0)
260 NEXT
270 GOTO 270
```

Drawing ellipses and arcs

The CIRCLE command is the most complex command in MSX–BASIC. The format of the command is as follows:

CIRCLE ⟨coordinate specifier⟩,⟨radius⟩[,⟨colour⟩][,⟨start angle⟩] [,⟨end angle⟩][,⟨aspect ratio⟩]

The coordinate specifier denotes where the centre of an ellipse or arc is to be. Starting with the simplest form of the command, we can draw ellipses as in Program 8.6.

Program 8.6

```
10 REM *
20 REM * Circle Drawing
```

```
30 REM *
40 T=RND(-TIME)
50 COLOR 15,1,1
60 SCREEN 2
70 FOR I=1 TO 10
80 X=RND(1)*224 + 16
90 Y=RND(1)*160 + 16
100 FOR R=5 TO 20 STEP 2
110 CIRCLE(X,Y),R
120 NEXT R
130 NEXT I
140 GOTO 140
```

The radius is given as an integer value. No error will be produced if part, or all of an ellipse extends beyond the screen limits.

Apart from drawing complete ellipses by using the start and end angle option, CIRCLE can be made to produce arcs. Logically enough, the start and end angles give the start and end positions of the arc. The angles must be given in radians, and so may range between 0 and 2 × PI. Figure 8.3 shows how MSX–BASIC views the graphics screen for the CIRCLE command.

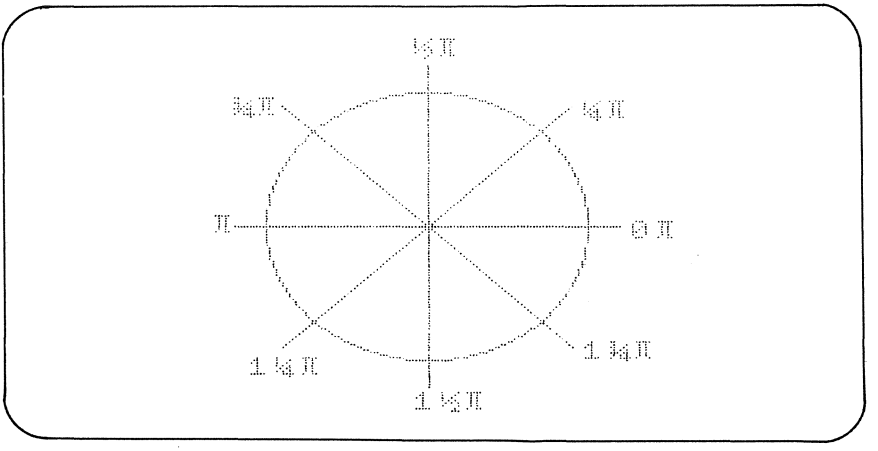


Figure 8.3 CIRCLE and the graphics screen

The effect of altering the start and end angles is shown by Program 8.7.

Program 8.7

```
10 REM *
20 REM * Start/End Angles
```

```
30 REM *
40 COLOR 15,1,1
50 SCREEN 2
60 REM *
70 REM * Alter Start Angle
80 REM *
90 R = 96
100 FOR SA = 0 TO 6.28 STEP .4188
110 CIRCLE(128,96),R,15,SA,0
120 R= R-6
130 NEXT SA
140 FOR J = 1 TO 300 : NEXT J
150 CLS
160 REM *
170 REM * Alter End Angle
180 REM *
190 R = 96
200 FOR EA = 0 TO 6.28 STEP .4188
210 CIRCLE(128,96),R,15,0,EA
220 R= R-6
230 NEXT EA
240 GOTO 240
```

If either of these start or end angles is prefixed with a minus sign, the line will be drawn from that point to the centre. Using this method, spoked wheels and pie charts may be produced. Figure 8.4 was produced from Program 8.8.

Program 8.8

```
10 REM *
20 REM * Radial Lines
30 REM *
40 COLOR 15,1,1
50 C=5
60 SCREEN 2
70 FOR Y = 40 TO 160 STEP 40
80 FOR X=40 TO 216 STEP 40
90 FOR EA = 0 TO 6.29 STEP .4188
100 CIRCLE(X,Y),19,C,0,-EA
110 NEXT EA
120 NEXT X
130 C=C+2
140 NEXT Y
150 GOTO 150
```

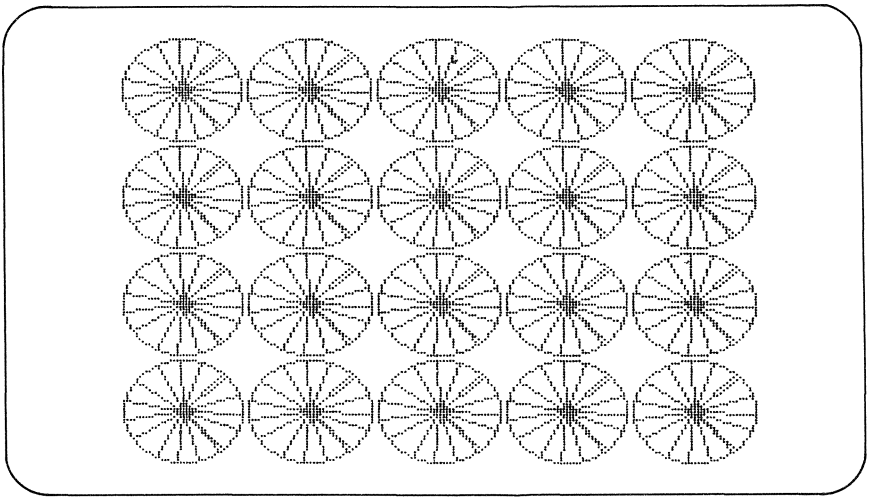


Figure 8.4 'Wheels' designed by Program 8.8

Finally, there is the aspect ratio. This is the ratio of the vertical height of an ellipse to its width. A value less than 1 produces an ellipse that is compressed in the vertical plane, while an aspect ratio greater than 1 produces elongation in the vertical plane. Program 8.9 and Figure 8.5 illustrate this.

Program 8.9

```
10 REM *
20 REM * Altering Aspect Ratio
30 REM *
40 COLOR 15,1,1
50 SCREEN 2
60 FOR I=80 TO 24 STEP -4
70 CIRCLE(128,96),96,,,,10/I
80 CIRCLE(128,96),96,,,,I/10
90 NEXT
100 GOTO 100
```

The default aspect ratio is 1, which you might expect to produce a true circle. Unfortunately, due to the way the TV picture is produced, this is not the case. An aspect ratio of 1.4 (or approx. 256/192) adjusts the circle so that it looks correct.

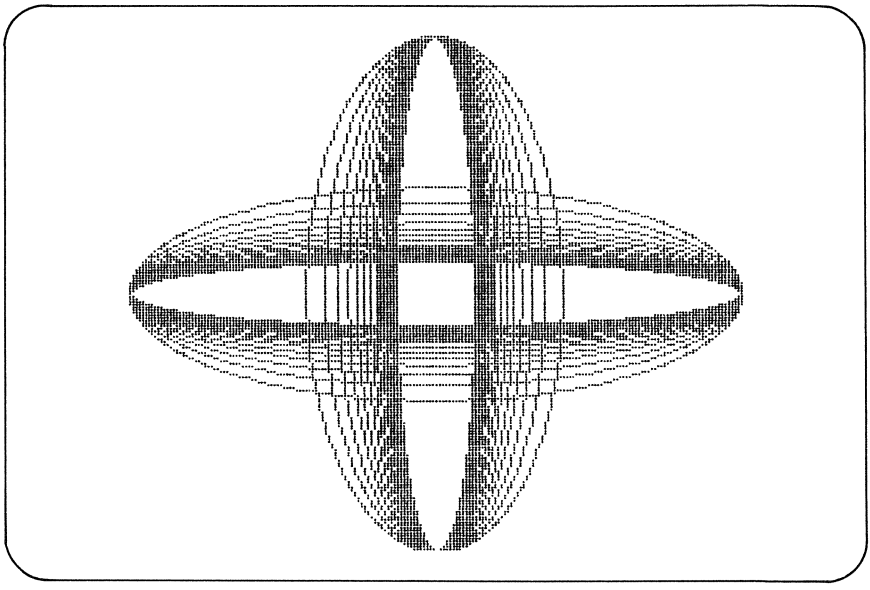


Figure 8.5 Program 8.9 – changing the aspect ratio

The use of colour

All the demonstration programs so far have been two-colour examples. In this section, the ways of setting foreground, background and border colours, and the rules governing colour presentation on the screen are considered.

Setting defaults

Any of the graphics commands which omit the colour parameter will use the relevent default colour. Default colours are always assumed to be those set by the most recent COLOR command as demonstrated by Program 8.10.

Program 8.10

```
10 REM *
20 REM * Altering Default Colour
30 REM * to Draw Colour Bars
40 REM *
50 SCREEN 2
60 C=0
70 FOR J = 0 TO 184 STEP 8
80 COLOR C
```

```
90 LINE (I,J)-STEP(256,8),,BF
100 C = C+1
110 IF C>15 THEN C = 1
120 NEXT J
130 GOTO 130
```

Note that although COLOR changes the default settings for the foreground and background colours, the colours of points already on the screen are not altered in any way. The way to effect a visible change in the background colour is to give a COLOR command followed by CLS, which (sadly) erases the entire screen. The border colour, however, may be seen to change instantly when altered by the COLOR command. Program 8.11 illustrates this.

Program 8.11

```
10 REM *
20 REM * Altering Border Colour
30 REM *
40 COLOR 15,1,1
50 SCREEN 2
60 LINE (40,40)-(216,160),4,BF
70 FOR I=1 TO 15
80 COLOR ,,I
90 FOR J=1 TO 100:NEXT J
100 NEXT I
110 GOTO 70
```

The high-resolution colour rule

As far as colour is concerned, the high-resolution screen may be divided into 6144 blocks, each block with a dimension of 8 × 1 pixels. As an example, the pixels given by coordinates (Ø,Ø) to (7,Ø) are in one block, coordinates (8,Ø) to (15,Ø) are in the next, and so on. The high-resolution colour rule stipulates that a maximum of two colours only are allowed in any one block. Program 8.12 demonstrates what happens when this rule is disobeyed.

Program 8.12

```
10 REM *
20 REM * Colour Rule Violation
30 REM *
40 COLOR 15,15,15
50 SCREEN 2
```

```
60 LINE (124,92)-STEP(8,8),8,BF
70 FOR I = 1 TO 500 : NEXT I
80 BEEP
90 PSET (129,96),1
100 GOTO  100
```

Initially, the block given by the coordinates (128,96) to (135,96) contains red and white only. When the black pixel is placed in this block at coordinate (129,96), all the red pixels in that block change to black. The reason for this will become clear when the way the VDP works is examined in Chapter 9. The best way to avoid this problem is to plan the colour scheme of the screen drawing in advance of programming.

The low-resolution screen is sometimes known as the 'multi-colour screen'. As each pixel is 4 × 4 points in size, it is impossible for more than two colours to exist in a block which is 8 points wide — so the colour clash problem never arises.

Painting shapes

Filling in shapes with colour could be done using PSET, but the technique is quite complex and very slow. MSX–BASIC provides a painting command which will paint any shape at quite a respectable speed. The PAINT command differs in its syntax depending on the graphics screen used.

Painting high-resolution shapes

The syntax of PAINT for use with SCREEN 2 is:

PAINT ⟨coordinate specifier⟩[,⟨paint colour⟩]

The coordinate specified must lie within the boundaries of the shape to be painted and also within the dimensions of the screen. In addition, the paint colour used must match that of the shape to be painted. Program 8.13 draws circles and paints them in.

Program 8.13

```
10 REM *
20 REM * High Resolution Painting
30 REM *
40 COLOR 15,1,1
50 SCREEN 2
```

```
60 FOR C = 15  TO 1 STEP -1
70 CIRCLE (128,96),C*7,C
80 PAINT (128,96),C
90 NEXT
100 GOTO 100
```

The way PAINT works is to paint outward from the centre until either a boundary of the matching colour, or the limits of the screen are encountered. If the paint and shape colours are different, a condition known as 'paint spilling' arises, where the paint colour overruns the boundary of the shape. In Program 8.14 the coordinate for PAINT lies outside the shape to be drawn, so producing paint spill.

Program 8.14

```
10 REM *
20 REM * Paint Spill
30 REM *
40 COLOR 15,1,1
50 SCREEN 2
60 LINE (10,10)-(20,20),8,BF
70 LINE (180,170)-(200,190),8,BF
80 CIRCLE (128,96),70,15
90 PAINT (10,10),15
100 GOTO 100
```

If the colour of the pixel at the coordinate specified by PAINT matches that of the paint colour, then the command assumes that it has done its job, and the shape will not be painted as expected.

Painting low-resolution shapes

The syntax of PAINT differs here, with the possible inclusion of an extra parameter:

> PAINT ⟨coordinate specifier⟩[,⟨paint colour⟩][,⟨colour regarded as the border⟩]

Here, the border colour is not that of the screen border, but that of the shape to be painted. Program 8.15 draws a white circle and paints it white, Program 8.16 draws a white circle and paints it red — with a white border.

Program 8.15

```
10 REM *
20 REM * Low Resolution Painting
30 REM *
40 COLOR 15,1,1
50 SCREEN 3
60 CIRCLE (128,96),70,15
70 PAINT (128,96),15,15
80 GOTO 80
```

Program 8.16

```
10 REM *
20 REM * Low-Res Painting with Border
30 REM *
40 COLOR 15,4,4
50 SCREEN 3
60 CIRCLE (128,96),70,15
70 PAINT (128,96),8,15
80 GOTO 80
```

It is not wise to assume default colours using this command. The border colour given in the PAINT command must *always* match the colour of the shape to be painted. A syntax error will result if a border colour is given for PAINT using SCREEN 2.

Determining pixel colour

The first of the graphics functions, POINT, returns the colour number of a specific point on a graphics screen. This function can only work with absolute coordinate values. The value it returns will be an integer between −1 and 15. If coordinates outside the screen are given, −1 will be returned.

POINT is one of the few graphics related commands that may be used during a text mode without producing a syntax error — not that it is much use, however. When used with a text screen, POINT will always return zero. I'll now show a rather different example of how POINT may be used.

In Program 8.17 a point is plotted and the X and Y coordinates are incremented. The POINT function is used to indicate when the coordinates of a point lie outside the screen limits, so allowing the direction and colour of the plotting to be changed.

If left to run for long enough, a tapestry effect is produced, which constantly changes due to violation of the colour rules.

Program 8.17

```
10 REM *
20 REM * Tapestry Weaving:
30 REM *
40 COLOR 1,1,1
50 SCREEN 2
60 COLOR 1,1,1
70 SCREEN 2
80 DX = 3 : DY = -3
90 X = 112 : Y = 96
100 C=15: R = 8
110 X= X + DX : Y =Y + DY
120 PSET (X,Y),C
130 IF POINT(X,Y) <> -1 GOTO 110
140 REM *
150 REM * Reverse Direction
160 REM * and Change Colour
170 REM *
180 DX = -DX
190 SWAP DX,DY
200 SWAP C,R
210 GOTO 110
```

Mixing text and pictures

Most program output is produced by using PRINT, or one or its variants, such as PRINT USING. PRINT has no effect on the graphics screens at all. TRON and TROFF are also useless while using a graphics mode. Instead, a file has to be opened and written to using PRINT# commands. The required file is named "GRP:" — the graphics screen. Once opened, the screen can be written to (but not read from).

The screen position where the data is to be printed can be determined by using a PRESET statement. This position defines the top left-hand corner of the first character of the data. If no print position is given, then data will be printed at the last position referenced.

When the position of the text goes beyond the bottom right position of the screen, the equivalent of a HOME command takes effect, and printing continues from the top left of the screen.

One novelty aspect of printing to the graphics screen is that the

colour of the text may be altered by changing the default foreground colour. Screen printing is illustrated in Program 8.18 and Figure 8.6.

Program 8.18

