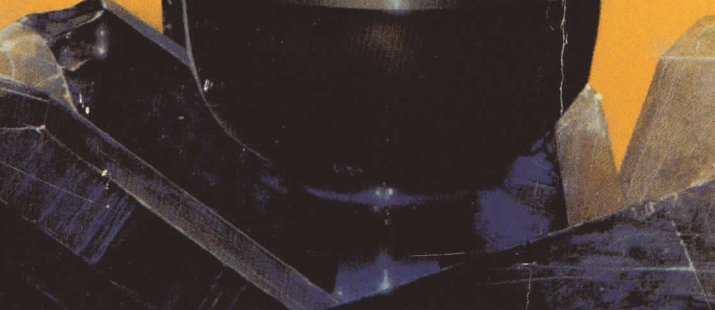
CENTURY COMMUNICATIONS

MSX

AN INTRODUCTION



Jonathan Pearce and Graham Bland

MSX
an introduction

GRAHAM BLAND AND JONATHAN PEARCE

CENTURY COMMUNICATIONS
LONDON

Copyright © Reflex Communications Limited 1984

All rights reserved

First published in Great Britain in 1984
by Century Communications Ltd,
12–13 Greek Street,
London W1V 5LE

ISBN 0 7126 0538 X

Printed in Great Britain by
Hazell, Watson & Viney Limited, Member of the BPCC Group,
Aylesbury, Bucks

Contents

Acknowledgements iv
Foreword v
Introduction viii
Introducing MSX 1
Programming in MSX-BASIC 20
Working with numbers 63
Interacting with your programs 93
MSX music and sound 107
Graphics using MSX-BASIC 119

Appendices
MSX-BASIC functions 153
Error codes and messages 157
Full screen editor control keys 161
Differences between SV-BASIC and MSX-BASIC 163

Acknowledgements

Jon would like to thank Jamie, for accepting that there was no such thing as Christmas 1983, nor were there pubs, nightclubs or friends throughout January and early February of 1984. He'd also like to thank his Mum and Dad for doing the things that Mums and Dads are generally good at; Jamie's dog Heidi, for continually resting her head on the keyboard of his typewriter, thereby making writing impossible; and Spectravideo, for "Frantic Freddie" and for making a computer that is so close to the MSX specification as makes no odds – except when you've got to write a book about it!

Graham would like to thank Janet Morrison for her scepticism and useful diversions; Algernon, the corporate frog for total devotion to elasticity at times of stress; numerous people for their enthusiasm and interesting discussions re. MSX, notably Toby Wolpe, Meiron Jones, Jim Lennox, Ben Woolley, Tony Hetherington and Basil Lane (for his advice on monitors). Finally, he would like to thank his parents for innumerable things, and Angus Annan and Arlen Michaels of the Microprocessor Group (University of Stirling) who taught him a lot about micros in a very short time.

Tom would like to thank Jane.

And finally, the authors would all like to thank Craig, Sue, Barbara, Tina and Karen for their support.

We are especially grateful to all at Microsoft in the U.K. who allowed us access to some of the very first MSX machines in this country.

Acknowledgements

Jon would like to thank Jamie, for accepting that there was no such thing as Christmas 1983, nor were there pubs, nightclubs or friends throughout January and early February of 1984. He'd also like to thank his Mum and Dad for doing the things that Mums and Dads are generally good at; Jamie's dog Heidi, for continually resting her head on the keyboard of his typewriter, thereby making writing impossible; and Spectravideo, for "Frantic Freddie" and for making a computer that is so close to the MSX specification as makes no odds – except when you've got to write a book about it!

Graham would like to thank Janet Morrison for her scepticism and useful diversions; Algernon, the corporate frog for total devotion to elasticity at times of stress; numerous people for their enthusiasm and interesting discussions re. MSX, notably Toby Wolpe, Meiron Jones, Jim Lennox, Ben Woolley, Tony Hetherington and Basil Lane (for his advice on monitors). Finally, he would like to thank his parents for innumerable things, and Angus Annan and Arlen Michaels of the Microprocessor Group (University of Stirling) who taught him a lot about micros in a very short time.

Tom would like to thank Jane.

And finally, the authors would all like to thank Craig, Sue, Barbara, Tina and Karen for their support.

We are especially grateful to all at Microsoft in the U.K. who allowed us access to some of the very first MSX machines in this country.

Foreword

The introduction of the MSX standard is unquestionably the most important event in the history of home computing. Indeed, its implications reach far beyond those of individual machines – even such outstandingly popular ones as the Sinclair ZX Spectrum.

Until now, the home computer market has been highly fragmented, made up of a number of largely or completely incompatible machines. Games and other software packages written for use on one machine will not run on another. Similarly, joysticks and other accessories designed for one machine cannot be connected to another. For the user, this means unnecessary difficulty and expense all the way. Although a mass of software exists for home micros, it is spread more or less evenly across the whole range of machines, so that only a fraction of available packages will run on each one. Even with the most popular machines, the user is not necessarily able to purchase a package perfectly suited to his or her specific needs.

MSX marks an end to problems like these, because it defines a common hardware specification which must be included in all MSX machines, and a common language – MSX-BASIC – in which applications and games packages can be written. The purchaser is therefore guaranteed that whichever MSX machine he chooses, absolutely any MSX software package will run on it – perfectly. This makes life a lot easier for software writers in one go, rather than rewriting and repackaging their software repeatedly for each of dozens of similar but incompatible machines. Even greater benefits arise for the user, who has access to an unparalleled range of software packages, which should be relatively inexpensive on account of the economies of scale involved.

The theory behind MSX is therefore a sound one, and one which will be of huge benefit to computer users. Indeed, it's no exaggeration to say that the arrival of the MSX standard will come to be recognised as a milestone on the way to universal computer literacy.

Interesting also is the motivation behind MSX, and the companies that have backed it. Unlike the home computer market in the UK and America, the Japanese market has so far been relatively undeveloped. In the spring of 1983, NEC and Matsushita approached Microsoft, one of the world's largest independent suppliers of microcomputer software, with the specification for a personal computer they were jointly developing. They wanted Microsoft to write a version of their industry-standard BASIC for the machine, with a view to developing sales in the Japanese domestic market. By June, a further 12 manufacturers, including one American company, had asked Microsoft for much the same thing.

To avoid the necessity of writing 14 different versions of BASIC and thus perpetuating the compatibility problems of the home micro market still further, Microsoft came up with the idea of the MSX standard. This includes commonly available processor chips and, of course, an optimised and extended version of Microsoft's BASIC interpreter. On June 16th 1983, Microsoft announced MSX, along with its support by Canon, Fujitsu, General, Hitachi, JVC, Kyocera, Matsushita, Mitsubishi, NEC, Pioneer, Sanyo, Sony, Spectravideo (US), Toshiba and Yamaha – all of them large, established companies in the consumer electronics markets.

What the announcement meant was that the entire Japanese computer and domestic electronics industry had committed itself to producing home computers built to a single specification. The first fruits of this appeared at the Japan Electronics Exhibition at Osaka in October 1983, when the ability to swap games cartridges between MSX machines from Hitachi, JVC, Matsushita, Mitsubishi, National, NEC, Sanyo, Sony and Toshiba caused a minor sensation.

The sheer strength with which the Japanese market their products, and the volumes in which they are able to produce them (initial production for MSX machines was 53,000 units per month), is imposing standardisation on the low-end of the computer industry in much the same way as cassette tapes revolutionised the home audio market in the early '70s.

Given the enormous rate at which MSX is growing, predictions as to what might happen next are obviously very difficult to make. However, the project has included a couple of milestones which permit a certain amount of speculation.

Firstly, on 5th October 1983, Microsoft announced MSX-DOS, an 8-bit operating system designed specifically for the MSX machines. Briefly, MSX-DOS not only allows MSX machines to support disk storage (thereby vastly increasing the amount of data they can hold), it also makes them upwards-compatible with the MS-DOS and XENIX operating systems, the operating systems used by many larger micros and minis.

In simple terms, this means that it will be possible for a user to run applications such as spreadsheet or word processing on his office micro and, at the end of the day, remove the floppy disk containing all of the data and take it home for completion on an MSX machine – an end to both duplication of work and late nights at the office!

Secondly, the BASIC supplied with MSX can be extended to control an almost unlimited range of add-ons. These include standard devices such as cassette recorders, printers, joysticks, games paddles and touch pads, as well as advanced devices such as voice synthesisers, FM-tuners and video disks. Machines that have already appeared support composite video and graphics, video cameras and recorders, light pens and robot arms. An early Yamaha machine has been designed to teach its user to play a synthesiser, and so comes complete with a full-range synthesiser keyboard. The first Sanyo machines incorporate a facility known as 'frame grabbing', which allows you to store a TV or video picture and subsequently alter it at will by adding your own graphics. There is even rumoured to be an MSX super hi-fi. and in principle there's no reason why every piece of domestic electronic equipment shouldn't be made MSX-compatible and controlled by a central MSX computer. MSX therefore opens up the way to the use of home computers in applications that have never been touched by computers before. It is worth bearing this in mind as you read this book, because the programming techniques we're introducing in the following pages will soon enable you to do much more than work out the compound interest on your building society account, or how economical your car is!

Introduction

MSX – An Introduction is the introductory book for new MSX users. We believe it'll teach you everything you need to know in order to write both enjoyable and useful programs in MSX-BASIC. That is not to say that you'll never want another book on MSX – everybody buys books with program listings to do specific things, and as you get more advanced, you may be interested in learning about MSX machine code, the MSX disk operating system, and so on.

The book has been structured to take you right the way from the very basics of programs – the names of the commands used in MSX-BASIC and their precise meanings and capabilities – through the concepts underlying program structure, right up to the complexities of advanced BASIC programming. We've also included chapters dedicated specifically to graphics and music. Both of these functions have separate, easily understood languages dedicated to them, which can be accessed from within MSX-BASIC, and which enable you to write your own games programs and tunes.

We shall begin the book, however, by introducing you to computers in general, and MSX computers in particular. From the preface to the book, you'll recognise that MSX marks something of a departure from existing home and personal computers. In the chapter which follows, we shall introduce you to the terminology that we'll be using to describe the computers, and introducing the rudiments of hardware, peripherals, and software.

We hope that you enjoy reading the book as much as we did writing it, and that you find the time spent worthwhile.

Jonathan Pearce
Graham Bland
Mark Adams
Tom Lewis
Reflex Communications Ltd

January 1984

Introducing MSX

Whatever your level of interest in home computers, the chances are that you already know a bit about how they work, or at the very least have heard some of the jargon associated with them. Our intention in this chapter is therefore to introduce you to the most commonly-used buzz words, and along the way explain exactly what differentiates your MSX computer from previous generations of home computers. We'll start off by looking at the different types of computers and the applications in which they're most commonly used, and then move on to the elements that go to make up all computer systems – the computer **hardware** itself, the **peripherals** that can be attached, and the **software** that runs on it.

Micros, minis and mainframes

Beyond any shadow of a doubt, it has been the advent of the **microcomputer** that has done more than anything else to promote the mass acceptance of computers. Inded, their impact has been so great that it is a widely held belief that to be 'computer literate' is one of the basic requirements of modern life, and that if someone can't grasp even the basics of computers they might as well give up!

Whilst the first home computers were really little more than extended calculators, they have now reached a high level of sophistication and a price that nearly everybody can afford. Indeed, they are now being put to real use, rather than just being played with, by millions of people in every corner of the earth. The very first MSX computers, for example, allow you not only to play games and peform ordinary calculations, but include specialist facilities to teach you how to play a synthesiser keyboard, or 'grab' a picture from your TV and add your own graphics-generated pictures to it. Capabilities such as these would have been unthinkable on home computers even a year ago, so it's easy to see what a quantum leap MSX represents (with no disrespect to Sir Clive Sinclair).

We must not forget, however, that home computers are no more than a small part of the microcomputer industry. Microcomputers are also making their presence felt in the business world, where they commonly handle tasks such as word processing, financial planning, accounts and stock control.

Moving one step up from micros, we come to **minicomputers**. These were very popular in the late 70s and early 80s, when they were seen as giving a practical alternative to **mainframes**. Mainframes are the enormous computers used mainly by large undertakings such as the Inland Revenue for storing massive amounts of information. Minicomputers, on the other hand, lie between micros and mainframes. They grew rapidly in popularity by allowing users to store information on a number of systems linked together by telephone lines, rather than obliging them to rely on a single enormous mainframe installation. In addition, the power of minicomputers was rapidly approaching that of mainframes, thereby permitting a truly decentralised base for information storage.

More recently, however, the increase in the power of the larger microcomputers has done to minicomputers what they did to mainframes a few years earlier. In other words, microcomputers have become progressively more powerful while their cost and physical size have fallen.

A manufacturer called Hewlett-Packard, for example, has even introduced what it calls 'A mainframe on a desk'. With the traditional boundaries between the different computers becoming increasingly blurred, micros are becoming a much more important part of the industry, to the extent that if you can learn how to use a micro, it's often unnecessary to go any further.

Finally, we ought to mention the so-called **supercomputers**. These are the sorts of computers you see in films like *2001 A Space Odyssey*, or *War Games*. Before the advent of the home micro, it was computers such as these that epitomised the public's view of computers – massive, threatening, and totally incomprehensible. As you progress through this book, we hope you'll come ot share our view of computers – that they're generally small, friendly, and actually very easy to use!

Having looked very briefly at the different types of computers and their various uses, we'll now move on to consider the parts that

go to make them up. In terms of its layout and the way its components function together, your MSX computer is similar to machines many times its size. Whilst other computers might do things in slightly different ways, the essentials of what's inside them, and the way they operate, are identical. We'll therefore use MSX as our basis for introducing the ideas to you, and we'll begin with the actual box you buy – the hardware.

Hardware

It's hard to conceive of anyone reading this book who has not heard of **microchips**. They appear to be cropping up everywhere you look, from toasters and microwave ovens to TVs and videos. They've even been built into cars to enable them to pass comment on their drivers! Indeed, it's difficult to think of a single electro-mechanical device that won't soon include a microchip.

Your MSX computer has three main microchips or **microprocessors**, each of which has a specific function to perform. The first, and most important of these is the **central processing unit**, or **CPU**, which in MSX machines is a Zilog Z80A (undoubtedly the most commonly used processor chip in home micros at the moment). The central processor is, if you like, the engine that drives the computer. It carries out all the mathematical functions and calculations necessary for the computer to function, and even has its own language in which it receives its instructions. This language is called **machine code**. No matter what language you use to talk to the computer – BASIC, LOGO, Assembly Language or whatever – it ultimately gets translated into the CPU's machine code. Languages like BASIC need a lot of translating and this takes time. Other languages, like Assembler, need little translation and can therefore run faster than BASIC. They are, however, very difficult and cumbersome to program with. Any games or sophisticated programs you might buy for your computer will probably be written in machine code.

The second major chip is the Texas Instruments TMS 9918A graphics processor. This is used by the computer to look after everything that appears on your TV screen. Many home computers have only one processor which has to do everything. With MSX computers, the central Z80 is used for processing and monitoring the keyboard only (so that it knows when you want it to do

something). The graphics processor and the General Instruments AY-3-8910 sound chip control the rest. This is why MSX computers have such excellent graphics and music capabilities, as we'll be demonstrating later. There are numerous programs in the sections on the Graphics and Music Macro Languages which show exactly how these capabilities can be put to use.

Memory

Moving on from your computer's processing power, you will have seen from the instruction book tht it also includes a minimum of 32K of **RAM** and 32K of **ROM**. But what do 32K, RAM and ROM actually mean?

In order for the processors in your MSX computer to function, they have to receive information from somewhere. They can't actually store it themselves, they can only process it. To allow the computer to store information, it has to be equipped with a memory, and this is where RAM and ROM come in.

Computers need to 'remember' two different types of information. The first type is the instructions which the CPU will go through to do work. These instructions are grouped together as programs to make the computer perform specific tasks, such as playing games or word processing. The second type of information the computer needs to remember is data. These are numbers or words used by programs – such as the score in a game, or the text of a word processing document.

Before the CPU can do any work, it must be told what to do (say multiply two numbers), and what to do it with (in this case, the two numbers we mentioned). The instructions and the data are both supplied to the CPU from the Random-Access Memory, or RAM. Whenever the CPU executes a program you have either written or bought for your computer, it goes to RAM to find the instructions and the data required by the program (this is called **reading** from RAM). If a program is supplied on a floppy disk or a cassette tape, it must first be placed in RAM before the CPU can use it.

When the computer performs a calculation, it places the answer back in RAM (called **writing** back to RAM). From there it may be displayed on the screen, or may be written onto a disk or tape.

There is one special kind of memory, which the CPU can read but not write into. This is called a Read-Only Memory, or **ROM**.

The difference between the two memories is that when the power to the computer is turned off, the data or the programs stored in the RAM are lost (as with pocket calculators). If you want to re-run a program after the power has been turned off, you will either have to reload it into RAM from the keyboard, or else retrieve it from disk or tape. ROM, on the other hand, will always retain its contents, even when the computer is switched off.