```
10 REM *
20 REM * Text and Graphics
30 REM *
40 COLOR 15,15,15
50 SCREEN 2
60 OPEN "GRP:" AS #1
70 LINE (0,0)-(255,191),1,B
80 LINE (120,0)-(120,191),1
90 PRESET (8,32)
100 FOR I = 1 TO 14
110 COLOR I
120 PRINT#1,"MSX COMPUTERS"
130 PRESET STEP (8,0)
140 NEXT  I
150 PRESET (164,16)
160 COLOR 1
170 PRINT#1,"Circle"
180 CIRCLE (188,64),30
190 PAINT (188,64)
200 PRESET (144,112)
210 COLOR 8
220 PRINT#1,"Box"
230 LINE (136,124)-STEP(40,40),8,BF
240 PRESET (186,112)
250 COLOR 4
260 PRINT#1,"Triangle"
270 LINE (208,128)-STEP(30,30)
280 LINE -STEP(-30,0)
290 LINE -STEP(0,-30)
300 GOTO 300
```

If you are writing programs which manipulate other files, such as the cassette ("CAS:") then be sure to increase MAXFILES.

Input while using graphics

If a program encounters an INPUT statement while using a graphics screen, the screen mode will revert to a text mode because the INPUT outputs a '?'. An alternative means has to be found to input data if a

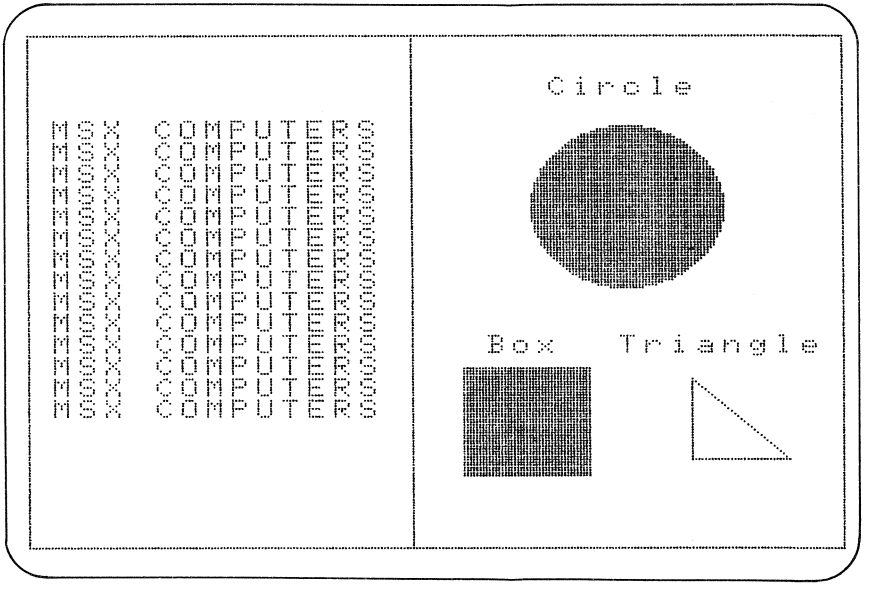


Figure 8.6 Program 8.18 – mixing text and pictures

graphics mode is to be maintained. A possible solution is to use one of the functions or commands that reads the keyboard: INKEY$, LINE INPUT or INPUT$( ). By printing the data to the graphics screen as it is received, the user can see what is typed. Program 8.19 uses INPUT$( ) to provide a simple input and echo routine for the high-resolution screen.

Program 8.19

```
10 REM *
20 REM * Input Routine
30 REM *
40 COLOR 15,15,15
50 SCREEN 2,0,0
60 OPEN "GRP:" AS #1
70 LINE (0,0)-STEP(255,191),1,B
80 PRESET (8,8)
90 COLOR 1
100 PRINT#1,"Radius : ";
110 N$ = ""
120 X = 80
130 PRESET(X,8)
140 A$ = INPUT$(1)
150 IF A$ = CHR$(13) THEN GOTO 250
```

```
160 REM *
170 REM * Validate Input
180 REM *
190 IF A$<"0" OR A$>"9" THEN PLAY "L2402
C" : GOTO 130
200 PRINT#1,A$; : N$ = N$+A$ : X = X+8
210 GOTO 130
220 REM *
230 REM * Draw Circle
240 REM *
250 R = VAL(N$)
260 IF R>80 THEN BEEP : GOTO 280
270 CIRCLE (128,100),R
280 LINE (80,8)-STEP(56,8),15,BF
290 GOTO 110
```

The DRAW command

DRAW allows control over the graphics screen in much the same way as PLAY controls sound. The DRAW command requires a character string composed of a series of characters which are part of another sublanguage — the **Graphics Macro Language** (GML). MSX–BASIC has the ability to keep track of the last position referenced on a graphics screen, which was seen in previous examples where relative coordinate specifiers were used. The last referenced position may be thought of as an invisible graphics cursor. DRAW subcommands are largely relative, so the command relies heavily on this cursor concept.

The drawing subcommands

There are eight basic subcommands. Each command is followed by an integer value which describes the number of points to be drawn. The commands are as follows:

U	Draw up.
R	Draw right.
D	Draw down.
L	Draw left.
E	Draw diagonally up and right.
F	Draw diagonally down and right.
G	Draw diagonally down and left.
H	Draw diagonally up and left.

The virtue of GML strings is that they can quite concisely define quite

complex shapes. They also provide a more natural way of drawing shapes than using the rather abstract constructs of lines and ellipses. Program 8.20 and Figure 8.7 illustrate their use.

Program 8.20

```
10 REM *
20 REM * GML Command
30 REM *
40 COLOR 15,1,1
50 SCREEN 2
60 REM *
70 REM * Draw Stairs
80 REM *
90 PSET(20,42)
100 FOR I = 1 TO 4
110 DRAW "E40R20D15G40U15E40G40L20R20D15
"
120 NEXT I
130 DRAW "L80U60"
140 PAINT STEP (1,1),15
150 DRAW "L1U1E40R80D60"
160 PAINT STEP (-1,-1),15
170 DRAW "R1D1G40"
180 COLOR 1
190 DRAW "U15"
200 COLOR 15
210 REM *
220 REM * Draw Spiral
230 REM *
240 PSET(188,40)
250 DRAW "R5D5L10U10R15D15L20U20"
260 DRAW "R25D25L30U30R35D35L40"
270 REM *
280 REM * Draw Playhouse
290 REM *
300 PSET (140,112)
310 DRAW "E15R30F15L60R5D40R5U30R15"
320 DRAW "D30L15R45U40"
330 PSET STEP (-25,10)
340 DRAW "R20D20L20U20D10R20L10U10D20"
350 PSET STEP (0,-45)
360 DRAW "L5U5R5D4"
370 GOTO 370
```

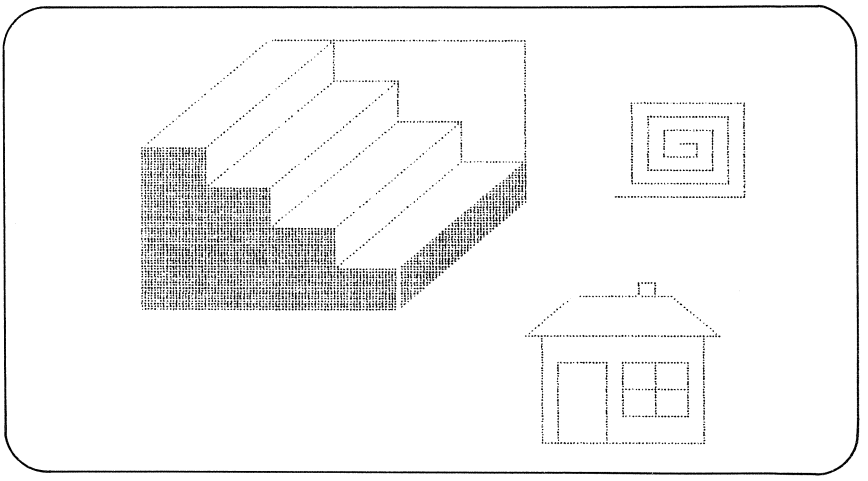


Figure 8.7 Drawings created by Program 8.20

The GML offers M (Move), an alternative to the LINE command. A coordinate pair may be given as offsets from the current cursor position, where the X and Y coordinates are prefixed with '+' or '–' signs; or as absolute screen coordinates. When used in the relative form, if one of the offsets is zero, that offset *must* still be prefixed with a sign character thus:

```
40 DRAW "M+100,+0"
```

The M subcommand works rather like LINE in relative form.

Positioning the cursor

PRESET or PRESET can be used to set the position of the graphics cursor, but in general, this is rather an untidy method. Both commands may alter the colour of pixels which is undesirable in some circumstances. If any of the GML's drawing or move commands are prefixed with the letter B (blank), the cursor will be moved by an appropriate number of points *without* setting or resetting any points on the screen. DRAW becomes particularly useful when setting the position for text output to the graphics screen.

If the letter N precedes a GML drawing command, a number of points are set, then the cursor is returned to its original position. The effect of both the N and B prefixes is demonstrated by Program 8.21.

Program 8.21

```
10 REM *
20 REM * "N" and "B" Subcommands
```

```
30 REM *
40 OPEN "GRP:" AS #1
50 SCREEN 2
60 DRAW "BM16,8"
70 PRINT #1,"DRAW demo"
80 DRAW "BM128,96"
90 DRAW "NU60NE40NR60NF40"
100 DRAW "ND60NG40NL60NH40"
110 REM *
120 REM * Reposition Cursor For Text
130 REM *
140 DRAW "BM16,176"
150 PRINT #1,"GML"
160 GOTO 160
```

Setting colour

GML drawing is usually carried out using the current foreground colour. Any of the MSX–BASIC colours may be specified within GML strings by using the C (colour) subcommand. This subcommand is followed by a colour code number between Ø and 15; e.g., C15 sets the colour to white. Unlike the COLOR command, GML colour settings do not affect the default colour used by graphics commands like PAINT and CIRCLE.

Scaling and rotation

Any shape drawn with the GML may be scaled up or down using the S (scale) subcommand. Taking the GML string "R12" as an example, we would normally interpret this to mean 'draw right 12 pixels'. The GML's scaling factor is 1/4 — a single move is interpreted to be 1/4 of a pixel. The default setting is S4 — 1 move is equivalent to 1 pixel. If the command 'S2R12' were given, a line 6 pixels in length would be drawn, while the command 'S8R12' would produce a line of 24 pixels.

Some problems arise when a shape is scaled up or down by a large factor. If we have the statement:

```
40 DRAW "S1R16D17R3U1"
```

the actual result produced will be:

Draw right 4 pixels.
Draw down 4 pixels.
Draw right Ø pixels.
Draw up Ø pixels.

Scaling will not always produce the results you expect as some move values will be rounded down. Program 8.22 illustrates scaling.

Program 8.22

```
10 REM *
20 REM * Scaling
30 REM *
40 SCREEN 2
50 COLOR 15,4,1
60 FOR I=4 TO 18
70 CLS
80 N=40+I : PLAY "N=N;"
90 DRAW "BM40,60"
100 DRAW "S"+STR$(I)
110 DRAW "R40D30L40U30BM+5,+30"
120 DRAW "U25R10D25BM+5,-20"
130 DRAW "R15D15L15U15BM+7,+0"
140 DRAW "D15BU7NR8L7BM+20,-18"
150 DRAW "R5M-10,-15L30M-10,15NR5"
160 DRAW "BM+40,+15"
170 FOR D=1 TO 250 : NEXT D
180 NEXT
190 GOTO 190
```

Once a scale has been set, it becomes the default for all subsequent GML commands. The first DRAW command in any program should set the scale.

This is also true of the A (angle) command. A shape will be rotated anti-clockwise by the number of degrees set by the A command; e.g., if a rotation of 90° is used, R commands will perform like U commands. The degree of rotation for each value of A is:

A0	0°
A1	90° anti-clockwise
A2	180° anti-clockwise
A3	270° anti-clockwise

Interesting geometric patterns can be produced using a combination of shape rotation and scaling. Program 8.23 produced the pattern shown in Figure 8.8.

Program 8.23

```
10 REM *
20 REM * Scaling and Rotation
30 REM *
40 COLOR 15,1,1
50 SCREEN 2
60 FOR S = 2 TO 8
70 FOR A = 0 TO 3
80 PSET (128,96)
90 DRAW "S=S;A=A;R30E15U30L30G15D30E15"
100 DRAW "R30L30U30D15L15R30E15G15"
110 DRAW "D30"
120 NEXT A
130 NEXT S
140 GOTO 140
```

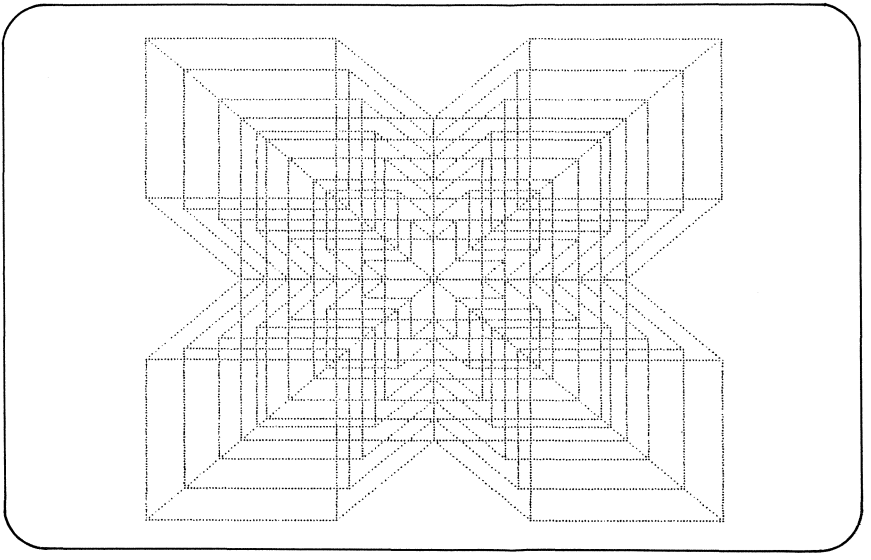


Figure 8.8 Patterns produced by Program 8.23

Variables and substrings

In common with the Music Macro Language, the GML may incorporate variables and substrings. The method is identical to the MML's, i.e., a command is followed by the '=' sign, a variable name and semicolon.

The X command allows frequently-used GML commands to be included as substrings. Another case for using a substring is when the

total length of a command exceeds 255 characters. The commands may be broken into substrings and called using X, so reducing the command string to a legal length.

Programming applications

In this section, a number of program examples are given to demonstrate some of the uses of the MSX–BASIC graphics commands.

Program 8.24 Image copying

Copying a screen image from one part of the screen to another can be achieved using a two-dimensional array. The first step is to decide the size of the area that is to be copied, and create an appropriately dimensioned array.

The colour of each pixel in the copy area is found using POINT and then stored in this array. This data is then used as colour information to set points using PSET. Holding the data in an array has other advantages. By altering the indexing system used, the image can be plotted upside down or reflected in the X or Y axis. If the screen uses two colours only, then colour reversal can be achieved quite easily.

Note that the reading and writing of the array data is excruciatingly slow, so some patience is required.