ROM's ability to do this is extremely useful. When the computer is switched on, for example, it immediately begins executing instructions stored in ROM. The first of these tell the computer to check its components. Because these instructions are stored in ROM, they do not need to be loaded each time the computer is switched on.

Another important use for ROM is to store the instructions which make the MSX-BASIC language operate. In this way, each time the computer is switched on and checks itself, it displays a welcoming message and 'OK' on the screen to show that MSX-BASIC is ready for use.

So much for RAM and ROM then, but what does 32K mean?

Bits and bytes

32K is a measure of the amount of memory that is present in the computer. In order to understand what this means in practice, we have to look briefly at how data is stored and used by the computer.

The computer transmits and stores information in the form of a code. This code is necessary because information can only be sent electrically, and it would be impractical for every piece of information that is transmitted, to be sent at a different voltage. The computer therefore only sends information using two voltages – 0 volts and around 5 volts. Because there are only two voltages, they have to be combined into groups to represent characters, such as the letters of the alphabet. These votages, or pulses, are termed **bits**. When we describe a bit, we say it is either a '1', to mean a 5V or 'on' signal, or a '0', to mean a 0V or 'off' signal.

For example, if three bits (either 5V or 0V signals) were sent at a time it would be possible to represent eight different characters as shown below.

Character Number	First Bit	Second Bit	Third Bit
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

You can probably see from this that if two computers agreed to send each other information in parcels each containing three pulses, they could communicate using eight different characters. In practice, being able to send only eight characters is pretty useless. In computer terminology, each group of bits is called a 'word'. The code above would be said to use a 3-bit word. It is important that you distinguish in your own mind that a computer word corresponds to a single character of our language.

By extending the code to four bits, the number of characters the computer can send is increased to 16. You can prove this for yourself by listing all the different codes it would be possible to send! One way of imagining how this increases the number of characters is to add a 0 in front of all the numbers listed above (giving you eight 4-bit numbers which all start with 0) and then replace this 0 with a 1 (giving you eight more 4-bit numbers, all of which start with a 1). Adding these together, gives you 16 different characters, each using four bits.

Because we need to send numbers and letters, punctuation marks, and odd characters like *, £ and >, the computer needs considerably more than four bit per character. In fact, computer designers and users have agreed to use 7 bits for a character. The agreed code, which virtually all microcomputers use, is called the American Standard Code for Information Interchange (ASCII). The full ASCII code is shown in Figure 1.

ASCII		ASCII		ASCII	
Code	**Character**	**Code**	**Character**	**Code**	**Character**
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	−	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	∧
009	HT	052	4	095	<
010	LF	053	5	096	′
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	>	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	″	077	M	120	x
035		078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	′	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal.
LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Rubout and are examples of the control codes numbers 1 to 32.

Figure 1 ASCII codes

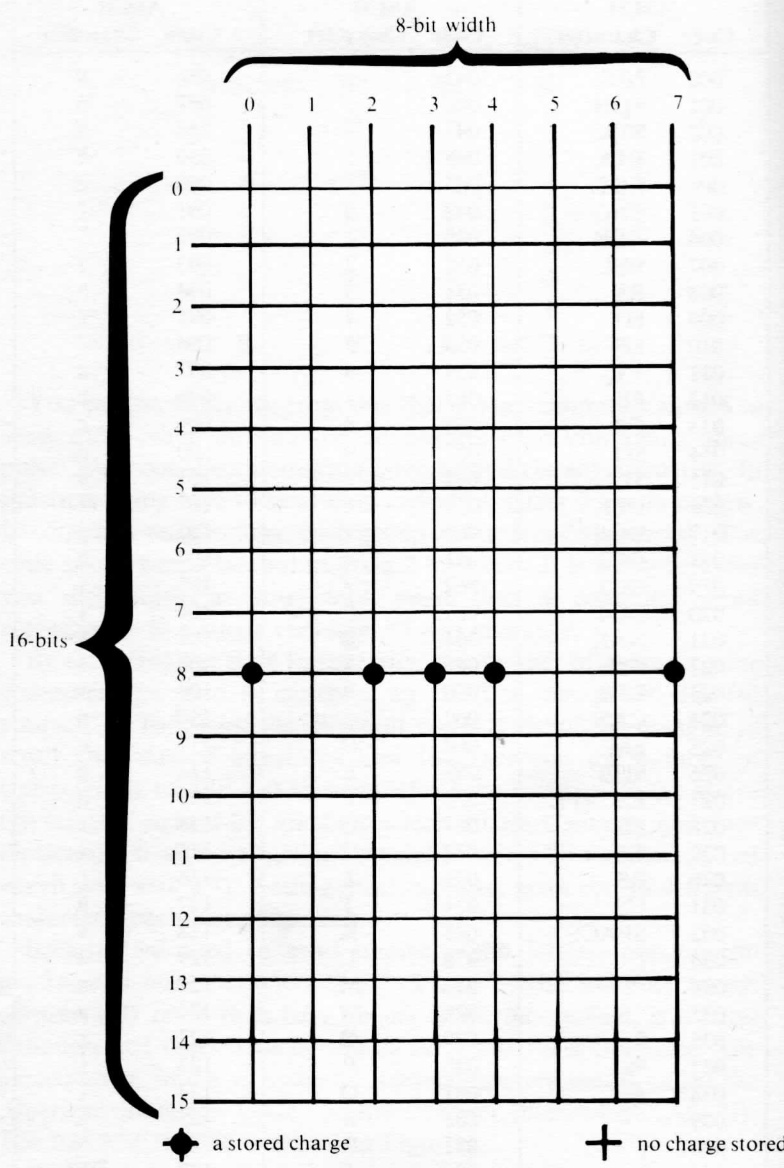


Figure 2 Diagrammatical representation of memory cells. 'On' bits, or 1s are represented by dark disks on intersecting wires. Bytes are referenced by their 'location on the grid. The binary word 10111001 is stored at memory location 8.

Although the ASCII code is a 7-bit code, information is sent in words of 8-bits, as it is possible to use the extra bit to make sure data is sent correctly.

As well as transmitting data, the computer also stores characters in 8-bit words. When referring to storage however, the word **byte** is used. The difference between a word and a byte is that a byte is always 8-bits, but a word can be of any length (as we demonstrated with our 3-bit words). In the case of an 8-bit word, a word is the same as a byte and the two can be used interchangeably.

Imagine the computer memory as a grid of wires. There are eight wires going down and 16 going across. At each point where wires cross, there is a means for storing a charge (like a small capacitor) (Figure 2).

In order to store the byte 10110011 (this is 51 in decimal, and you can look up what this character is in the ASCII table) the first, third, fourth, seventh and eighth intersections will be occupied.

As you will see from this diagram, the total capacity of our memory is 128 bits or 16 bytes. The memory is arranged as 8×16 bits.

By normal computer standards, this is an incredibly small amount of memory as it will store only sixteen letters (the word 'PROGRAMMABILITY.' for example, could be stored). A few years ago, 4096 bytes was considered a reasonable amount of memory for a small micro.

Rather than write 4096 bytes, the computer industry uses the letter 'K' to represent 1024 bytes, (as we use k to represent 1000 as in 1 km), and so 4096 would be written as 4K. The origins of the rather obscure number 1024 lie in the fact that 8-bit words are quite common in computers. If you keep adding one bit to the word length, you double the number of different words – just as you did earlier in moving from a 3-bit word to a 4-bit word. If you keep doubling 8, you will eventually get to 1024, and as every memory will be a multiple or a fraction of 1024, it makes sense to use this figure rather than 1000.

As we've said, a memory of 4K was common on older micros. Today though, the types of programs that are written often require more memory in order to be stored and run, and memory capacities of 8K and 16K are now common. On large business microcomputers, 64K and 128K are most common (note that 128K

is double 64K). They can often be extended up to 256K, 512K or even 1024K. At this point, the letter K is superseded by the letter M, standing for Megabytes, where 1 Megabyte is 1024 Kilobytes. Figure 3 summarises the nomenclature discussed above.

1 bit =	0V or 5V electrical signal
1 word =	a number of bits (the number depends on the computer but will usually be 8 or 16)
1 byte =	8 bits
1 Kilobyte (K) =	1024 bytes
1 Megabyte (M) =	1024 Kilobytes

Figure 3 Nomenclature summarised

Your MSX computer is based on a microprocessor that uses 8-bit words and will come with a ROM memory of 32K, and a RAM of between 32K and 64K.

One important point that must be considered is that the larger and more complex the program you wish to run on the computer, the more memory the computer needs to accommodate it. All MSX computers have the capability of adding more RAM and ROM, and this is something that you may have to bear in mind when you come to write large programs. Also worth considering is the fact that although your computer includes a minimum of 32K of RAM, 16K of this is used for storing information used by the screen, so that there is only 16K left (if you have 32K to start with) for you to use. Having said this, all the programs in this book are well inside this capacity – which should give you an idea of just how much you can do, even with a limited amount of memory.