```
10 REM *
20 REM * Screen Area Copying
30 REM *
40 CLEAR
50 DEFINT A-Z
60 REM *
70 REM  * Holding Array for Screen Data
80 REM *
90 DIM A(50,50)
100 BC = 1 : SC = 15
110 COLOR 15,1,1
120 SCREEN 2
130 ON INTERVAL = 100 GOSUB 630
140 OPEN "grp:" AS #1
150 LINE (56,0)-STEP(128,10),15,BF
160 PSET (68,1) : COLOR 1
170 PRINT#1,"Screen Copying"
180 COLOR 15
190 REM *
```

```
200 REM  * Draw Screen Image
210 REM *
220 LINE (80,80)-(130,130),15,BF
230 LINE (82,82)-(128,128),1,B
240 FOR I = 2 TO 12 STEP 2
250 CIRCLE (105,105),20,1,,,12/I
260 CIRCLE (105,105),20,1,,,I/12
270 NEXT I
280 REM *
290 REM * Fill Holding Array
300 REM *
310 INTERVAL ON
320 FOR I = 1 TO 50
330 FOR J = 1 TO 50
340 A(I,J) = POINT (80+I,80+J)
350 NEXT J
360 NEXT I
370 INTERVAL OFF
380 COLOR ,,1
390 REM *
400 REM * Draw Copies
410 REM *
420 FOR I = 1 TO 2
430 READ X,Y
440 FOR J = 1 TO 50
450 FOR K = 1 TO 50
460 PSET (X+J,Y+K),A(J,K)
470 NEXT K
480 NEXT J
490 NEXT I
500 REM *
510 REM * Draw Inverted Copy
520 REM *
530 FOR I = 1 TO 50
540 FOR J = 1 TO 50
550 IF A(I,J)=1 THEN C=15 ELSE C=1
560 PSET (29+I,80+J),C
570 NEXT J
580 NEXT I
590 GOTO 590
600 REM *
610 REM * Interval Routine
620 REM *
630 PLAY "T255L64SM100004C05C"
```

```
640 SWAP BC,SC : COLOR ,,SC
650 RETURN
660 REM *
670 REM * Copy Co-ordinate data
680 REM *
690 DATA 30,28,80,28
```

Program 8.25 Mirroring points

This program produces simple kaleidoscope effects. The cursor keys are used to trace on the screen. The points drawn are reflected in the X and Y axes, which are taken to be (0,96)–(255,96) and (128,0)–(128,191) respectively.

The method by which the cursor position is set from the STICK( ) function uses array indexing. The array contains the offsets needed to move the cursor in the direction returned from STICK( ). This is considerably faster than a series of eight IF . . . THEN statements, or even the use of ON GOTO branching.

```
10 REM *
20 REM * Mirroring
30 REM *
40 COLOR 15,1,1
50 SCREEN 2,0,0
60 DIM DIR(9,2)
70 X = 128 : Y = 96
80 REM *
90 REM * Set up the Joystick Offsets
100 REM *
110 FOR I = 0 TO 8
120 FOR J = 0 TO 1
130 READ DIR(I,J)
140 NEXT J
150 NEXT I
160 REM *
170 REM * Main Loop
180 REM *
190 X = X + DIR(STICK(0),0)
200 Y = Y + DIR(STICK(0),1)
210 REM *
220 REM * Range Checking
230 REM *
240 IF X > 255 THEN X = 255
250 IF Y > 191 THEN Y = 191
```

```
260 IF X < 0 THEN X = 0
270 IF Y < 0 THEN Y = 0
280 REM *
290 REM * Plot Points
300 REM *
310 PSET(X,Y)
320 PSET(256-X,Y)
330 PSET(X,192-Y)
340 PSET(256-X,192-Y)
350 GOTO 190
360 DATA 0,0,0,-1,1,-1,1,0,1,1
370 DATA 0,1,-1,1,-1,0,-1,-1
```

Program 8.26 Moving second hand

A clock simulation can be produced using the ON INTERVAL interrupt and the point rotation method discussed in Chapter 3. An array stores the coordinates of all the points produced by a full rotation through 360° of a single point on the Y axis. This gives the second hand divisions of the clock face.

```
10 REM *
20 REM * Circle Drawing:
30 REM * Analogue Clock
40 REM *
50 COLOR 15,1,1
60 DIM T(60,2)
70 ON INTERVAL=50 GOSUB 310
80 SCREEN 2
90 X = 0 : Y = 60
100 IX=0  : M$="T255L24S1M80006C"
110 FOR I = 0 TO 6.17847 STEP .10471
120 REM *
130 REM * Set Up the Plot array
140 REM * and plot Seconds markers
150 REM *
160 NX =(X * COS(I))-(Y*SIN(I))
170 NY =(X * SIN(I))+(Y*COS(I))
180 A = (128 - (NX * .9))
190 B = 96 - NY
200 MA=A+SGN(A-128):MB=B-SGN(96-B)
210 CIRCLE(MA,MB),1
220 T(IX,0)=A : T(IX,1)=B
230 IX=IX+1
```

```
240 NEXT I
250 IX=59 : INTERVAL ON
260 GOTO 260
270 REM *
280 REM * Update Clock Hand Position
290 REM * Once a Second
300 REM *
310 IF IX<59 THEN 360
320 LINE(128,96)-(T(59,0),T(59,1)),1
330 IX=0 : PLAY M$
340 LINE(128,96)-(T(IX,0),T(IX,1)),15
350 RETURN
360 LINE(128,96)-(T(IX,0),T(IX,1)),1
370 IX=IX+1 : PLAY M$
380 LINE(128,96)-(T(IX,0),T(IX,1)),15
390 RETURN
```

Program 8.27 Pie charts

Pie charts can be produced using the point rotation method, however, the start/end angle option of CIRCLE is a much faster and more convenient method. Up to 12 items may be charted, but this may easily be changed as required. A typical pie chart is shown in Figure 8.9.

```
10 REM *
20 REM * Pie Charts
30 REM *
40 OPEN "GRP:" AS #1
50 COLOR 15,15,15
60 DIM A(12)
70 V=0
80 FOR I=1 TO 12
90 READ A(I)
100 V = V+A(I)
110 NEXT I
120 REM *
130 REM * Calculate Proportions
140 REM * and Plot Graph
150 REM *
160 SCREEN 2
170 UNIT = 6.283/V
180 LINE (0,0)-(255,191),1,B
190 LINE (4,4)-(80,187),1,BF
200 LINE (84,4)-(251,187),1,BF
```

```
210 LINE (86,6)-(249,185),15,B
220 CIRCLE (176,96),80,15,,,1.2
230 CIRCLE (176,96),80,15,-6.283,-6.283,
1.2
240 SA = 0
250 FOR I=1 TO 12
260 SA=SA+(UNIT*A(I))
270 CIRCLE (176,96),80,15,-SA,-SA,1.2
280 NEXT
290 LINE (6,6)-(78,185),15,B
300 LINE (6,22)-STEP(72,0),15
310 DRAW "BM24,12":PRINT#1,"Data":PRINT#
1,STRING$(3,13)
320 FOR I=1 TO 12
330 PRINT#1,USING " ###.";I;
340 PRINT#1, USING "#####";A(I)
350 NEXT
360 GOTO 360
370 DATA 100,80,66,90,89,14
380 DATA 150,120,72,35,16,94
```

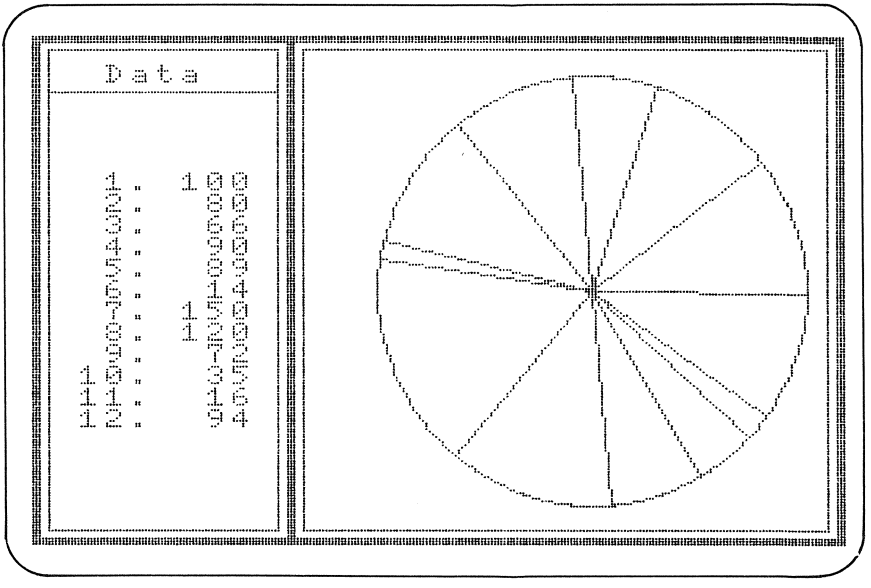


Figure 8.9 Pie chart, courtesy of Program 8.27

Program 8.28 Graphing

The final program of this chapter is a complete graphing system to produce bar, scatter and line diagrams. Axes are scaled as on graph paper, with the maximum and minimum values of the data set determining the scaling factor on the Y axis, and the position of the X axis.

The intervals on the X axis are the same, thus excluding the scientific type of graph where X is plotted *against* Y. Typical graphs are shown in Figures 8.10a, b and c.

```
10 REM ****************************
20 REM *                          *
30 REM * Graph Drawing Program *
40 REM *                          *
50 REM ****************************
60 REM *
70 REM * Branch to Main Input Routine
80 REM *
90 GOTO 200
100 REM *
110 REM * Input Scrolling Routine
120 REM *
130 LOCATE 0,4,0
140 FOR I = ITEM-18 TO ITEM-1
150 PRINT USING "###";I;:PRINT TAB(7);
TBL(I)
160 NEXT I
170 LOCATE 3,22 : PRINT SPC(16)
180 LOCATE 0,22
190 RETURN
200 REM *
210 REM * Data Input Section
220 REM *
230 SCREEN 0
240 OPEN "GRP:" FOR OUTPUT AS #1
250 PLAY "T255L12SM2000"
260 COLOR 15,4,4 : KEY OFF
270 DIM TBL(110)
280 CLS
290 PRINT "Data Input:" : PRINT
300 PRINT "Maximum of 110 Data Items (*
exits)." : PRINT
310 ITEM = 1 : MX=0 : MIN=0
```

```
320 PRINT USING "###";ITEM;: PRINT ":";
TAB(8)
330 LINE INPUT A$
340 REM *
350 REM * Terminate Data input if "*"
360 REM *
370 IF A$="*" THEN GOTO 570
380 X=VAL(A$)
390 REM *
400 REM * Fill Array and Determine
410 REM * Max. and Min. Data Values
420 REM *
430 TBL(ITEM)=X : IF X>MX THEN MX=X
440 IF X<MIN THEN MIN = X
450 ITEM = ITEM + 1
460 REM *
470 REM * If Screen Full - Scroll
480 REM * Display
490 REM *
500 IF ITEM>19 THEN GOSUB 130
510 IF ITEM < 111 THEN GOTO 320
520 PRINT
530 PRINT "Array Full: Press key for men
u." : X$ = INPUT$(1)
540 REM *
550 REM * Main Menu
560 REM *
570 SCREEN 0
580 PRINT TAB(1);"Graph Drawing:"
590 PRINT : PRINT
600 PRINT TAB(8);"1. Bar Chart" : PRINT
610 PRINT TAB(8);"2. Line Graph" :PRINT
620 PRINT TAB(8);"3. Scatter Diagram" :
PRINT
630 PRINT TAB(8);"4. Data Entry":PRINT
640 PRINT TAB(8);"5. Exit Program":
PRINT
650 PRINT TAB(8);"Input Option (1-5): (
)";
651 LOCATE 2,20,0:PRINT "* Press Space B
ar to Return to Menu"
660 BL$ = " " : BC$ = "_"
670 LOCATE 29,13,0
680 A$=INKEY$
```

```
690 IF A$<>"" THEN 740
700 SWAP BL$,BC$
710 PRINT BL$;
720 FOR D = 1 TO 100 : NEXT D
730 GOTO 670
740 PRINT A$
750 REM *
760 REM * Validate Input
770 REM *
780 IF A$<"1" OR A$>"5" THEN PLAY "O2F"
: LOCATE 29,13: PRINT " " : GOTO 670
790 PLAY "O4CO6C"
800 PRINT : PRINT
810 PRINT TAB(8);"Option ";A$;" Selected
."
820 FOR D = 1 TO 500 : NEXT
830 SEL = VAL(A$)
840 IF SEL = 4 THEN GOTO 280
850 IF SEL = 5 THEN CLS : END
860 CLS
870 COLOR 15,4,4
880 SCREEN 2
890 REM *
900 REM * Graph Calculations
910 REM * Determine Scaling Factor
920 REM *
930 R = ABS(MX)+ABS(MIN)
940 IF R = 0 THEN GOTO 570
950 FTR = 176/R
960 IF NOT(SGN(MIN)) THEN OG = 183 : GOT
O 1010
970 REM *
980 REM *Determine Origin and Draw Axes
990 REM *
1000 OG =183 + (MIN*FTR)
1010 LINE (15,OG)-STEP(240,0)
1020 LINE (15,0)-STEP(0,191)
1030 CIRCLE(8,OG),2
1040 REM *
1050 REM * Scale Y Axis^10
1060 REM *
1070 GOSUB 1560
1080 IF OG = 183 THEN GOTO 1130
1090 PSET (16,OG)
```

```
1100 FOR I = 0 TO INT(ABS(MIN)/10^P)
1110 LINE (12,OG+(FTR*(10^P)*I))-STEP(3,
0)
1120 NEXT
1130 PSET (16,OG)
1140 FOR I = 0 TO INT(MX/10^P)
1150 LINE (12,OG-(FTR*(10^P)*I))-STEP(3,
0)
1160 NEXT
1170 REM *
1180 REM * Determine Step Width
1190 REM * for the X axis
1200 REM *
1210 STP =INT(224/(ITEM-1))
1220 REM *
1230 REM * Graph Selection and Drawing
1240 REM *
1250 ON SEL GOTO 1290,1390,1490
1260 REM *
1270 REM * Draw Bar Chart
1280 REM *
1290 SP = 15 : PSET(SP,OG)
1300 FOR I = 1 TO ITEM-1
1310 LINE (SP,OG)-STEP(STP-2,-(TBL(I)*FT
R)),,BF
1320 SP = SP + STP
1330 NEXT I
1340 FOR D=1 TO 500:NEXT
1350 X$ = INPUT$(1) : GOTO 570
1360 REM *
1370 REM * Draw Line Graph
1380 REM *
1390 SP = 15 : PSET(SP,OG)
1400 FOR I = 1 TO ITEM-1
1410 LINE -(SP,OG-(TBL(I)*FTR))
1420 SP = SP + STP
1430 NEXT I
1440 FOR D=1 TO 500:NEXT
1450 X$ = INPUT$(1) : GOTO 570
1460 REM *
1470 REM * Draw Scatter Diagram
1480 REM *
1490 SP = 20 : PSET(SP,OG)
1500 FOR I = 1 TO ITEM-1
```

```
1510 CIRCLE(SP,OG-(TBL(I)*FTR)),1
1520 SP = SP + STP
1530 NEXT I
1540 X$ = INPUT$(1) : GOTO 570
1550 FOR D=1 TO 500:NEXT
1560 REM *
1570 REM * Determine the Scaling
1580 REM * for the Y Axis ^ 10
1590 REM *
1600 P=4
1610 IF R <10 THEN P = 0 : RETURN
1620 IF INT(R/10^P) = 0 THEN P = P-1:
GOTO 1620
1630 RETURN
```

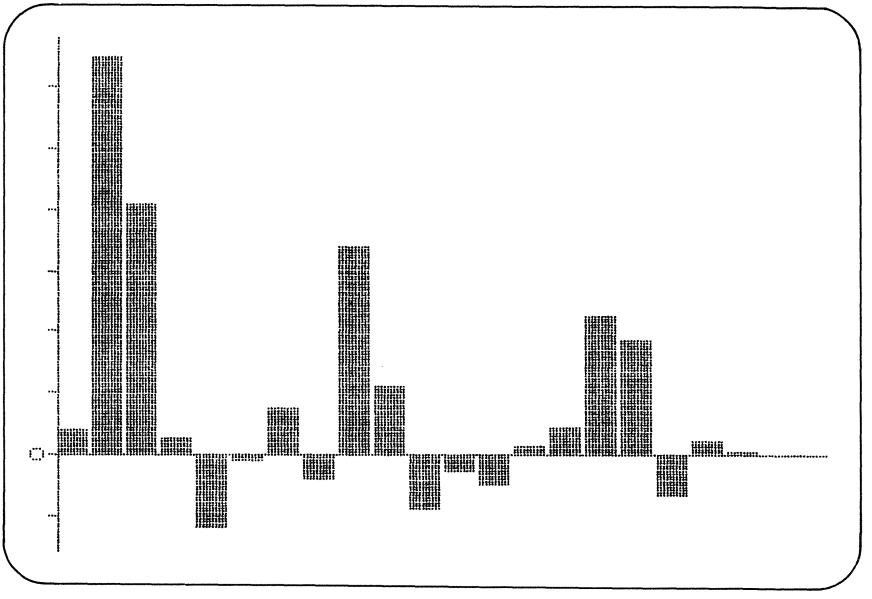


Figure 8.10a Program 8.28 – bar chart

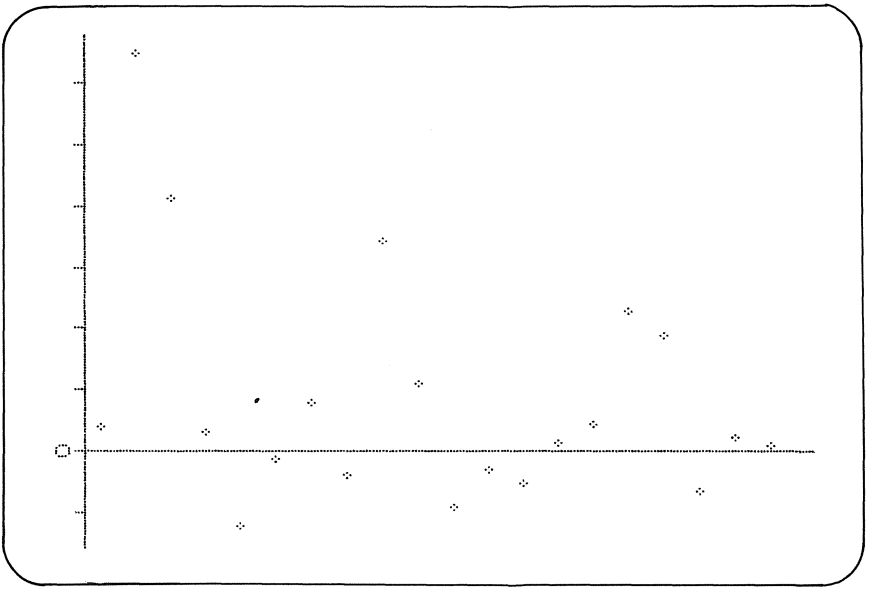


Figure 8.10b Program 8.28 – scatter diagram

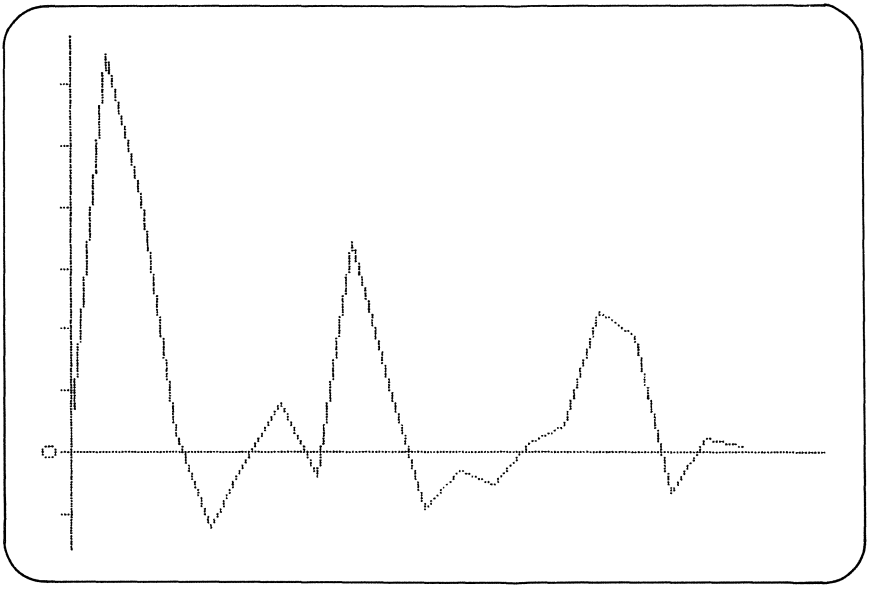


Figure 8.10c Program 8.28 – line graph

Summary

Coordinate specifiers

STEP (⟨Xoffset⟩,⟨Y offset⟩)

(⟨X absolute⟩,⟨Y absolute⟩)

General Commands

PSET ⟨coordinate specifier⟩[,⟨colour⟩]

PRESET ⟨coordinate specifier⟩[,⟨colour⟩]

LINE [⟨coordinate specifier⟩]–⟨coordinate specifier⟩[,⟨colour⟩] [B ¦ BF]

CIRCLE ⟨coordinate specifier⟩,⟨radius⟩[,⟨colour⟩][,⟨start angle⟩] [,⟨end angle⟩][,⟨aspect ratio⟩]

PAINT ⟨coordinate specifier⟩[,⟨colour⟩][⟨colour regarded as border⟩]

POINT (⟨X⟩,⟨Y⟩)

DRAW subcommands

General drawing commands
U, R, D, L, E, F, G, H

Line drawing commands
M ⟨X⟩,⟨Y⟩

M ⟨X offset⟩,⟨Y offset⟩

Prefixes
B Move but do not plot.

N Draw and set cursor to original position.

Others
C Set current colour.

S Set current scale factor.

A Set current drawing angle.

X Execute substring.

9 Advanced graphics

In this chapter, I'll show the MSX–BASIC commands that allow the creation and manipulation of user-defined shapes. In addition, direct access to the video display processor and its support memory will be examined in some detail.

Animation

The illusion of movement can be created following these simple steps:

1. Draw and object at position X,Y.
2. Erase the object using the background colour.
3. Increment/decrement X,Y
4. Goto step 1.

This principle is demonstrated in Program 9.1 where a circle is moved back and forth across the screen.

Program 9.1

```
10 REM *
20 REM * Simple Animation
30 REM *
40 COLOR 15,4,4
50 SCREEN 2
60 X=5:SX=5
70 CIRCLE (X,96),10,15
80 REM *
90 REM * Main Loop
100 REM *
110 CIRCLE (X,96),10,4
120 X=X+SX
130 IF X<5 OR X>250 THEN SX=-SX
140 CIRCLE (X,96),10,15
150 GOTO 110
```

The results are hardly spectacular: the drawing and erasing process is not smooth enough to give an adequate impression of movement. Another disadvantage of this method is seen when the animated object passes over an area of the screen which is not of the background colour. The 'refresh' process will be seen to be destructive. This drawback can be remedied only by first saving the attributes of the area to be drawn over, then restoring this area when the object moves on.

This method is far too time-consuming to be accomplished satisfactorily in BASIC. Fortunately, the video display processor has a built-in ability to carry out animation using user-defined objects called **sprites**.

Properties of sprites

The main features of sprites are summarized below:

1. *Flicker-free movement* Sprites may be moved across a screen smoothly, and without affecting the background display.
2. *Three-dimensional displays* An impression of screen perspective is given as sprites appear to move above and below each other.
3. *Automatic collision detection* The video chip can recognize the collision of two or more sprites on screen. This opens up great possibilities for the games playing fraternity. For example, the obligatory explosion can be created should a missile sprite strike a spaceship sprite.
4. *Automatic wrap-around* When a sprite moves off-screen, it will re-emerge on the opposite side of the screen.

Sprites may be used on three of the MSX screens: the 32×24 text screen and the low- and high-resolution graphics screen. A maximum of 32 sprites may be placed on screen at any time, in either of two sizes.

Creating 8×8 sprites

The default size for sprites in MSX–BASIC is 8×8 pixels. A library of up to 256 unique sprite patterns may be created for these sprites. First, let's examine the steps required to code a sprite pattern. The shape that is to be created (in this case a crosshair cursor) may first be planned on an 8×8 grid as shown in Figure 9.1.

On a row-by-row basis, if an empty grid element is assumed to be '0', and all others to be '1', the grid pattern can be reduced to a series of binary numbers thus:

```
00111000
00010000
```

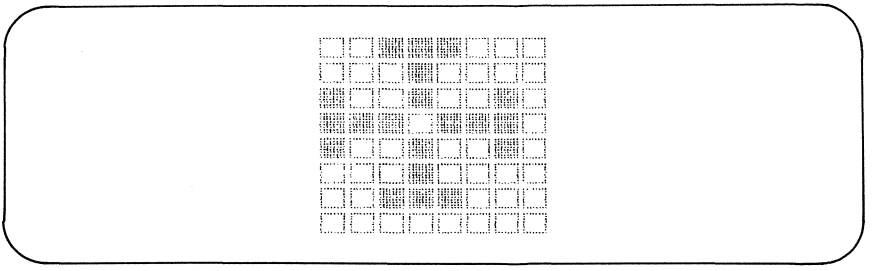


Figure 9.1 Design for a crosshair cursor

```
10010010
11101110
10010010
00010000
00111000
00000000
```

This binary data forms the basis of the pattern definition. It may be left as a series of binary constants or, if memory space is in short supply, converted to a list of decimal or hex numbers.

The next step is to select a screen mode. For convenience, here is the full syntax of the SCREEN command again.

SCREEN [⟨mode⟩][,⟨sprite size⟩][,⟨key click⟩][,⟨baud rate⟩][,⟨printer option⟩]

The options that need to be considered are the ⟨mode⟩ and ⟨sprite size⟩ options. Any screen mode except the 40×24 text mode may be selected, and the sprite size used may be one of the following:

0	8×8	unmagnified (default)
1	8×8	magnified
2	16×16	unmagnified
3	16×16	magnified

Magnification simply doubles the apparent size of a sprite on the screen. For this example, a sprite size or '0' or '1' should be selected.

The actual sprite pattern is created when the definition data is read into a special variable called SPRITE$. SPRITE$ may be thought of as an array variable, with each element of the array containing a sprite pattern.

As SPRITE$ is of the string data type, the numeric sprite data must be converted using the CHR$( ) function before assignment is possible.

Each pattern has an array index, or as it is termed here, a **sprite pattern number**. To create sprite pattern 0, the pattern data must be

read into SPRITE$(Ø). One way of carrying out this assignment is as follows:

```
SPRITE$(Ø)=CHR$(56)+CHR$(16)+CHR$(146)+CHR$(238)
          +CHR$(146)+CHR$(16)+CHR$(56)+CHR$(56)
          +CHR(Ø)
```

(The numbers in brackets are the decimal equivalents of the bit pattern in Figure 9.1.) Alternatively, this assignment could be carried out in a FOR . . . NEXT loop using READ and DATA statements.

Sprite placement

A sprite is put onto the screen using the PUT SPRITE command which is given by the following syntax:

PUT SPRITE ⟨sprite plane number⟩,⟨coordinate specifier⟩ [,⟨colour⟩][,⟨pattern number⟩]

The coordinate specifier gives the position of the *top left-hand corner* of the sprite, and may be given in relative or absolute form as seen for other graphics commands.

Each of the MSX screens that supports sprites may be viewed as a series of display layers or **planes**. The ⟨sprite plane number⟩ determines the display plane where a sprite is to be placed. Only one sprite per plane is allowed (see Figure 9.2).

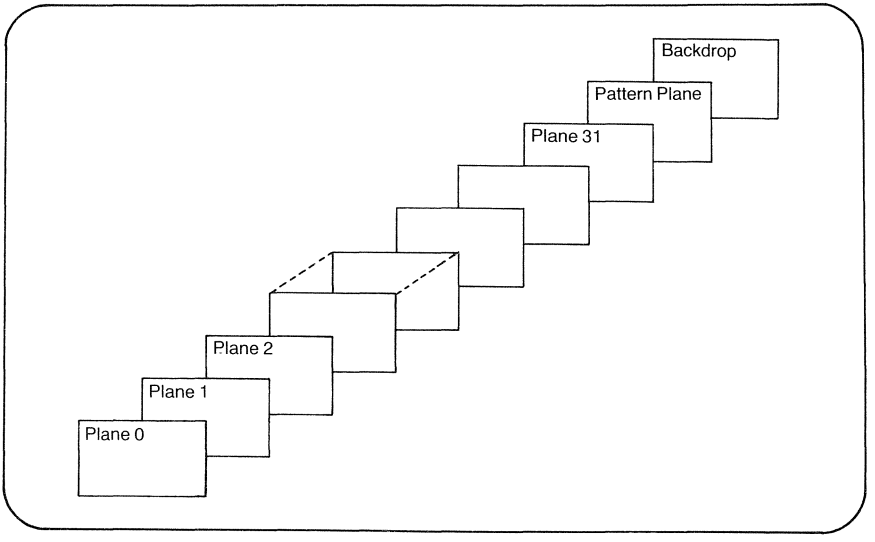


Figure 9.2 Display planes for sprites

A sprite on plane 0 will appear to move above those sprites with lower plane numbers. The lower a sprite's plane number, the higher its priority.

The **pattern plane** is not accessible to sprites, and is the plane used by graphics commands like LINE, CIRCLE and PAINT. Given this planar architecture, it is easy to see why sprites cannot influence existing screen data.

If a pattern number is omitted from a PUT SPRITE command, MSX-BASIC assumes that the pattern number is the same as the plane number.

Program 9.2 defines the cursor sprite and places it on the screen.

Program 9.2