ROM cartridges

Short though the life of the home computer industry is, the traditional way of getting large amounts of information into and out of the computer has been through the use of a cassette recorder. We'll take a closer look at this in a second, but it has been a method which has been dogged by problems. The ROM cartridge, by comparison, is simplicity itself.

When a ROM cartridge is plugged into the computer, it effectively replaces the built-in ROM. Instead of being given access to MSX-BASIC therefore, you'll be presented on the screen with a game, or home management program, or whatever else you've bought the ROM cartridge for. Not only does this provide a very simply way of getting programs into the computer, but it also means that the program size is not limited by the amount of RAM you have in the computer. There is no reason why you shouldn't be able to load a 64K ROM cartridge even if you've only got 32K of RAM in your computer.

The other major benefit is that you don't have to sit around for interminable lengths of time waiting for a program to load from cassette, only to find that something's gone wrong and you've got to start again.

The keyboard

To complete our look at hardware, we'll turn finally to the keyboard. There are many good reasons for including the keyboard in the section on peripherals, as you'll see when you read it. The reason that we've included the keyboard here is that it includes a feature found on many better home computers, – programmable function keys. As you can see when you switch on the computer, there are five keys which, when used in conjunction with the SHIFT key, can perform ten special functions. If you're interested in how you can change the purpose of the function keys, take a look at the KEY command in the next chapter on page 000.

Having reviewed what your computer can do and how it achieves its results, we'll now explain how it communicates with the outside world through its **peripherals**.

Peripherals

A peripheral is, quite simply, anything that you can attach to a computer. Given such a definition, the term covers an enormous range of devices, including those which allow you to get information into and out of the computer, store information, or perform specialist tasks. We shall look at these different types in order.

Input and output

The most obvious way of getting information into the computer is via the keyboard, so you can see our reluctance to include it under hardware. BEcause it allows you to put in information, it is called an **input** device. The MSX specification does not include a standard keyboard layout, however, all the MSX computers have the same number of function keys we described in the previous section which, although at a superficial level appear only to save you a bit of typing, can be put to a variety of uses, as we'll be demonstrating in later chapters.

Before we look at other input devices, let's look briefly at the standard *output* device – the TV screen. Almost all the home computers currently available use the TV as the primary means of displaying information since, like the cassette recorder, anyone who buys a home computer will almost certainly already own one.

Given the increasing pressure on the home TV set from sources such as videos, teletext, and so on, colour monitors can provide a cheap solution for users who find themselves banished from the lounge. Not only are monitors cheaper than a second TV, the screen and colour quality are also higher, thus making MSX graphics even more impressive than they otherwise would be.

For those of you who have little use for MSX's colour facilities, a simple black-and-white portable (or monitor) will provide the cheapest solution of all. Even without colour, the MSX graphics are still stunning (as we found whilst writing this book).

Having described, if somewhat briefly, the standard devices for input and output, the remainder are really just alternatives to be used for specific applications. Alternatives to the keyboard, for example, are joysticks (which give much faster response than the keyboard ever could for playing a computer game), light pens (whose most obvious application is in conjunction with a drawing package), and touch pads (best used when the computer displays a range of options on the screen which correspond to the keys on the touch pad. Touch pads essentially provide an extension to the function keys). By the time you read this book, there will probably also be a number of alternative devices that we don't even know about yet, each tailored to a specific function, and each with numerous adverts extolling its virtues in the popular computer magazines!

Printers

To turn to alternative output devices, the most obvious of these are printers, which allow you to keep a **hard copy** (a copy on paper) of your work. There are three major types of printer, described below.

Dot matrix printers are currently the cheapest form of printer you can get. Characters are formed by a matrix of dots which is typically seven dots vertically by five dots horizontally, although 9×7 matrices are also common. The dots on the paper result from pins being pressed either against a standard print ribbon, or by heated pins being pressed against heat sensitive paper. This allows them to print standard characters and, in some cases, a limited range of graphics characters.

Daisywheel printers, on the other hand, incorporate a more complicated mechanism and are comparable in their operation to a golfball typewriter. Each letter or symbol in the character-set sits on a 'petal' of the daisywheel. The daisywheel spins round at high speed and a hammer strikes the petal for a particular character, pressing it against a ribbon onto the paper. Because daisywheels are made from plastic or metal, a very high print quality (letter quality) can be obtained. Also, daisywheels can be changed quickly and easily, allowing different founts, languages, and so on to be printed. Daisywheel printers are generally less flexible than dot matrix printers; they cannot, for example, print out graphics and do screen dumps. They are also slower (due to the way they work) and more expensive. Their main advantage over dot matrix printers is that they produce a much higher quality of print, although they do tend to be much slower.

Finally, **plotters** are available to produce hard copy output of any graphics generated on the system. These work by using a pen, or different coloured pens in the case of colour plotters, to draw out precisely the image that appears on the screen. Plotters are much more complex machines than printers, and cannot be driven by your MSX micro in its raw state. They require a special piece of software called a **device driver** which translates wht appears on the screen into commands to make the plotter plot. Device drivers require an **operating system (MSX-DOS)**, and **floppy disks**, all of which we'll be talking about later. Suffice it to say that this makes them much more expensive than the other output devices, and really only applicable to users who want to be able to produce a lot

of complex graphics work, for example engineering drawings, printed circuit board designs, and so on.

Having said all this, the MSX standard is certain to produce a whole new range of MSX-compatible printers and plotters which, hopefully, will send prices tumbling and bring them within the reach of most users.

Storage devices

Moving on from the peripherals that allow you to get information into and out of your computer, we come to those that allow you to store information. We've already looked at the computer's ROM and RAM. The problem with RAM, as we said, is that when you've spent hours typing in a program and you want to stop and switch the machine off, you lose the program. What is needed is a memory, or storage device, which can store data even when the power is disconnected.

There are essentially two types of storage device that can be used with home micros – cassette recorders and floppy disks. Each of these has its advantages and disadvantages. We've already looked at some of the problems that can be encountered when cassettes are used for data storage, but they do provide a very cheap, if slow method of storing information. Floppy disks, on the other hand, can store much larger quantities of information (from 100K up to 400K or 800K on a single disk). They're fast, but they are also relatively expensive. Having said that, if you're going to use the computer for serious business purposes rather than just as a hobby, floppy disks are essential.

Robots and speech synthesis

The final sort of peripheral we have to look at, and which we can expect to see in ever expanding numbers, are those designed to perform specific functions. In their own way peripherals such as joysticks, light pens and touch pads fall into this category. In addition, there are devices such as voice synthsisers, speech recognition devices, robots, and virtually anything else which at present might sound like science fiction, but which will very soon become everyday reality. As we mentioned in the foreword to the book, your MSX computer is capable of talking to virtually any device you can think of. All that is needed is someone to provide

simple domestic devices with the capability to be interfaced (or connected) to an MSX computer.

On that cheerful note, and in order to prepare you for the next chapter, we'll now turn to the subject of computer software.

Software

We've now introduced you to the basics of the MSX computer hardware, and to the peripherals that you can add to the computer in order to get information into and out of it – but how do you get this mass of technology to do the things you want it to do? The answer is that you tell it wht to do through the use of programs or **software**.

As we said in the section on hardware, the language understood by the central processor is machine code. It's highly unlikely that you've ever written anything in this language as it's incredibly difficult to understand and takes a lot of time to learn. Obviously, it would be a lot easier if you could simply tell the computer in English what you want it to do. Unfortunately, English being what it is, there's too much ambiguity in the language to allow you to do this. The BASIC programming language, however, is about as close to programming in English as you're likely to get.

Microsoft BASIC is the industry-standard implementation of the language, both in terms of its sophistication and its popularity. It has currently been installed on well over 2 million microcomputers worldwide. MSX-BASIC has been developed from Microsoft BASIC, and incorporates many special commands to take full advantage of the MSX hardware. What MSX-BASIC essentially does, therefore, is to take instructions written in a language that people can understand and translate them into the language that the machine can understand.

We'll be looking in increasing detail over the coming chapters at wht the instructions in MSX-BASIC are and what they do. Any computer language is made up of **commands, functions** and **statements** and these can be used either on their own or in combination with each other, in which case they constitute a program.

Piecing all the MSX-BASIC instructions together to form a program requires an organised approach to problem solving. To assist in cultivating this approach, computer programmers use wht

is called a **flowchart** to break a problem down in to its constituent parts.