```
10 REM *
20 REM * Sprite Animation
30 REM *
40 COLOR 15,4,4
50 SCREEN 2,0
60 FOR I=1 TO 8
70 READ A : S$=S$+CHR$(A)
80 NEXT
90 SPRITE$(0)=S$
100 INC = 1 : S = 20 : E = 235
110 FOR I = S TO E STEP INC
120 PUT SPRITE 0,(I,92),15,0
130 NEXT I
140 SWAP S,E : INC = -INC
150 GOTO 110
160 DATA &B00111000
170 DATA &B00010000
180 DATA &B10010010
190 DATA &B11101110
200 DATA &B10010010
210 DATA &B00010000
220 DATA &B00111000
230 DATA &B00000000
```

If you are using several sprites at a time, it is best not to use relative coordinate specifiers to give their position. The reference point for such a specifier is that of the last sprite referenced, making it difficult, if not impossible, to move two sprites independently.

Defining 16×16 sprites

To define a 16×16 pixel sprite requires a total of 32 elements of sprite data. Again, a grid can aid the design stage. The way the data values for the sprite pattern are determined is slightly more complex than for 8×8 sprites. If the 16×16 grid is split up into four 8×8 blocks, the order the data is read into SPRITE$ is: top-left, bottom-left, top-right, bottom-right.

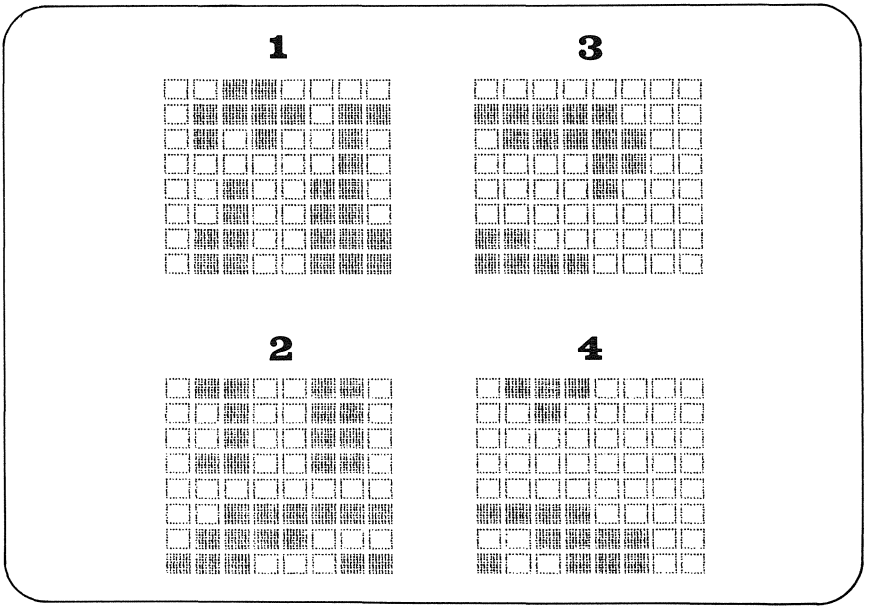


Figure 9.3 Design for a 'large' sprite

The sprite shown in Figure 9.3 is created using Program 9.3. Note the selection of the appropriate sprite size (2) in the SCREEN command.

Program 9.3

```
10 REM *
20 REM * Sixteen x Sixteen Sprite
30 REM * Gothic letter "E"
40 REM *
50 COLOR 1,15,15
60 SCREEN 2,2
70 FOR I=1 TO 16
80 READ A$:S$=S$+CHR$(VAL("&B"+LEFT$(A$,
8)))
90 NEXT
```

```
100 RESTORE 180
110 FOR I=1 TO 16
120 READ A$:S$=S$+CHR$(VAL("&B"+RIGHT$(A
$,8)))
130 NEXT
140 SPRITE$(0)=S$
150 LINE (124,92)-(183,131),15,B
160 PUT SPRITE 0,(112,80),1,0
170 GOTO 170
180 DATA 0011000000000000
190 DATA 0111101111111000
200 DATA 0101001001111100
210 DATA 0000001000001100
220 DATA 0010011000001000
230 DATA 0010011000000000
240 DATA 0110011111000000
250 DATA 0110011111110000
260 DATA 0110011001110000
270 DATA 0010011000100000
280 DATA 0010011000000000
290 DATA 0110011000000000
300 DATA 0000000000000000
310 DATA 0011111111110000
320 DATA 0111100000111100
330 DATA 1110001110011100
```

Owing to the greater amount of data required in their definition only 64 large sprite patterns are allowed. In common with their smaller brethren, they may also be magnified.

The fifth sprite rule

This is the major restriction in the use of sprites. The rule stipulates:

Only four sprites may exist in a given horizontal line at any time.

If five or more sprites are placed in one horizontal line, only the four sprites with the highest priority (i.e., lowest plane numbers) will be displayed. A simple demonstration of this rule is given by Program 9.4.

Program 9.4

```
10 REM *
20 REM * The Fifth Sprite Rule
30 REM *
40 COLOR 15,4,4
```

```
50 SCREEN 2,1
60 SPRITE$(0)=STRING$(8,255)
70 PUT SPRITE 1,(48,96),8,0
80 PUT SPRITE 2,(80,96),9,0
90 PUT SPRITE 4,(128,96),10,0
100 PUT SPRITE 5,(156,96),11,0
110 Y=16:SY=.5
120 PUT SPRITE 3,(104,Y),1,0
130 Y=Y+SY
140 IF Y<16 OR Y>176 THEN SY=-SY : PLAY
"L2405C06C"
150 GOTO 120
```

Sprite bleeding and erasure

Sprites may be placed using any valid integer coordinates, but only the values −32 to 255 for the horizontal (X) coordinate, and −32 to 191, 208, and 209 for the vertical (Y) coordinate have any significance. Negative values allow sprites to be **bled** onto the screen — the sprite can be seen to creep onto the screen from either the top or left of the screen. Similarly, if values of 255 and 191 are given (for the X and Y coordinates respectively), the sprite may be bled onto the screen from the right or bottom of the screen.

When the Y coordinate of a sprite is set to 208, all sprites with lower priority are not displayed. They may be restored to the display by setting the original sprite's Y coordinate to a value other than 208.

A sprite may be totally removed from the screen by setting its Y coordinate to 209.

Monitoring sprite collision

Two sprites are said to collide when one or more of their pixels overlap. This event can be trapped by setting a BASIC interrupt — ON SPRITE GOSUB.

This branch instruction operates in much the same manner as the other BASIC interrupts seen in Chapter 4. Trapping is turned on by issuing a SPRITE ON command. When the branch to the subroutine occurs, trapping is automatically suspended by the issue of a SPRITE STOP command, followed by an automatic SPRITE ON on return from the subroutine. Normally, the programmer would include his own SPRITE OFF and SPRITE ON statements in the interrupt servicing routine.

This interrupt cannot decide *which* sprites collided in the way that ON KEY GOSUB could decide which function key was pressed. It is

therefore up to the programmer to structure programs in a manner that allows the sprite collision to be interpreted. Program 9.5 illustrates this.

You control a spaceship which may be moved up and down. You may fire missiles at a roving alien ship which also fires at you. The program is structured so that only the following sprite collisions may occur:

1. Alien ship + your missile.
2. Your ship + alien missile.
3. Alien missile + your missile.

Lives are lost when you are struck by an alien missile, and your score is increased when you hit the alien or one of its missiles.

Program 9.5

```
10 REM ************************
20 REM *                      *
30 REM * Game Using Sprite    *
40 REM *     INTERRUPTS       *
50 REM *                      *
60 REM ************************
70 DEFINT A-Z
80 OPEN "GRP:" AS #1
90 DIM M(100)
100 DIM T(7)
110 R=RND(-TIME)
120 ON STRIG GOSUB 1000
130 ON SPRITE GOSUB 1250
140 ON INTERVAL=10 GOSUB 1110
150 ON SPRITE GOSUB 1250
160 COLOR 15,1,1
170 SCREEN 2,0,0
180 REM *
190 REM * Initialize Sprites
200 REM *
210 FOR I = 0 TO 3
220 S$ = ""
230 FOR J = 1 TO 8
240 READ S : S$=S$+CHR$(S)
250 NEXT J
260 SPRITE$(I)=S$
270 NEXT I
280 REM *
290 REM * Draw Background
```

```
300 REM *
310 DRAW "BM0,176"
320 DRAW "S4A0"
330 LINE(0,0)-(255,191),15,B
340 LINE(16,2)-(239,2),15
350 LINE(16,2)-(239,189),15,B
360 LINE(16,16)-(239,16),15
370 DRAW "BM136,5":COLOR 15
380 T$="Score: ":GOSUB 920
390 SC = 0
400 PRINT#1,USING "######";SC
410 REM *
420 REM * Plot "Stars"
430 REM *
440 FOR I=1 TO 100
450 E = INT(RND(1)*224)+16
460 F = INT(RND(1)*160)+16
470 PSET(E,F)
480 NEXT I
490 REM *
500 REM * Set Up Random Movement
510 REM * For Alien Ship
520 REM *
530 FOR I=1 TO 100
540 M(I)=INT(RND(1)*130)+16
550 NEXT I
560 CIRCLE(160,60),40,11:PAINT (160,60),
11
570 CIRCLE(175,45),10,9,,,13/10:PAINT
(175,45),9
580 REM *
590 REM * Place Markers
600 REM *
610 DRAW "BM10,4"
620 FOR I=10 TO 13
630 PUT SPRITE I,STEP(10,0),15,1
640 NEXT
650 X=24:Y=96:LI=14
660 F$="T180V1202S11M1000E16R16D#2"
670 M$="T180V1206L24BGBGAB"
680 W$="T180S1M300003C"
690 MX=200:MY=96:IX=IX+1:T=-2:F=0:J=0
700 STRIG(0) ON
710 SPRITE ON
```

```
720 INTERVAL ON
730 IF IX>100 THEN IX=0
740 L = SGN(M(IX)-MY)
750 REM *
760 REM * Main Loop
770 REM *
780 PUT SPRITE 0,(X,Y),15,1
790 PUT SPRITE 1,(MX,MY),15,0
800 IF F THEN GOSUB 1040
810 IF J THEN GOSUB 1160
820 IF STICK(0)=1 THEN Y=Y-2
830 IF STICK(0)=5 THEN Y=Y+2
840 IF Y<=16 THEN Y=16
850 IF Y>=176 THEN Y=176
860 MY=MY+L:IF  MY=M(IX) THEN IX=IX+1:
GOTO 730
870 MX=MX-T:IF MX<40 OR MX>228 THEN
T=-T
880 GOTO 780
890 REM *
900 REM * Print to Screen Routine
910 REM *
920 FOR I = 1 TO LEN(T$)
930 PRINT#1,MID$(T$,I,1);
940 DRAW "BM-2,0"
950 NEXT
960 RETURN
970 REM *
980 REM * Plot Friendly Missile
990 REM *
1000 IF F THEN RETURN
1010 PLAY "L8S8M30006C","L8S8M30007C"
1020 A=Y:B=X+16:F=1
1030 RETURN
1040 B=B+7
1050 IF B>232 THEN PUT SPRITE 3,(0,209):
F=0:RETURN
1060 PUT SPRITE 3,(B,A),15,2:RETURN
1070 RETURN
1080 REM *
1090 REM * Plot Unfriendly Missile
1100 REM *
1110 IF J THEN RETURN
1120 PLAY "L16S14M80002A"
```

```
1130 BX=MX-8:BY=MY+4
1140 PUT SPRITE 4,(BX,BY),8,3
1150 J=-1 : STRIG(0) ON:RETURN
1160 BX=BX-5
1170 IF BX<16 THEN BY=209: J=0
1180 PUT SPRITE 4,(BX,BY),8,3
1190 RETURN
1200 REM *
1210 REM * Interrupt Handling Routine
1220 REM * Work out which sprites
1230 REM * by reference to position
1240 REM *
1250 SPRITE OFF : INTERVAL OFF
1260 IF A+2>=MY AND A+5<=MY+7 AND B+8>=
MX THEN SI=30:PLAY M$:GOTO 1300
1270 IF BY+2>Y AND BY-7<Y AND BX<34 THEN
LI=LI-1:PLAY F$:GOTO 1300
1280 IF A+2>=BY AND A+5<=BY+7 AND BY+8>=
36 THEN SI=10:PLAY W$:GOTO 1400
1290 SPRITE ON:INTERVAL ON:STRIG(0) ON:
RETURN
1300 REM *
1310 REM * Screen Flash
1320 REM *
1330 FOR I=0 TO 50
1340 COLOR ,,RND(1)*16
1350 NEXT
1360 MY=96:MX=220
1370 REM *
1380 REM * Remove Missiles
1390 REM *
1400 PUT SPRITE 3,(0,209):F=0:A=0:B=209
1410 PUT SPRITE 4,(0,209):J=0:BX=0:BY=
209
1420 IX=IX+1
1430 REM *
1440 REM * Update Score
1450 REM *
1460 DRAW "BM178,5":COLOR 1
1470 PRINT#1,USING "#######";SC
1480 SC=SC+SI
1490 DRAW "BM178,5":COLOR 15
1500 PRINT#1,USING "#######";SC
1510 COLOR ,,1
```

```
1520 IF LI=9 THEN GOTO 1590
1530 PUT SPRITE LI,(0,209)
1540 SPRITE ON:INTERVAL ON
1550 RETURN  730
1560 REM *
1570 REM * End of Game
1580 REM *
1590 STRIG(0) OFF:INTERVAL OFF:SPRITE
OFF
1600 PUT SPRITE 0,(0,209)
1610 PUT SPRITE 1,(0,209)
1620 PUT SPRITE 3,(0,209)
1630 PUT SPRITE 4,(0,209)
1640 LINE (24,60)-(232,76),15,BF
1650 DRAW "BM26,64":COLOR 1
1660 T$="GAME OVER - Press UP for new ga
me"
1670 GOSUB 920
1680 FOR I=1 TO 5000:NEXT
1690 S=STICK(0) :IF S=0 THEN 1690
1700 IF S<>1 THEN 1720
1710 CLS : GOTO 310
1720 COLOR 15,4,4
1730 END
1740 REM *
1750 REM * Sprite Data
1760 REN *
1770 DATA 0,60,66,255,66,255,66,60
1780 DATA 0,224,68,255,255,68,224,0
1790 DATA 0,64,34,63,34,64,0,0
1800 DATA 0,2,68,252,68,2,0,0
```

This sort of game cruelly exposes the great weakness of BASIC — it is generally quite a slow language. Sprites speed things up a bit, but it is clear that no arcade classics are going to be written without recourse to machine code!

Programmed screen design

Sprites are used extensively in Program 9.6. Designing a graphics screen is a tedious process, so this program can be used to draw screens. When the picture is complete, a BASIC program listing can be produced or SAVEd to tape. A saved program can then be MERGEd or LOADed when required.

There are only a few drawing commands to worry about. These are:

L Pressing L defines the start point for a line. A second press draws the line.
P This key paints in an area of the screen.
U This will 'undo' the last command used.
DEL Deletes the program being created from memory and clears the screen.
ESC Generates the program.

One sprite is used as a cursor while another is used as a marker for the start point of a line. A further six sprites are used to indicate the cursor's current X and Y position on the screen. When the cursor touches any of these 'coordinate' sprites, they are moved out of the way. The latter feature is accomplished using the ON SPRITE GOSUB interrupt.

Program 9.6

```
10 GOTO 180
20 REM *
30 REM * Update Coordinate Display
40 REM * (Subroutine)
50 REM *
60 PUT SPRITE 0,(16,P),1,H
70 PUT SPRITE 1,(22,P),1,T
80 PUT SPRITE 2,(28,P),1,U
90 PUT SPRITE 3,(16,P+8),1,H1
100 PUT SPRITE 4,(22,P+8),1,T1
110 PUT SPRITE 5,(28,P+8),1,U1
120 RETURN
130 REM ************************
140 REM *                      *
150 REM * Screen Generator     *
160 REM *                      *
170 REM ************************
180 REM *
190 REM * Graphics Program Generator
200 REM *
210 SCREEN 0 : KEY OFF : WIDTH 38
220 DEFINT A-Z
230 MAXFILES = 2
240 DIM BUF(100,5)
```

```
250 REM *
260 REM * Open Program File
270 REM *
280 PRINT "SCREEN GENERATOR"
290 PRINT
300 PRINT "Ensure PLAY and RECORD are se
t."
310 PRINT : PRINT
320 PRINT "Enter File Name: ";
330 A$=INPUT$(1)
340 IF A$=CHR$(13) AND F$<>"" THEN 390
350 A$ =CHR$(8) AND POS(0)-1>16 THEN
LOCATE POS(0)-1:PRINT CHR$(32);:LOCATE
POS(0)-1:F$=LEFT$(F$,LEN(F$)-1):GOTO330
360 IF LEN(F$)=6 THEN GOTO 330
370 IF (A$>="A" AND A$<="Z") OR (A$>="a"
AND A$<="z") THEN F$=F$+A$ ELSE GOTO330
380 PRINT A$; : GOTO 330
390 PRINT : PRINT
400 PRINT USING "Opening File '&' now...
";F$
410 PRINT : PRINT
420 OPEN F$ FOR OUTPUT AS #2
430 PRINT USING "File '&' Open";F$
440 PRINT : PRINT
450 PRINT "Press Any Key to Continue"
460 A$=INPUT$(1)
470 SCREEN 2,0,0
480 GOSUB 800
490 IX=-1:P=8:Q=168:X=128:Y=96
500 H=1:T=2:U=8:H1=0:T1=9:U1=6
510 MX=1:MY=1:HX=8:HY=8:LX=0:LY=0
520 ON SPRITE GOSUB 1780 : SPRITE ON
530 PUT SPRITE 6,(X,Y),1,10
540 GOSUB 60
550 A$=INKEY$:IF A$="" THEN 550
560 IF A$=CHR$(127) THEN IX=-1:CLS
570 IF A$=CHR$(27) THEN GOTO 1320
580 IF A$="P" THEN GOSUB 910
590 IF A$="L" THEN GOSUB 1000
600 IF A$="U" THEN GOSUB 1160
610 D=STICK(0) : IF D=0 THEN 550
620 IF D=1 THEN Y=Y-MY
```

```
630 IF D=2 THEN Y=Y-MY:X=X+MX
640 IF D=3 THEN X=X+MX
650 IF D=4 THEN X=X+MX:Y=Y+MY
660 IF D=5 THEN Y=Y+MY
670 IF D=6 THEN Y=Y+MY:X=X-MX
680 IF D=7 THEN X=X-MX
690 IF D=8 THEN X=X-MX:Y=Y-MY
700 IF X>255 THEN X=255
710 IF Y>191 THEN Y=191
720 IF X<0 THEN X=0
730 IF Y<0 THEN Y=0
740 H=INT(X/100):T=(XMOD100)/10:U=X MOD
10
750 H1=INT(Y/100):T1=(YMOD100)/10:U1=Y
MOD 10
760 GOTO 530
770 REM *
780 REM * Initialize Sprites
790 REM *
800 FOR I=0 TO 11
810 S$ = ""
820 FOR J=1 TO 8
830 READ A$ : S$=S$+CHR$(VAL("&H"+A$))
840 NEXT J
850 SPRITE$(I)=S$
860 NEXT I
870 RETURN
880 REM *
890 REM * Paint Subroutine
900 REM *
910 IF POINT(X,Y)=15 THEN RETURN
920 IF IX+1 = 500 THEN RETURN
930 IX=IX+1
940 PAINT(X,Y)
950 BUF(IX,0)=1:BUF(IX,1)=X:BUF(IX,2)=Y
960 RETURN
970 REM *
980 REM * Line Drawing Routine
990 REM *
1000 IF L=0 THEN 1070
1010 L=0
1020 LINE(X,Y)-(BUF(IX,1),BUF(IX,2))
1030 IF (X=BUF(IX,1)) AND (Y=(BUF(IX,2))
```

```
THEN BUF(IX,0)=2:PUT SPRITE 7,(0,209):
SPRITE ON:RETURN
1040 BUF(IX,3)=X : BUF(IX,4)=Y
1050 PUT SPRITE 7,(0,209)
1060 BUF(IX,0)=3:SPRITE ON:RETURN
1070 IF IX+1=100 THEN RETURN
1080 IX=IX+1
1090 L=1:SPRITE OFF
1100 BUF(IX,1)=X:BUF(IX,2)=Y
1110 PUT SPRITE 7,(X-1,Y-2),1,11
1120 RETURN
1130 REM *
1140 REM * Undo
1150 REM *
1160 IX=IX-1
1170 CLS
1180 IF IX<=-1 THEN IX=-1:BUF(0,0)=0:
RETURN
1190 FOR I=0 TO IX
1200 ON BUF(I,0) GOSUB 1230,1250,1270
1210 NEXT
1220 RETURN
1230 PAINT(BUF(I,1),BUF(I,2))
1240 RETURN
1250 PSET(BUF(I,1),BUF(I,2))
1260 RETURN
1270 LINE(BUF(I,1),BUF(I,2))-(BUF(I,3),B
UF(I,4))
1280 RETURN
1290 REM *
1300 REM * Program Generation
1310 REM *
1320 SCREEN 0
1330 PRINT "LIST to Printer (P)
1340 PRINT "LIST to Screen  (S)
1350 PRINT:PRINT
1360 PRINT "Select ( );
1370 LOCATE 8,4
1380 A$=INKEY$ : IF A$="" THEN 1380
1390 PRINT A$;
1400 IF A$<>"P" AND A$<>"S" THEN BEEP :
GOTO 1370
1410 IF A$="P" THEN OPEN "LPT:" AS #1
```

```
1420 IF A$="S" THEN OPEN "CRT:" AS #1
1430 CLS
1440 REM *
1450 REM * Write out Program
1460 REM *
1470 FOR D = 1 TO 2
1480 PRINT #D,"10 REM *"
1490 PRINT #D,USING "20 REM * Program Na
me: &";F$
1500 PRINT #D,"30 REM *"
1510 PRINT #D,"40 SCREEN 2"
1520 LN = 50
1530 FOR I = 0 TO IX
1540 PRINT #D,MID$(STR$(LN),2);SPC(1);
1550 ON BUF(I,0) GOSUB 1610,1650,1690
1560 LN=LN+10
1570 NEXT I
1580 PRINT #D,MID$(STR$(LN),2);SPC(1); "
GOTO ";MID$(STR$(LN),2)
1590 NEXT D
1600 END
1610 T$=MID$(STR$(BUF(I,1)),2)
1620 U$=MID$(STR$(BUF(I,2)),2)
1630 PRINT #D, USING "PAINT(&,&)";T$;U$
1640 RETURN
1650 T$=MID$(STR$(BUF(I,1)),2)
1660 U$=MID$(STR$(BUF(I,2)),2)
1670 PRINT #D, USING "PSET(&,&)";T$;U$
1680 RETURN
1690 T$=MID$(STR$(BUF(I,1)),2)
1700 U$=MID$(STR$(BUF(I,2)),2)
1710 V$=MID$(STR$(BUF(I,3)),2)
1720 W$=MID$(STR$(BUF(I,4)),2)
1730 PRINT #D, USING "LINE(&,&)-(&,&)";T
$;U$,V$,W$
1740 RETURN
1750 REM *
1760 REM * Sprite Interrupt Routine
1770 REM *
1780 SPRITE OFF
1790 SWAP P,Q
1800 GOSUB 60
1810 FOR D=1 TO 100 : NEXT D
1820 SPRITE ON
```

```
1830 RETURN
1840 DATA 70,88,98,A8,C8,88,70,0
1850 DATA 20,60,20,20,20,20,70,0
1860 DATA 70,88,08,30,40,80,F8,0
1870 DATA F8,08,10,30,08,88,70,0
1880 DATA 10,20,50,90,F8,10,10,0
1890 DATA F8,80,F0,08,08,88,70,0
1900 DATA 38,40,80,F0,88,88,70,0
1910 DATA F8,08,10,20,40,40,40,0
1920 DATA 70,88,88,70,88,88,70,0
1930 DATA 70,88,88,78,08,10,E0,0
1940 DATA 80,40,20,10,8,4,2,1
1950 DATA 40,E0,40,0,0,0,0,0
```

Frame-by-frame animation

The form of animation used so far has been rather static — although the position of an object has been moved, the shape of the object has not been changed in any way. We can also create the illusion of movement by repeatedly changing the sprite pattern displayed in a single position.

The running speed of cinema film is about 24 frames per second. Using all 256 sprite patterns, by displaying a new pattern every 1/24th of a second, an animation sequence over 10 seconds in length can be created. A shorter, but more detailed sequence (about 2.5 seconds in length), can be created using the 64 large sprites.

By using the ON INTERVAL interrupt, 1/24 second update can be approximated quite easily. Program 9.7 shows a simple animated sequence.

Program 9.7

```
10 REM *
20 REM * Sprite Animation
30 REM *
40 COLOR 15,15,1
50 SCREEN 2,2
60 LINE (60,112)-(212,112),1
70 FOR I = 1 TO 4
80 S$=""
90 FOR J=1 TO 32:READ A:S$=S$+CHR$(A) :
NEXT
100 SPRITE$(I)=S$
110 NEXT
120 X=60 : SX=1 : I=1
```

```
130 PUT SPRITE 0,(X,96),1,I
140 X=X+SX:I=I+1
150 IF X<60 OR X>196 THEN SX=-SX
160 IF I>4 THEN I=1
170 FOR J=1 TO 25 : NEXT J
180 GOTO 130
190 DATA 0,2,5,5,2,19,15,3
200 DATA 3,3,2,2,2,4,4,6
210 DATA 0,0,0,0,0,128,64,32
220 DATA 0,16,240,0,0,0,0,0
230 DATA 0,2,5,5,2,3,7,11
240 DATA 19,3,2,2,2,2,2,3
250 DATA 0,0,0, 0,0,224, 0,0
260 DATA 0,0,128,64,40,16,0,0
270 DATA 0,2,5,5,2,3,7,11
280 DATA 11,11,2,2,2,2,2,3
290 DATA 0,0,0,0,0,32,192,0
300 DATA 0,0,128,64,64,64,96,0
310 DATA 0,2, 5, 5, 2, 3, 7, 11
320 DATA 19,19,2,2,2,3,2,3
330 DATA 0,0,0,0,64,64,192,0
340 DATA 0,0,128,64,128,0,128,0
```

The video random access memory (VRAM)

In addition to the memory used by MSX–BASIC, programs and variables, every MSX computer has 16K of memory dedicated to the video display processor. It is very fast access memory (i.e., with low read and write times), and contains the information needed to maintain the screen display. Information is arranged in the form of tables, with each table responsible for some aspect of the screen's appearance.

In this section, the means of accessing VRAM will be discussed, along with a detailed examination of its contents for each screen mode.

Reading and writing to VRAM

VRAM is composed of 16384 8-bit memory locations with addresses 0–16383. A value can be read from any of these locations using the special function VPEEK( ). The address of the VRAM location to be read is given as an argument, and the current contents of this location are returned.

A complementary command is VPOKE which writes a value into a VRAM location. The values to be written must be integers in the range 0–255.

The 40×24 text screen

SCREEN 0 is the least greedy of all screen modes as far as video memory is concerned. A total of 4K of VRAM is set aside when this screen is selected, of which only 3008 bytes are actually used. The memory is divided into two tables: the **pattern name table** and the **pattern generator table**.

The pattern name table is 960 bytes long and starts at address 0000. This text screen has a total of 960 positions where characters may be placed. Each one of these positions has an associated byte in the pattern name table: the top left-hand position of the screen is linked with byte 0000, the top right with byte 0039 and the bottom right with byte 0959.

A pattern name table entry contains the code of the pattern currently displayed at the corresponding screen position. For example, if the top left-hand position displays 'A', then byte 0000 contains the number 65. This code number is the pattern number for the character 'A'.

The pattern generator table contains the pattern definitions for all the 256 patterns that may be displayed. Each pattern definition requires 8 bytes of data (like 8×8 sprites), so this table is 2048 bytes long and starts at address 2048.

What an entry in the pattern name table does is to *reference* a pattern in the generator table. The start of a pattern definition in VRAM can be found by multiplying a pattern name table entry by 8 (as there are 8 bytes of definition data for each pattern), and adding 2048 (the start of the pattern generator table). Using the pattern number 65 as our example, the start address for this pattern is given thus:

$$2048 + (8 \times 65) = 2568$$

Looking at the binary representation of the pattern definition for 'A' (addresses 2568 to 2575) we see:

```
00100000
01010000
10001000
10001000
11111000
10001000
10001000
00000000
```

One thing we can do with these tables is to alter their contents. Program 9.8 redefines the alphabetic characters, both upper and lower case to produce a tiny character set. This new definition data is VPOKEd into the pattern generator table. Only the upper six bits of the pattern data are significant for this screen mode.

Program 9.8

```
10 REM *
20 REM * Tiny Character Loader
30 REM *
40 SCREEN 0 : KEY OFF : PLAY"T255L6404"
50 PRINT "Loading Data" : PRINT
60 FOR I = 2568 TO 2775
70 READ A$ : VPOKE I,VAL("&H"+A$)
80 VPOKE I+256,VAL("&H"+A$)
90 NEXT I
100 PLAY "CD"
110 REM *
120 REM * Alphabetic Data
130 REM *
140 DATA 00,00,70,88,f8,88,88,00:'A
150 DATA 00,00,f0,48,70,48,f0,00:'B
160 DATA 00,00,78,80,80,80,78,00:'C
170 DATA 00,00,f0,88,88,88,f0,00:'D
180 DATA 00,00,f0,80,e0,80,f0,00:'E
190 DATA 00,00,f0,80,e0,80,80,00:'F
200 DATA 00,00,78,80,B8,88,70,00:'G
210 DATA 00,00,88,88,F8,88,88,00:'H
220 DATA 00,00,70,20,20,20,70,00:'I
230 DATA 00,00,70,20,20,A0,E0,00:'J
240 DATA 00,00,90,A0,C0,A0,90,00:'K
250 DATA 00,00,80,80,80,80,F8,00:'L
260 DATA 00,00,88,D8,A8,88,88,00:'M
270 DATA 00,00,88,C8,A8,98,88,00:'N
280 DATA 00,00,F8,88,88,88,F8,00:'O
290 DATA 00,00,F0,88,F0,80,80,00:'P
300 DATA 00,00,F8,88,A8,90,E8,00:'Q
310 DATA 00,00,F8,88,F8,A0,90,00:'R
320 DATA 00,00,78,80,70,08,F0,00:'S
330 DATA 00,00,F8,20,20,20,20,00:'T
340 DATA 00,00,88,88,88,88,70,00:'U
350 DATA 00,00,88,88,90,A0,40,00:'V
360 DATA 00,00,88,88,A8,D8,88,00:'W
370 DATA 00,00,88,50,20,50,88,00:'X
380 DATA 00,00,88,50,20,20,20,00:'Y
390 DATA 00,00,F8,10,20,40,F8,00:'Z
```

Program 9.9 generates giant sized characters made up of asterisks. The contents of a series of locations in the name table are read, and the

large characters are produced by looking up the appropriate pattern definition.

Program 9.9