When a problem has been defined in this way, it is easy to translate the various parts into instructions that MSX-BASIC will understand. When all these are pieced together, they are a program. All it needs is a little imagination to see how the problem you want to solve can be turned into a flowchart.

MSX-BASIC provides the programmer with commands, functions and statements, and it's useful at this stage to look at each.

Commands

Commands tell the computer to do something like RUN or LIST. Commands work the moment they are typed in and don't need to be part of a program.

Statements

Statements are the numbered instructions that go to make up a program. LET X = 5 is an example of a statement. It sets X to the value 5.

Functions

MSX-BASIC offers a number of mathematical functions such as sine, cosine and tangent. Functions are always followed by a number, variable or expression in brackets. An example of a function is SIN (the sine function). PRINT SIN (PI/2) will give the sine of 90° in radians, as you'll possibly remember from your schooldays!

This essentially, is how the language works. We've covered a lot of jargon in this section, so it's perhaps appropriate to end by giving you an example of a 'typical' programming program.

Programming principles

Having said at the beginning that we were going to try to avoid problems with bank accounts and compound interest, these are issues that are close to everyone's heart, and also provide a convenient method of demonstrating the ideas we've been talking about. Have no fear though – the rest of the examples we present will be a lot more interesting.

Here's the problem. I have £100 invested in a building society account. Assuming that this gains 10% interest per year, and that the interest rate remains constant, how much will I have in my account in 10 years' time?

The first phase of analysing the problem is to break it down into smaller components.

The task is comprised of three main stages. Firstly, the computer needs to be told the initial value of the money deposited in the account. Then the first year's interest needs to be calculated. This must then be repeated ten times. Expanding these steps into a flow diagram, can end up with one similar to Figure 4.

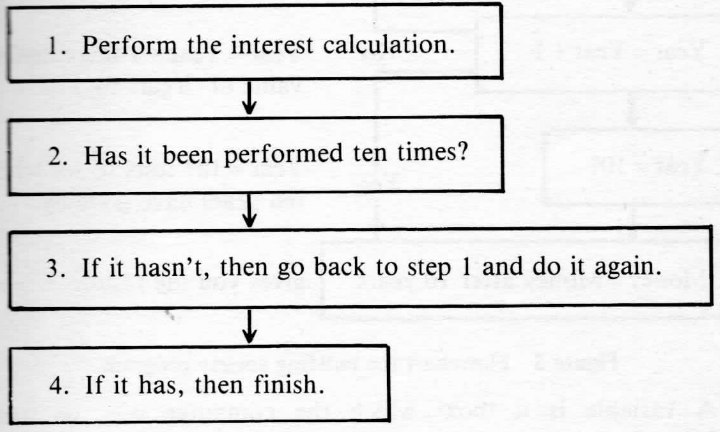


Figure 4 Simple flowchart

These steps are also shown in Figure 5 where we have also shown how BASIC **variables** are used, and how the computer might perform functions like 'repeat ten times'.

is called a **flowchart** to break a problem down in to its constituent parts.

When a problem has been defined in this way, it is easy to translate the various parts into instructions that MSX-BASIC will understand. When all these are pieced together, they are a program. All it needs is a little imagination to see how the problem you want to solve can be turned into a flowchart.

MSX-BASIC provides the programmer with commands, functions and statements, and it's useful at this stage to look at each.

Commands

Commands tell the computer to do something like RUN or LIST. Commands work the moment they are typed in and don't need to be part of a program.

Statements

Statements are the numbered instructions that go to make up a program. LET X = 5 is an example of a statement. It sets X to the value 5.

Functions

MSX-BASIC offers a number of mathematical functions such as sine, cosine and tangent. Functions are always followed by a number, variable or expression in brackets. An example of a function is SIN (the sine function). PRINT SIN (PI/2) will give the sine of 90° in radians, as you'll possibly remember from your schooldays!

This essentially, is how the language works. We've covered a lot of jargon in this section, so it's perhaps appropriate to end by giving you an example of a 'typical' programming program.

Programming principles

Having said at the beginning that we were going to try to avoid problems with bank accounts and compound interest, these are issues that are close to everyone's heart, and also provide a convenient method of demonstrating the ideas we've been talking about. Have no fear though – the rest of the examples we present will be a lot more interesting.

Here's the problem. I have £100 invested in a building society account. Assuming that this gains 10% interest per year, and that the interest rate remains constant, how much will I have in my account in 10 years' time?

The first phase of analysing the problem is to break it down into smaller components.

The task is comprised of three main stages. Firstly, the computer needs to be told the initial value of the money deposited in the account. Then the first year's interest needs to be calculated. This must then be repeated ten times. Expanding these steps into a flow diagram, can end up with one similar to Figure 4.

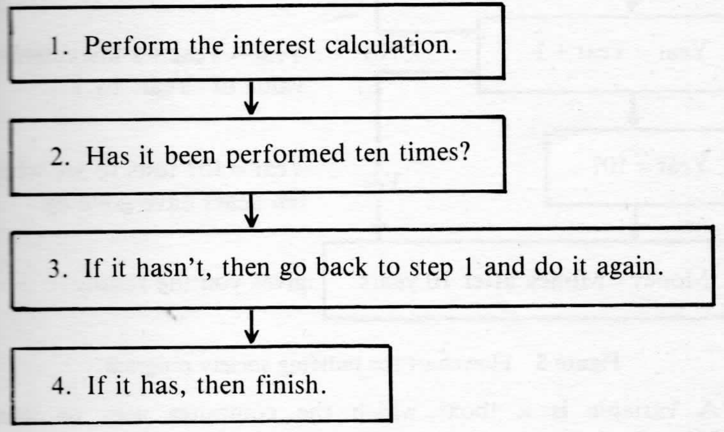


Figure 4 Simple flowchart

These steps are also shown in Figure 5 where we have also shown how BASIC **variables** are used, and how the computer might perform functions like 'repeat ten times'.

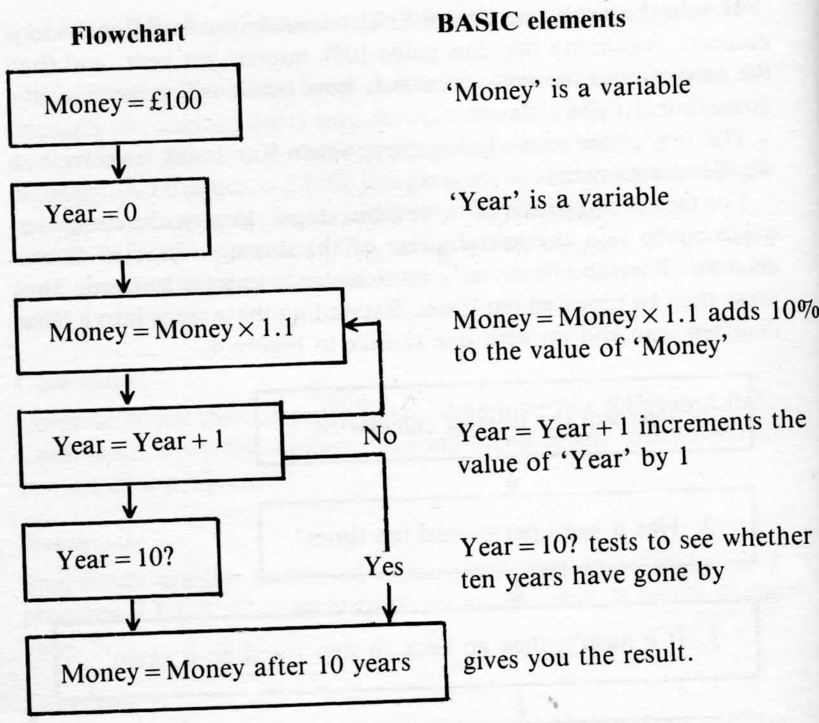


Figure 5 Flowchart for building society program

A variable is a 'box' which the computer uses to store information that is going to change. In the example above, we know that the amount of money in the account, and the number of years over which the investment is made, are going to have to change in order to find the solution to the problem. The variables 'Year', which will grow from 0 to 10, and 'Money', which will grow from £100 at a compound rate of 10% per year, are therefore declared at the start of the flowchart. The Year variable is then tested each time the loop is completed to see if it has reached the value of 10. If it hasn't, then the loop is performed again. When it does, the value for Money has grown to the amount in the account after 10 years, and the program, which essentially is what this is, stops.

If you can understand this, you'll have no trouble at all with the following chapters. What we're going to do is to introduce you to

the instructions that are available and what they do, and then show you how they can be connected to form increasingly complex programs.

Anyway, we hope that this introduction to computers in general, and MSX in particular, has proved useful. You've now come to the end of the preliminary part of the book, and the time has come for some audience participation, so plug in your computer and exercise those digits. It's time to try out some commands!

Programming in MSX-BASIC

Interpreted commands

When you turn on your MSX computer, the screen displays the prompt ''OK'' which tells you that it is ready to receive and process BASIC instructions. You can get it to do this in one of two ways. The first is to enter the instructions directly, in which case the computer **interprets** and acts on them immediately (this is known as **direct mode**). The second is to precede the instructions by line numbers, in which case they are treated as a program, and are executed in the order in which they appear in the program. A program is executed in **indirect mode**, and only begins when you type the command RUN directly. By far the simplest way of understanding what a command is and what it does is to type it in and see how the computer interprets it. As an example, try typing the following:

```
PLAY "04CDEFGAB05C"
```

and press the RETURN key. PLAY tells the computer that it is being asked to play something. ''04CDEFGAB05C'' tells the computer exactly what it has to play. As you can hear, the result of typing this is that the computer plays a complete scale. Later on, you'll see exactly how this is achieved.

Most instructions in MSX-BASIC, and any other BASIC for that matter, are composed of the name of the instruction and an expression defining how it is used. It is vital that when an instruction is used, the syntax of the notation that follows it is correct, which is another good reason for looking at direct-mode first. Having said this, some of the instructions that we'll introduce in this section have to be put into a program in order to have any meaning. Wherever this is the case, we've kept the programs as short as possible, while still demonstrating fully how the instruction can be used. Whenever you have to run a program to see how the instruction works, please do the following:

1 Type in the program exactly as it is printed in the book.
2 Either type RUN without a line number preceding it (i.e. in direct mode), or press function key 5 on your keyboard.
3 Most programs that you write for yourself will end with the MSX-BASIC "OK" prompt reappearing on the screen. In the programs that follow, you must press the CONTROL and STOP keys at the same time to stop the program and bring back the "OK" prompt. You can then type LIST to list the program if it is to be altered, or NEW to get rid of the program and allow you to enter a new one.

One of the features of MSX-BASIC is that you can **edit** statements on the screen. In the example above using PLAY, you could move the cursor up and change any of the letters. Pressing the ENTER or carriage return key will enter the modified line, just as if you had typed the whole line from scratch.

This applies to any line on the screen. If you typed in a line and it's still on the screen, you can go back, edit that line and press the ENTER key to save the line. If you forget to press the ENTER key, the line will remain as it was before you made changes.

This facility is very useful in MSX-BASIC programs, where you may have long and complicated lines which contain an error. By listing the line on the screen, you can correct it without retyping the whole line. In many of the example programs in this book, we stress the need to experiment for yourself with the programs. Once you have typed our programs in, you should use the edit facility to experiment with various alterations.

The full features of the MSX-BASIC Editor, as it is called, are listed in Appendix C. It is well worth familiarising yourself with how it works before proceeding.

If everything goes to plan, by the end of this section and the one which follows on graphics, you should have achieved two things. First and foremost, you will understand the most commonly used BASIC instructions – both what they do and how they do it. Secondly, from the simple programs we have included in this section, you will already have an idea of what a BASIC program is and how it works and will be well prepared for the last section of

the chapter on how to write more complex, and hence more useful, programs.

Incidentally, it might be approprite here to introduce the notation we'll be using to describe each of the instructions. As you'll see from the AUTO command (below), the syntax includes the [ ] and <> symbols, and later statements also include ( ). These have the following meanings:

<> Anything which appears inside these brackets must be used with the instruction, otherwise the computer won't understand it.

[ ] Things in square brackets are optional – it's up to you whether or not you include them. If you don't, the computer will simply assume that you're happy with the default condition (for example with AUTO, if you don't specify the start line or increment, numbering will start at line 10, with an increment of 10).

( ) These are the only brackets you have to type in as they are actually part of the syntax. You'll see exactly how they are used when you get on to an instruction which uses them.

To begin with, then, we'll have a look at the commands in detail. Good luck!

AUTO [<start line>] [,<increment>]

All the lines in a program must be numbered. The computer then carries out the instructions contained in the lines in numerical order. The AUTO command automatically numbers lines as they are entered into the computer, thereby saving you from an unnecessarily time-consuming task. AUTO is used only when a program is to be typed in. As examples, AUTO 100,10 would number lines in increments of 10 starting with line 100; AUTO 100,20 would give a 20 line increment, and so on. If you use AUTO with a partly written program, then if AUTO generates a line number which already exists, an asterisk will appear at the end of the line to warn you that this new line will replace the existing one. To turn AUTO off, press the CONTROL and STOP keys at the same time.

BEEP

Type in BEEP and a beep sound is generated. This is the most basic of the sound generating commands. For some more exhilarating ones, see the description of the SOUND command itself, or have a look at the music chapter.

CLEAR [**<string space>**] [,**<highest location>**]

The CLEAR command will clear the computer's memory of all variables, without erasing the program currently stored in RAM. After a program has run, the variables it used remain assigned in RAM. To put it simply, what CLEAR does is to clear any rubbish out of memory that you don't particularly want any more.

CLS

CLS quite simply clears the screen of anything that was previously on it.

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

The COLOR command (with apologies for the Americanism) is used to set the screen colours. As can be seen from the syntax, it is quite flexible, permitting control over the foreground, background and border colours through a single command line. The COLOR command is used by the computer when it is first switched on to set the colours that you see on the screen. The computer has a **default** value for COLOR of 15,5,4, with the numbers corresponding to white, dark blue and light blue respectively. The default is what an instruction does if no other comments or text are added.

NB: It is important to note here that according to the characteristics of your television set, you will not necessarily be able to see the border colour all of the time. It will appear however when you run a program, (always assuming you specify it, of course). We have attempted to avoid the use of programs in this section for the sake of simplicity. In this case, however, a short program is unavoidable just to prove that all three screen areas do exist, so try typing:

```
10 SCREEN 2
20 COLOR 14,2,9:CLS
30 CIRCLE (125,100),50
40 GOTO 40
```

What should appear is a grey circle on a green background, surrounded by a pink border.

You can use the COLOR to set the screen colours you find easiest to look at. For example, typing COLOR 12,15 will produce green characters on a white background, and it may be worth experimenting to find the combination best suited to you. For reference, the various colours and their numbers are as follows:

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

CONT

The CONTinue command is used in a BASIC program, or directly, to instruct the computer to continue program execution after it has been halted; for example, after the program has executed a STOP or BREAK command.

DELETE [<start line number>] [–<finish line number>]

As its name suggests, the DELETE command removes unwanted

lines from a program. All that must be specified are the first and last lines to be deleted. For example, DELETE 20–40 would delete all the lines from 20 to 40 inclusive from a program, DELETE 20 would delete line 20 only, and so on. A simpler way of deleting single lines is simply to type in the line number followed by ENTER. The computer will then store a blank line rather than what was previously there.

KEY <key number>,"<command>"

The MSX computers include 5 programmable function keys which, in conjunction with the SHIFT key, permit 10 pre-programmed functions to be stored. To obtain a list of the default functions type KEY LIST. Any of the pre-programmed functions can be changed using the KEY command. As an example type:

KEY 3, "COLOR 12,15"

If function key 3 is now pressed, the screen will change to display green text on a white background. Any function key can be programmed to produce any programming instruction, thus providing a considerable saving in time and effort when typing often-used commands. Key labels are displayed on the screen (as you can see!), and this display can be turned off using KEY OFF, or back on again using KEY ON. To make a command instantly executable it is normally necessary for it to incorporate a carriage return (or ENTER). This can be achieved by adding "CHR$(13)" to the key designation so that, for example:

KEY 1, "TRON" + CHR$(13)

will switch on the trace command (TRON) by pressing function key 1 only.

LIST [<line number>] – [<line number>]

The LIST command can be used in a number of ways to list a program. When typed on its own, LIST will display on the screen the entire contents of the program currently stored in the computer's memory. Alternatively, line numbers can be added to the LIST command, for example:

LIST 20–40	would list line numbers 20–40 inclusive of the current program
LIST 20	lists line 20 only
LIST 20	lists all lines from line 20 onwards
LIST 40	lists all lines up to and including line 40

Listing can be suspended by pressing the STOP key. Pressing STOP again will resume the listing. Pressing the CONTROL and STOP keys at the same time terminates the listing.

LLIST [**<line number>**] – [**<line number>**]

LLIST works in exactly the same way as LIST, the only difference between the two commands being that rather than displaying the program on the screen, LLIST sends it to a printer – always assuming that there is one connected to the computer!

MAXFILES = <expression>

As we've mentioned, whenever a BASIC program is run, the computer sets up a number of files in its memory to handle the work it has to do at any one time. The MAXFILES command allows you to limit the number of files that the computer can open during program execution, for example MAXFILES = 10 would allow the computer to open 10 files. Any number of files in the range 0 to 15 can be set. When MAXFILES = 0 is used, only SAVE and LOAD functions can be performed.