```
10 REM *
20 REM * Banner
30 REM *
40 CLS
50 DIM A(80)
60 SCREEN 0 : KEY OFF : WIDTH 40
70 PRINT "Type in twenty characters"
80 FOR I=1 TO 20
90 A$=INPUT$(1)
100 PRINT A$;
110 NEXT
120 PRINT
130 REM *
140 REM * Send DECREASE LINE FEED code
150 REM *
160 LPRINT CHR$(27);"3";CHR$(18);
170 REM *
180 REM * Main Printing Loop
190 REM *
200 FOR L=1 TO 20
210 X=2048+(VPEEK(39+L)*8)
220 FOR I=7 TO 2 STEP -1
230 FOR J=7 TO 0 STEP -1
240 IF (VPEEK(X+(J)) AND (2^I))<>0 THEN
LPRINT "*"; ELSE LPRINT SPACE$(1);
250 NEXT J
260 LPRINT
270 NEXT I
280 NEXT L
```

The 32×24 text screen

SCREEN 1 uses more video memory as sprites may be used in this mode. The pattern name and generator tables perform the same function as before, and are located at addresses 6144 and 0000 respectively.

Colour usage is less restrictive in this mode. 32 bytes of VRAM are set aside as the **colour table**.

The pattern generator table may be divided up into 32 groups of 8

pattern definitions, i.e., patterns Ø–7, 8–15 and so forth. For each of these blocks there is an entry in the colour table. The upper four bits determine the foreground colour when the pattern is displayed, while the lower four bits describe the background colour.

The colour table starts at address 8192. Program 9.1Ø demonstrates how this table may be used to alter the display colour.

Program 9.1Ø

```
10 REM *
20 REM * Colour Manipulation
30 REM *
40 SCREEN 1
50 COLOR 15,4,4
60 FOR I=0 TO 255:PRINT CHR$(I);:NEXT
70 FOR I=0 TO 255
80 FOR J=8192 TO 8223
90 VPOKE J,I
100 FOR K=1 TO 25:NEXT K
110 NEXT J
120 NEXT I
```

The final point worth noting is that all eight bits of a pattern generator table entry are significant in this mode.

The low-resolution screen

This mode also has pattern name and generator tables, but their purpose is defined rather differently. This time the name table references two bytes in the generator table which determine the colour of four pixels on the screen. The way the contents of the generator affects the screen is shown in Figure 9.4.

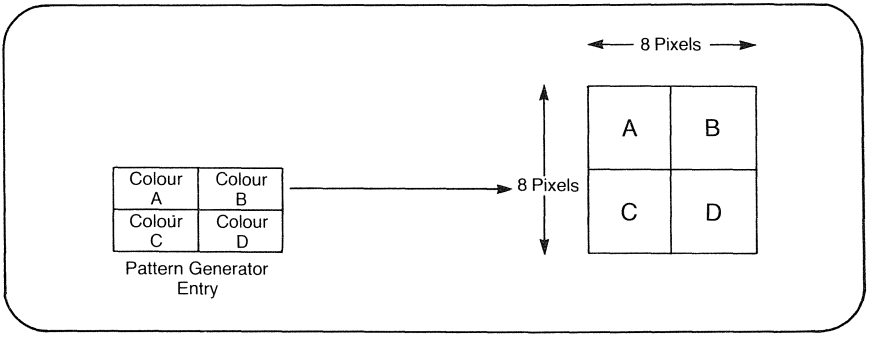


Figure 9.4 The lo-res screen and the pattern generator

To be honest, this screen mode is not one of MSX–BASIC's finer points, and meddling with VRAM in this case is unlikely to be worth the trouble. By way of a demonstration, Program 9.11 fills the screen with a black and white chequerboard pattern.

Program 9.11

```
10 REM *
20 REM * Chequerboard
30 REM *
40 COLOR 15,1,1
50 CLS
60 SCREEN 3
70 FOR I = 0 TO 7 STEP 2
80 VPOKE I,241 : VPOKE I+1,31
90 NEXT
100 FOR I=2048 TO 2815
110 VPOKE I,0
120 NEXT
130 GOTO 130
```

The high-resolution screen

The pattern generator and colour tables for this mode are each 6144 bytes long, one byte for each of the possible 8×1 blocks on the screen.

A colour table entry defines colour in the following manner. If we have the pattern generator entry:

00110011

and the corresponding colour table entry is:

11110001

the pattern will be displayed as follows (where W is white and B is black):

BBWWBBWW

Where there is a '1' in the generator table entry, the upper nibble of the colour entry gives that pixel's colour. Where a bit position is '0' the pixel's colour is determined by the lower nibble.

The relationship between the name and generator table is less straightforward. The screen may be divided horizontally into thirds.

1. Patterns 0–255 for the top third of the screen are found in the first third of the generator table.
2. Patterns 0–255 for the middle third of the screen are found in the second third of the generator table.

3. Patterns 0–255 for the bottom third of the screen are found in the last third of the generator table.

So the pattern definition used for a given pattern number will depend on the screen position where the pattern is to be placed.

As this screen mode is so well supported by BASIC, there is little virtue in fiddling with these tables. However, one obvious use of these tables is to fill the screen with a single pattern as shown in Program 9.12. This is much faster than using a series of PSET statements.

Program 9.12

```
10 REM *
20 REM * Hires Patterns
30 REM *
40 COLOR 15,1,1
50 CLS
60 SCREEN 2
70 COLOR 15,1:CLS
80 FOR I = 0 TO 7
90 READ A : VPOKE I,A:VPOKE I+2048,A:VPO
KE I+4096,A
100 NEXT
110 FOR I=6144 TO 6144+767
120 VPOKE I,0
130 NEXT
140 FOR I=8192 TO 14435
150 VPOKE I,241
160 NEXT
170 GOTO 170
180 DATA &B11111111
190 DATA &B10011001
200 DATA &B10111101
210 DATA &B11100111
220 DATA &B11100111
230 DATA &B10111101
240 DATA &B10011001
250 DATA &B11111111
```

Sprites

All the screen modes that support sprites have two areas of VRAM set aside to deal with them. The first of these is the **sprite generator table**,

which is 32 or 8 bytes long for each sprite definition, depending of course on the sprite size selected. This performs the same function as any other pattern generator table.

The **sprite attribute** table is of much more interest. Each entry is four bytes long. Figure 9.5 shows the function of these bytes.

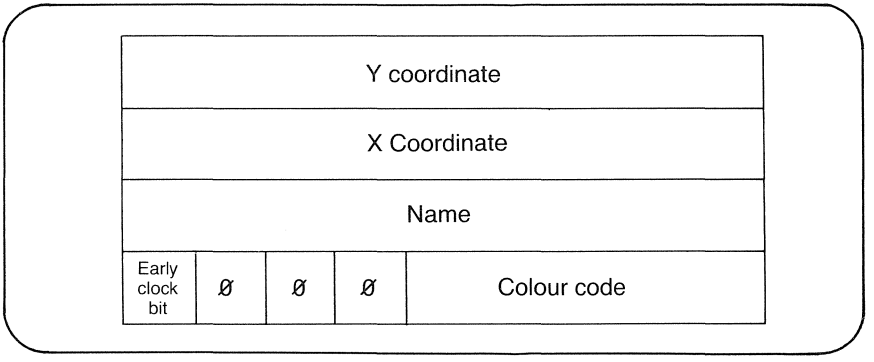


Figure 9.5 The sprite attribute table

The first and second bytes give the current vertical (Y) and horizontal (X) coordinates of the sprite. This gives the programmer an absolute reference to any sprite's position at any time. This is particularly useful when relative PUT SPRITE coordinate specifiers are used.

The third byte refers to the current pattern of the sprite. This is a value from Ø to 255 (or Ø to 63 for 16×16 sprites), from which the pattern definition start address can be determined.

The lower four bits of byte four of the attribute entry define the current sprite colour. The upper bit of this byte is known as the **early clock bit**. If it is set to '1', the sprite position is moved to the left by 32 pixels. This can be used to make a sprite appear miraculously on the screen as shown by Program 9.13.

Program 9.13

```
10 REM *
20 REM * Early Clock Bit
30 REM *
40 COLOR 1,15,15
50 SCREEN 1,3 : WIDTH 30
60 SPRITE$(0)=STRING$(32,255)
70 PUT SPRITE 0,(0,80),8,0
80 VPOKE 6915,&B10001000
90 LOCATE 2,8,0:PRINT "Press a Key"
```

```
100 A$=INPUT$(1)
110 VPOKE 6915,&B00001000
120 END
```

Sprites conclude this section on the Video RAM. The most useful aspects of this memory are seen in the text modes, where new character sets may be set up with ease. In the graphics modes, using the VRAM is a much more complicated process, and requires a great deal of practice to get used to.

BASE addressing

Although existing documentation refers to BASE as a function, BASE may more appropriately be thought of as an array variable. It stores the start (base) addresses for all the tables in VRAM for each of the screen modes. The meaning of each element of BASE and the default values for the base addresses are given in Table 9.1. The abbreviations used are:

PG Pattern generator table
PN Pattern name table
CT Colour table
SA Sprite attribute table
SP Sprite pattern table

Table 9.1

	Address	Table
BASE (0)	0000	PN SCREEN 0
BASE (2)	2048	PG SCREEN 0
BASE (5)	6144	PN SCREEN 1
BASE (6)	8192	CT SCREEN 1
BASE (7)	0000	PG SCREEN 1
BASE (8)	6912	SA SCREEN 1
BASE (9)	14436	SP SCREEN 1
BASE (10)	6144	PN SCREEN 2
BASE (11)	8192	CT SCREEN 2
BASE (12)	0000	PG SCREEN 2
BASE (13)	14436	SA SCREEN 2
BASE (15)	2048	PN SCREEN 3
BASE (17)	0000	PG SCREEN 3
BASE (18)	6912	SA SCREEN 3
BASE (19)	14436	SP SCREEN 3

You can alter these base addresses by assignment. This must be done with great caution, otherwise the screen display may be lost completely. (The simple cure for this is to reset the computer.)

The BASE( ) addresses must be set to values which fall on certain boundaries. The boundary limitations for VRAM tables are given in Table 9.2.

Table 9.2

VRAM Table	Boundary	Example Base Addresses
PN (text 0)	1K	1024, 6144, 7066
PN (text 1)		
PN (lo-res)		
PN (hi-res)		
PG (text 0)	2K	4096, 6144
PG (text 1)		
PG (lo-res)		
SP (any)		
CT (text 1)	64 bytes	0000, 128, 512
PG (hi-res)	8K	Only 0000 and 8192 allowed
SA (any mode)	128 bytes	2456, 4024, 14436

The most useful base address to change is that of the pattern generator table, particularly in SCREEN 0. This text mode uses very little VRAM, so there is enough room to define up to seven different character sets. Using the small character set defined in Program 9.8, but this time loading it at addresses beyond 4096 (the third 2K boundary), we can easily switch between the two character sets. Save Program 9.14 to tape *before* you run it in case the screen display is altered for the worse!

Program 9.14

```
10 REM *
20 REM * Tiny Character Loader
30 REM *
40 SCREEN 0 : KEY OFF : PLAY"T255L6404"
50 BASE(2)=2048
60 PRINT "Loading Data" : PRINT
70 FOR I = 4616 TO 4823
80 READ A$ : VPOKE I,VAL("&H"+A$)
90 VPOKE I+256,VAL("&H"+A$)
100 NEXT I
110 PLAY "CD"
120 PRINT "Press a key to change"
```

```
130 PRINT "the character set"
140 A$=INPUT$(1)
150 PLAY "L2405C06C"
160 BASE(2)=4096
170 REM *
180 REM * Alphabetic Data
190 REM *
200 DATA 00,00,70,88,f8,88,88,00:'A
210 DATA 00,00,f0,48,70,48,f0,00:'B
220 DATA 00,00,78,80,80,80,78,00:'C
230 DATA 00,00,f0,88,88,88,f0,00:'D
240 DATA 00,00,f0,80,e0,80,f0,00:'E
250 DATA 00,00,f0,80,e0,80,80,00:'F
260 DATA 00,00,78,80,B8,88,70,00:'G
270 DATA 00,00,88,88,F8,88,88,00:'H
280 DATA 00,00,70,20,20,20,70,00:'I
290 DATA 00,00,70,20,20,A0,E0,00:'J
300 DATA 00,00,90,A0,C0,A0,90,00:'K
310 DATA 00,00,80,80,80,80,F8,00:'L
320 DATA 00,00,88,D8,A8,88,88,00:'M
330 DATA 00,00,88,C8,A8,98,88,00:'N
340 DATA 00,00,F8,88,88,88,F8,00:'O
350 DATA 00,00,F0,88,F0,80,80,00:'P
360 DATA 00,00,F8,88,A8,90,E8,00:'Q
370 DATA 00,00,F8,88,F8,A0,90,00:'R
380 DATA 00,00,78,80,70,08,F0,00:'S
390 DATA 00,00,F8,20,20,20,20,00:'T
400 DATA 00,00,88,88,88,88,70,00:'U
410 DATA 00,00,88,88,90,A0,40,00:'V
420 DATA 00,00,88,88,A8,D8,88,00:'W
430 DATA 00,00,88,50,20,50,88,00:'X
440 DATA 00,00,88,50,20,20,20,00:'Y
450 DATA 00,00,F8,10,20,40,F8,00:'Z
```

Additional uses for VRAM

Saving space

If you are short of storage space at any time, integers or ASCII codes *could* be stored in VRAM. However, this method assumes that text mode 0 is used constantly — changing screen modes will wipe out your data. Program 9.15 illustrates the technique.

Program 9.15