MOTOR [**ON**] [**OFF**]

The MOTOR command is used when you use a dedicated cassette recorder with your MSX computer, and is used to turn the recorder's motor on and off. This can be very useful, for example, when a long program has been written, only parts of which are desired to be stored to tape. MOTOR ON/OFF commands can then be inserted into the program at the appropriate points.

NEW

Whenever you have finished working with a program, there are

two ways that you can get rid of it. The first is to switch the computer off and then on again which is, to say the least, crude. The other is to type NEW, which will immediately delete the current program and any files or variables associated with it from the computer, thereby enabling you to load in or type a new program from scratch.

RENUM [**<new number>**] [,**<old number>**] [,**<increment>**]

The RENUMber command is used to renumber lines in a BASIC program, for example when the program has become overcrowded through the addition of new lines. The command can be used in a number of ways. By itself, RENUM will number all lines in the program in multiples of 10, starting (naturally enough) with line 10. For example, a program with line numbers 10,13,15,20,30 would be renumbered 10,20,30,40,50. Using the new and old number parts of the syntax, this can be changed. For example RENUM 15,10 used on a program with lines 10,20,30,40,50 would produce the result 15,25,35,45,55 – line 10 is renumbered to 15 and all lines following line 15 are incremented by 10. If the computer encounters a line number that already exists, a new number is created. Finally, the line increment can be changed so that for example RENUM,,20 would change program lines 10,20,30 to 10,30,50. Whenever line numbers are included in a command line, for example in a GOTO statement, then these will also be renumbered, so that the structure of the program remains unchanged.

RUN [**<line number>**]

The RUN command is used to instruct the computer to execute a program once it has been typed in. The use of RUN by itself will cause the computer to execute the whole of the current program – that is, all the lines that have been typed in. If a line number is specified, the computer will execute the program from that point onwards. For example, with a program containing lines numbered 10,20,30,40,50, RUN would cause all lines to be executed, whilst RUN 30 would result in lines 30,40 and 50 only being executed.

SOUND

The SOUND command can be used to make the sound chip in your computer do truly wondrous things. You'll be amazed at the huge variety of sounds that your computer is capable of making, shown in the music chapter, but for now, you may be interested in running the following program:

```
10 SOUND 8,5
20 FOR I=1 TO 255 STEP 2
30 SOUND 0,I
40 NEXT I
50 GOTO 20
```

(This could come in very handy if your house ever catches fire!)

TRON

It is unusual, to say the least, for anyone to get the program they are writing to perform exactly how they want the first time it is typed in. Indeed, with anything but the smallest programs, a considerable amount of time has to be spent in 'debugging' – that is, ironing out the bugs (or errors) in the program. To help with this procedure, MSX-BASIC includes the TRACE command, which is activated by typing TRON (TRACE ON). What TRACE does is to follow the progrm by printing on the screen the program line numbers as they are executed. This allows you to see the way the program is working as it is running. TROFF is the opposite of TRON, in that it turns TRACE off again. These commands would rarely be used in a program. Rather, they would be entered in direct mode before and after the program is run.

TROFF

See above.

WIDTH <width of screen in text mode>

The WIDTH command sets the line length, in characters, that can be displayed on the screen in text mode. When the 40×24 Screen 0 has been selected, the value given for the screen width can be any value up to and including 40. In 32×24 Screen mode 1, the

screen width can be any number up to and including 32.

The WIDTH command concludes our summary of the basic BASIC commands. WE shall now look briefly at the special graphics commands included in MSX-BASIC (a detailed description is included in the chapter devoted to the subject), before moving on to look at how these commands can be combined together into programs which do useful things.

Graphics commands

Up until now, we've only shown you part of your computer's capabilities – its ability to display 23 lines of text, each of 40 characters, along with the function key line.

In computer terms, we say this screen has a **resolution** of 920 (23 × 40). We could make up a picture using ordinary ASCII characters. This could not depict anything in great detail, but could draw simple pictures and bar charts, for example.

Certain characters, called **graphics characters**, are incorporated into MSX-BASIC which allow you to draw more elaborate pictures than you could using just letters of the alphabet or punctuation marks. Your MSX microcomputer may or may not have these characters shown on the keyboard, but a good way of seeing what characters are available is to run the program below. What this program does is to print out the entire character set on the screen:

```
10 FOR I=1 TO 255
20 PRINT I,CHR$(I)
30 A$=INKEY$:IF A$="" THEN 30
40 NEXT I
```

The program prints a character symbol alongside a number. Press any key to get another number and character printed on the screen. Look at each number and note its corresponding character. If you want to incorporate any character into a program, 'PRINT CHR$(170)' (or whatever the number is) will print the character on the screen. **Character graphics** use what MSX-BASIC calls **Screen 0**.

In order to draw more complicated pictures, we need a higher resolution screen, where we can control elements which are smaller than whole characters. This would allow us to draw pictures which

contained a lot more detail than pictures made up of ASCII characters.

As well as Screen 0, MSX-BASIC gives us access to three other screens – numbers 1, 2 and 3. Screen 1 is used as an alternative to the text screen and Screens 2 and 3 for high and low resolution graphics.

In order to use the graphics commands, you must first instruct the computer that you wish to use one of the two graphics screens. If you try to use a graphics command with the standard character screen, the computer will produce an error message. In order to see the differences between the two screens, therefore, try typing in the following program:

```
10 SCREEN 2
20 CIRCLE (125,100),50
30 GOTO 30
```

What you ought to see on the screen is a reasonable approximation of a white circle on a dark blue background, surrounded by a light blue border. Now run the program again, but this time replace SCREEN 2 with SCREEN 3. This time, not only is the circle drawn with a much thicker line than it was before, but it also looks a lot less like a circle than did the previous one. The reason for this difference is the screen resolution.

Screen 2 is rather like a piece of graph paper with 256 squares horizontally and 192 vertically (thus giving it a resolution of 49152 squares). Because a television screen is fairly compact, the squares, or **pixels**, are very small. It is possible to **address**, or change the colour of, any pixel on the screen, and it is this facility which allows you to draw circles, lines, and all the other shapes we'll be talking about later on. You can change the colour of a given pixel using the PSET command. All you have to specify is the pixel on the screen that you want to change, using its x,y coordinate (the point 100 pixels from the left of the screen and 50 pixels down it has coordinate 100,50). To demonstrate how this works, type in the following program:

```
10 SCREEN 2
20 PSET(120,20)
30 PSET(121,30)
```

```
40 PSET(122,40)
50 PSET(123,50)
60 PSET(124,60)
70 PSET(125,70)
80 PSET(126,80)
90 PSET(127,90)
100 PSET(128,100)
110 GOTO 110
```

As you can see, the program has changed the colour of a series of pixels, starting at the point 120,20 and ending at 128,100. Now replace SCREEN 2 with SCREEN 3, run the program again and see what happens.

The most obvious difference, as you saw when the computer drew a circle, is that pixels are drawn rather differently on Screen 3 than on Screen 2. Rather than changing the colour of a single pixel, PSET causes a 4×4 block of pixels to change colour. As you can see, a horizontal coordinate of 120, 121, 122 or 123 makes no difference to the horizontal position at which the block is drawn. When 124 is reached, the block is plotted at the next horizontal position, and again, this does not change until 128.

Like Screen 2, Screen 3 has 256×192 addressable pixels, but it can only plot to 64×48 4-pixel blocks (thus giving it a resolution of 3072 blocks). It is important to bear this in mind when using Screen 3, otherwise it's easy to end up with some very confusing results.

CIRCLE (<x coordinate>,<y coordinate>),<radius>[,<colour>] [,<start angle>] [,<end angle>] [,<aspect ratio>]

The CIRCLE command has by far the most complex syntax of any we've come across so far, so we'll take a little time to explain exactly wht all of its elements do. As you can see from the above program, we quite happily drew a circle using only the first three elements of the syntax. The first two of these define where the centre of the circle is to be (how far across the screen, and how far down). The third element defines the radius of the circle. If you simply change the values in the above program, you'll soon see how the circle can be moved around the screen and changed in size. Moving on to the fourth element of the syntax, the colour that the circle is drawn in can also be changed, so that, for example:

CIRCLE (125,100),50,10

will produce a yellow circle rather than a white one (see the COLOR command for a full list of available colours). Should you wish to draw only part of a circle, then the next two parts of the syntax are definitely for you. Both elements can be defined in the range −2xPI to +2xPI – PI is the ratio of the circumference of a circle to its radius (as we all learnt at school!) and is roughly equal to 3.142, hence the range of values accepted by the computer is −6.284 to +6.284. Rather than trying to explain in detail what the effects of varying these two elements of the syntax are, we suggest you play about with them yourself to see exactly what happens. When you think you've seen enough, try running this program, which is derived from the one we first used to introduce CIRCLE:

```
10 SCREEN 2
20 LET X=0:LET Y=-.1
30 CIRCLE(125,100),90,7,Y,X
40 LET X=X+.1:LET Y=Y-.1
50 IF X<6.2 THEN GOTO 30
60 GOTO 60
```

Now run the program again but change X=0 in line 20 to X=0.1.