```
10 REM *
20 REM * SCREEN 0 storage
30 REM *
40 TP=4096: '* top of available memory"
50 SCREEN 0:WIDTH 38
60 REM *
70 REM * Input Routine
80 REM *
90 INPUT "Name (* to finish)";N$
100 IF LEFT$(N$,1)="*" THEN 210
110 IF TP+LEN(N$)+1>16383 THEN PRINT "me
mory full" : GOTO 210
120 FOR I=1 TO LEN(N$)
130 VPOKE TP,ASC(MID$(N$,I,1))
140 TP=TP+1
150 NEXT
160 VPOKE TP,0:TP=TP+1
170 GOTO 90
180 REM *
190 REM * Output routine
200 REM *
210 CLS
220 I=4096
230 X=VPEEK(I)
240 IF X=0 THEN PRINT : GOTO 260
250 PRINT CHR$(X);
260 I=I+1
270 IF I<TP THEN 230
280 END
```

This releases around 12K of memory, but with the overhead of processing time for the VPEEK and VPOKE operations needed. If speed is not of the essence, then storing and reading values to VRAM could come in useful for very large programs.

The VDP registers

In common with the sound chip, there are a number of registers on the video chip that may be looked at and written to. The contents of each of these registers are shown in Figure 9.6.

The registers of most interest are 0, 1, 7 and the status register, as registers 2 to 6 duplicate information that may be found using BASE( ) .

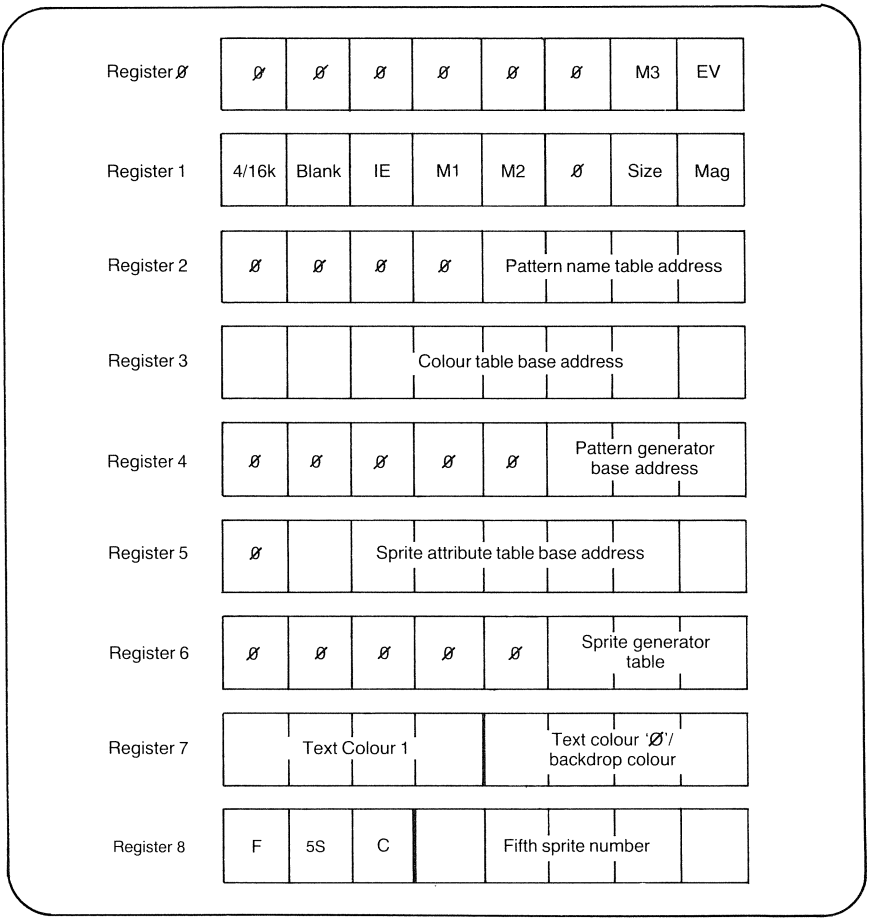


Figure 9.6 The VDP registers

Registers Ø and 1

The bits in register one have the following purposes:

B7 This determines the size of VRAM memory allocated in the system. If set to zero, then the video chip will assume it has 4K of VRAM. It is normally set to 1 to allow for the 16K of video memory needed for the graphics modes. There is no need to alter this bit.

B6 This bit is used to turn the screen display off (Ø) and on (1). By first turning the display off, a screen image can be plotted or printed without the user seeing it. The display can be turned back on when the plotting is finished.

B5 The interrupt generated by the VDP every 1/50th of a second may be turned off by setting this bit to Ø and turned on by setting it to 1. Again, there is no great need to alter this bit.

B4–3 These two bits, in combination with B1 of register Ø, may be read to determine the current screen mode. The combination of the three bits for each screen mode is:

M1	M2	M3	Mode
Ø	Ø	Ø	Text 1
Ø	Ø	1	High resolution
Ø	1	Ø	Low resolution
1	Ø	Ø	Text Ø

B1 The current sprite size used is given by this bit. It is set to 1 when 16×16 sprites are selected and to Ø for 8×8 sprites.

BØ This is the sprite magnification bit. When set to 1 the sprites are magnified.

Register 7

The main function of this register is to set the foreground and background colours for text mode Ø. The upper nibble contains the foreground colour for text mode Ø, and the lower nibble contains the background colour for all modes. So if the command COLOR 15,4 was given in text mode, register 7 would contain the value:

1111Ø1ØØ (244)

Register 8 — the status register

This is a read-only register. The bits are defined with the following purposes:

B7 When the VDP generates an interrupt signal, the bit is set to 1. It is reset whenever the status register is read. This is of no real use to the BASIC programmer.

B6 This bit is set to 1 whenever the fifth sprite rule is broken. The status of this bit can be polled to handle any incidences where sprites occur five or more in a line.

B5 When sprites collide, this bit will be set. This allows the programmer to poll for sprite collision instead of using ON SPRITE GOSUB interrupts.

B4–Ø When the fifth sprite rule is violated, the number of the offending sprite is placed in these 5 bits.

The VDP function

This allows all of the registers to be read and values to be written to all but register 8. Where a value is to be written, it is best to first read the current value of the register, and to perform a logical AND or OR to set or reset the bits as desired. Program 9.16 shows how screen blanking may be achieved by manipulation of register 1 (given by VDP(1)).

Program 9.16

```
10 REM *
20 REM * Screen Blanking
30 REM *
40 COLOR 15,4,4
50 SCREEN 2
60 VDP(1)=VDP(1) AND &B10111111
70 CIRCLE (128,96),90,15:PAINT (128,96)
80 PLAY "c
90 VDP(1)=VDP(1) OR &B01000000
100 GOTO 100
```

Program 9.17 shows how the status register (VDP(8)) can be monitored for sprite collision.

Program 9.17

```
10 REM *
20 REM * VDP Sprite Monitoring
30 REM *
40 SCREEN 2,2
50 SPRITE$(0)=STRING$(32,255)
60 PUT SPRITE 0,(112,80),1,0
70 PUT SPRITE 1,(10,80),8,0
80 IF (VDP(8) AND  32) = 32 THEN GOTO 11
0
90 PUT SPRITE 1,STEP(1,0),8,0
100 GOTO 80
110 BEEP:GOTO 110
```

This concludes the overview of the VDP in MSX computers. Great care must be taken when fiddling with both the VDP registers and VRAM. If there is a BASIC command that will carry out the operation you need *then use it*, as it is probably more convenient and also easier to understand.

Summary

Sprite associated commands

SPRITE$(⟨pattern number⟩)

PUT SPRITE ⟨plane number⟩,⟨coordinate specifier⟩ [,⟨colour⟩][,⟨pattern number⟩]

Video RAM commands and functions

VPEEK(⟨address in VRAM⟩)

VPOKE ⟨address in VRAM⟩,⟨value to be written⟩

BASE(⟨subscript⟩)

Video display processor commands and functions

VDP(⟨register number⟩)

Appendices

1 The remaining functions

This appendix completes the list of MSX–BASIC functions. Their syntax is listed in the normal manner along with a brief description of their purpose.

EXP (⟨X⟩) returns the value of the natural exponential function of a numeric expression, i.e., e^x. The value of X must not exceed 145.Ø6286Ø85862.

LOG (⟨X⟩) is a complementary function to EXP ( ) that returns the natural logarithm of the expression.

SQR (X) returns the square root of a given numeric expression.

PAD (N) reads touchpads attached to the joystick ports. The function reads port A if N is Ø, 1, 2 or 3, and port B if N is 4, 5, 6 or 7.

The value returned has the following meanings for each value of N:

Ø or 4	Returns Ø or −1 depending on whether the pad is touched or not.
1 or 5	Returns the X coordinate of the area touched.
2 or 6	Returns the Y coordinate of the area touched.
3 or 7	Returns the status of the pad's switch.

PDL (⟨N⟩) returns the current value of a paddle. If N is even, the paddle checked is that attached to port B, if odd, the paddle at port A is read. An integer in the range Ø–255 is returned.

PEEK (⟨address⟩) reads the contents of the specified byte in RAM. The address is supplied as an argument. Adding 65536 to a negative argument gives the true addresses of the byte to be read.

POKE ⟨address⟩,⟨value to be written⟩ writes a value into a specified byte in RAM. It is not a function, but is included here as it is complementary to PEEK ( ). It operates in the same way as VPEEK. Note that VRAM and system RAM are totally separate. POKE and PEEK have no effect on VRAM.

VARPTR (⟨variable name⟩) returns the address in memory where a particular variable is stored. If the address returned is negative, adding 65536 gives the true location of the variable.

For string data, VARPTR returns the *address* where the string is stored. Strings are stored in a separate area of RAM to normal variables. A variable must previously have been assigned a value before it may be submitted as a parameter for VARPTR ( ).

USR [⟨routine number⟩](⟨argument⟩) passes control to a machine code routine. This function is looked at in more detail in Appendix 6.

2 Error codes

All the possible error codes and messages that may be generated by the current version of the MSX interpreter are detailed here.

56 Bad file name.
52 Bad file number.
17 Can't CONTINUE.
19 Device I/O error.
57 Direct statement in file.
(If this message occurs with program LOADing, check that the cassette volume level is not set too high or too low.)
11 Division by zero.
54 File already OPEN.
59 File not OPEN.
12 Illegal direct.
5 Illegal function call.
55 Input past end.
51 Internal error.
25 Line buffer overflow.
24 Missing operand.
1 NEXT without FOR.
21 No RESUME.
4 Out of data.
7 Out of memory.
14 Out of string space.
6 Overflow.
22 RESUME without error.
3 RETURN without GOSUB.
1Ø Redimensioned array.
16 String formula too complex.
15 String too long.
9 Subscript out of range.
2 Syntax error.
13 Type mismatch.
8 Undefined line number.

18 Undefined user function.
20 Verify error.

Error codes 23, 26–49 and 60–255 are undefined in MSX–BASIC. If unused by user-defined error messages, they will produce the message:

Unprintable error.

3 Additional reserved words

All the BASIC commands and special variables are reserved words. There are some commands that the interpreter may recognize but may not yet interpret. These are primarily associated with the manipulation of disk-based files. The oddity is the reserved word MAXFILES which is in fact made up of two reserved words.

The additional reserved words are as follows:

ATTR$	CMD	COPY	DSKI$	DSKO$	FIELD
FILES	GET	IPL	KILL	LFILES	LSET
MAX	NAME	RSET	SET		

4 Logic tables

All the logical operators are summarized as logic tables.

NOT

X	NOT X
Ø	1
1	Ø

AND

X	Y	X AND Y
1	1	1
1	Ø	Ø
Ø	1	Ø
Ø	Ø	Ø

OR

X	Y	X OR Y
1	1	1
1	Ø	1
Ø	1	1
Ø	Ø	Ø

XOR

X	Y	X XOR Y
1	1	Ø
1	Ø	1
Ø	1	1
Ø	Ø	Ø

EQV

X	Y	X EQV Y
1	1	1
1	Ø	Ø
Ø	1	Ø
Ø	Ø	1

IMP

X	Y	X IMP Y
1	1	1
1	Ø	Ø
Ø	1	1
Ø	Ø	1

5 Frequency tables

This appendix lists the fine and coarse tune values which may be used with SOUND statements and their MML N command equivalents.

N	Fine tune	Coarse tune
0	85	0
1	156	12
2	231	11
3	60	11
4	155	10
5	2	10
6	115	9
7	235	8
8	107	8
9	242	7
10	128	7
11	20	7
12	175	6
13	78	6
14	244	5
15	158	5
16	78	5
17	1	5
18	186	4
19	118	4
20	54	4
21	249	3
22	192	3
23	138	3
24	87	3
25	39	3
26	250	2
27	207	2
28	167	2
29	129	2
30	93	2
31	59	2
32	27	2
33	253	1

N	Fine tune	Coarse tune
34	224	1
35	197	1
36	172	1
37	148	1
38	125	1
39	104	1
40	83	1
41	64	1
42	46	1
43	29	1
44	13	1
45	254	0
46	240	0
47	227	0
48	214	0
49	202	0
50	190	0
51	180	0
52	170	0
53	160	0
54	151	0
55	143	0
56	135	0
57	127	0
58	120	0
59	113	0
60	107	0
61	101	0
62	95	0
63	90	0
64	85	0
65	80	0
66	76	0
67	71	0
68	67	0
69	64	0
70	60	0
71	57	0
72	53	0
73	50	0
74	48	0
75	45	0
76	42	0
77	40	0
78	38	0
79	36	0
80	34	0
81	32	0
82	30	0
83	28	0

N	Fine tune	Coarse tune
84	27	0
85	25	0
86	24	0
87	22	0
88	21	0
89	20	0
90	19	0
91	18	0
92	17	0
93	16	0
94	15	0
95	14	0
96	13	0

6 Memory map and USR function

Memory map

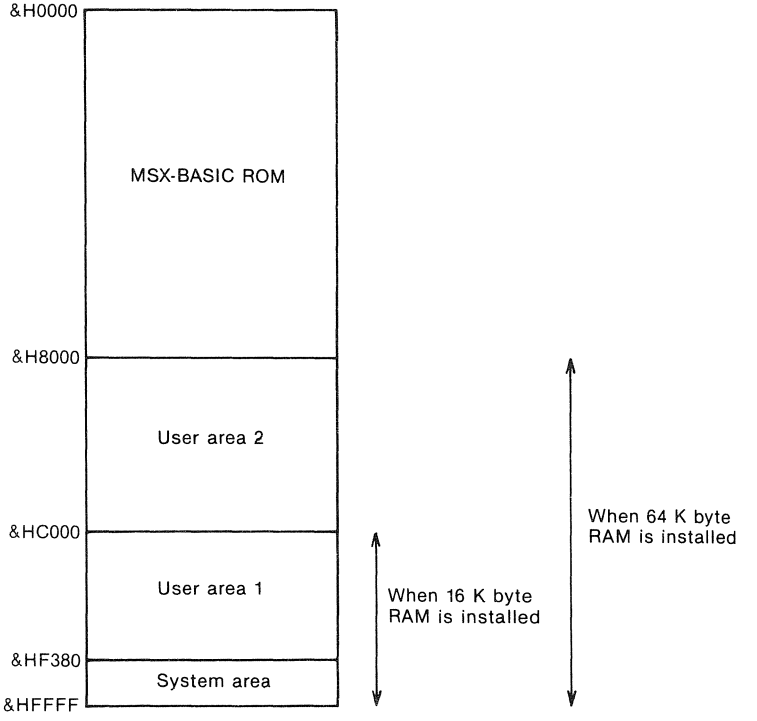


Figure A6.1 Memory map

The area of memory allocated for use by the system is fixed. The rest of memory (except for the ROM area) may be divided between machine code routines and BASIC programs. The CLEAR command is used to limit the memory that MSX–BASIC can use. If the command:

CLEAR 200,&H9FFF

were given, BASIC programs and variables etc., could occupy the space from the bottom of memory (&H8000 for a 64K system) up to and

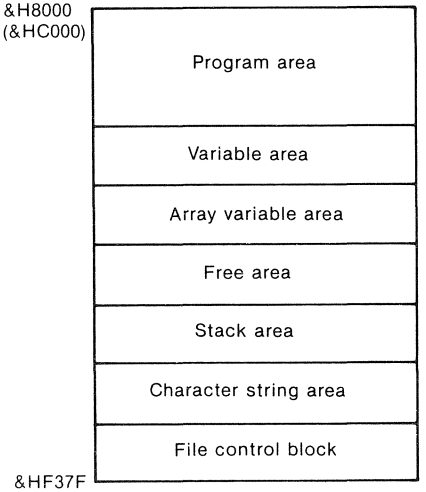


Figure A6.2 Layout of user area 1

including address &H9FFF. All the memory above this (&HAØØØ to &HF379) could be used for machine code programs.

DEFUSR statements define the starting addresses for up to 1Ø machine code subroutines. These routines are numbered Ø to 9. The default is Ø.

The statement:

DEFUSR1=&HAØØØ

defines the start address of routine number 1 to be at &HAØØØ.

Once defined, a machine code routine is used in the same manner as a function, i.e.:

⟨variable⟩ = USR ⟨subroutine number⟩(⟨argument⟩)

The value of the variable is **passed** to the machine code routine. When a routine is called, the number of bytes that the value occupies, thereby giving its type, is entered to the Z8Ø's A register, and the memory location &HF663. These values are:

2	Integers
4	Single precision numbers
8	Double precision numbers
3	Strings

The address of the variable is entered to the HL register.

Strings differ however. The address where the string is stored is entered into the DE register.

If the argument in a USR function call is zero, no values are passed to the machine code routine.

Index

MSX Programming is an ideal introduction to the fundamentals of MSX-BASIC programming incorporated in all micros adopting the standard MSX operating system, such as Sony, Hitachi, Sanyo, Canon, Panasonic and Toshiba.

The text develops to an advanced level and contains a large number of programs that demonstrate the different features of the language as they are introduced. Wherever possible, these are designed to be useful in their own right. Chapter five, for example, introduces a function and shows how it may be used to centre text on a display.

A special emphasis has been placed on graphics, one of the most interesting and dynamic features of microcomputing, and well supported by MSX BASIC. In particular, the advanced graphics chapter provides important documentation relating to the use of Video Memory and the Video Display Processor.

ISBN 0-273-02302-0