The reason that we've included the program is to demonstrate how quickly the flexibility and usefulness of a command can be built up simply by specifying further elements of its syntax or adding a couple of extra command lines. We do not expect you to be able to understand the program at this point – more important is to see just how flexible CIRCLE actually is. Finally, let's look at the aspect ratio. This is the ratio of the verticl height of the circle to its horizontal width, and allows you to draw a variety of ellipses. Again, no further explanation is really necessary here – it's much better for you to try changing the numbers around for yourself. Having said that, it may be of interest to add a further X to the above program making line 30 into:

30 CIRCLE (125,100),90,7,Y,X,X

Silly, isn't it?

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

See previous section, 'Interpreted Commands', page 23.

DRAW <string expression>

DRAW, like PLAY (which you'll meet later) uses a powerful macro language, the Graphics Macro Language. DRAW allows you to draw lines on the screen by colouring in points on the screen (pixels). The Graphics Macro Language uses a shorthand for directing which pixels are coloured in – U for up, D for down, L for left and R for right. For example, you can draw a box using a command that colours in 100 pixels to the right, then moves down 100 pixels, then left 100 pixels and then up again 100 pixels. The program to do this is as follows:

```
10 SCREEN 2
20 DRAW "R100D100L100U100"
30 GOTO 30
```

This is pretty well the simplest thing that you can do with the Graphics Macro Language. You can see exactly how flexible the language is by looking at the full description of it given in Chapter 6.

LINE (<x1,y1>)–(<x2,y2>[,<colour>][,<B/BF]

The LINE command draws a line between the two points specified. As an example, try typing:

```
10 SCREEN 2
20 LINE(20,10)-(240,180)
30 GOTO 30
```

As you can see, you get a straight line between the points 20,10 and 240,180. The line can be changed in colour by adding a number in the range 0 to 15, for example changing line 20 to:

20 LINE (20,10) – (240,180),10

will produce a yellow line.

LINE can also be used to draw a rectangle which has the line as

its axis (add,B to line 20), which can also optionally be filled with colour (change the B in line 20 to BF), hence:

```
20 LINE (20,10) – (240,180),10,BF
```

will change the line into a yellow rectangle.

LOCATE <x,y>

The locate command can be used with both the graphics screens and the character screens to locate the position at which drawing or printing will commence. As an example, type in the following program:

```
10 SCREEN 0
20 LOCATE 20,10
30 PRINT "Hello"
```

When you run the program the computer prints the word "Hello" on line 10, starting at position 20. The same will happen with Screen 1. To show what happens with the graphics screens, a couple of lines have to be added to the program, so that it looks like:

```
10 OPEN "GRP:" FOR OUTPUT AS #1
20 SCREEN 2
30 LOCATE 20,10
40 PRINT#1,"HELLO"
50 GOTO 50
```

(The extra lines are needed to allow you to print on the graphics screens.) Now run the program again. Again, the computer prints "Hello" starting at position 20,10 which since the screen cn define 256 × 192 positions rather than the 40 × 23 positions of Screen 0 shifts "Hello" upwards and to the left. Note also that the size of the characters has changed since the high resolution grphics screen defines its characters in a different way to Screen 0. As a further demonstration of this, change SCREEN 2 in line 20 to SCREEN 3. It's a bit different, isn't it?

PAINT (<x,y>) [,<paint colour>] [,<border colour>]

The PAINT command is used to fill an arbitrary graphics figure

with the specified fill colour. We shall look more fully at how the command is used in the following chapter, but essentially what PAINT does is to fill whatever shape you have drawn on the screen with the colour you choose, starting at a specified point within that shape. To demonstrate the command, we've been forced to resort to a slightly longer program than usual, but the results are quite colourful and it is worth the typing effort!

```
10 SCREEN 2
20 LET C=2
30 FOR I=100 TO 1 STEP -10
40 CIRCLE (125,100),I,C
50 CIRCLE (125,100),C
60 LET C=C+1
70 NEXT I
80 GOTO 80
```

As you can see, what the program is doing is drawing decreasing circles and filling each one with colour starting at the centre. You may be able to see from the program why this is happening, but once again, if you can't, don't worry about it – by the end of the next section you will!

POINT (<x,y>)

The POINT command allows you to determine the colour of a particular pixel on the screen. POINT is only really necessary for advanced graphics programming, so we won't confuse you by attempting to describe it in any greater detail here.

PSET (<x,y>)[,<colour>]

Whilst the PAINT command allows you to colour in a complete block or shape, the PSET command allows you to specify the colour of an individual pixel. For example, if you run the following program:

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

then you can see that the pixel at position 100,000 has been changed to the foreground colour. If you change line 20 to read

```
20 PSET (100,100),10
```

then the dot will change to yellow. In common with the other graphics commands, changing Screen 2 to Screen 3 will produce a yellow box, and so on.

PRESET (<x,y>),<array name>[,<option>]

PRESET is the exact opposite of PSET in that if the colour is not specified then PRESET will draw the pixel or pixels in the background colour, rather than in the foreground colour as is the case with PSET. If the colour is specified, then PSET and PRESET both work in exactly the same way.

PUT SPRITE <sprite plane>[,<,x,y>][,<colour>] [,<n>]

A sprite is a graphics character which can be moved quickly across the screen. Sprites will be familiar to anyone who has ever played Space Invaders, or any other video game for that matter, since it is through the use of sprites that the invaders, and most other things that move across the screen, are drawn.

You are used by now to the fact that the graphic screen is made up of 256 × 192 pixels. Sprites are defined in the program, and can be any shape or size, made from one to 32 × 32 pixels in size. You can have up to 256 of them in a program, of which up to 32 can be on the screen at once, on up to 32 separate planes – enough for even the most enthusiastic game designer!

The shape of the sprite is defined using the SPRITES variable, and the sprite's attributes are defined using the PUT SPRITE command. The syntax of this command gives the plane on the screen on which the sprite is to be drawn, which can be in the range 0 to 31; where the sprite is to appear (x coordinates can be in the range − 32 to + 255, y from − 32 to + 191, for general use); and a pattern number for the sprite (this must be less than 256 when the sprite is made up of 8 × 8 pixels, and less than 64 when it is made up of 32 × 32 pixels). For example:

```
PUT SPRITE 0,(100,100),7,17
```

would mean put a cyan coloured sprite of pattern number 17 on plane zero at position 100,100. (This assumes, of course, that sprite 17 has previously been defined.)

NB: The PUT SPRITE command makes use of the following: If the x and y coordinates are specified as STEP (x,y) rather than simply as x and y on their own, then the sprite will move the x and y distances relative to where it last appeared, ie STEP (100,50) means put the sprite 100 pixels to the right of and 50 pixels down from where it last was, rather than putting it at position 100,50. If the y coordinate is specified as 208, then all planes behind the one on which the sprite appears will disappear until y is changed to something other than 208. If the y coordinate is specified as 209, then the sprite will disappear from the screen.

SCREEN [<**mode**>][,<**sprite size**>][,<**key click**>] [,<**cassette baud rate**>][,<**printer option**>]

The SCREEN command is one with which you should now be familiar, to say the least! It does however include a number of features which we haven't had cause to touch on so far. As we're sure you're only too well aware, the mode option is used to define the screen type that you want to use, with the alternatives being as follows:

0 40×24 text mode
1 32×24 text mode
2 256×192 high resolution graphics mode
3 multicolour low resolution graphics mode

Sprite size can be selected as follows:

0 8×8 unmagnified
1 8×8 magnified
2 16×16 unmagnified
3 16×16 magnified

If you don't like the keys on your MSX computer clicking, then switch the key click option to 0. 1 will switch the key click back on again.

The last two elements of the syntax should help out with any problems you might have with a cassette recorder or printer, if you

have one connected. The first allows you to set the baud rate at which programs are sent to the cassette recorder, and can be either 1 for 1200 baud or 2 for 2400 baud. The final element is used if your printer does not conform to the MSX standard, and will be non-zero if this is the case.

SPRITE$ (<n>)=<string expression>

SPRITE$ isn't actually a command, it's a variable. (See Chapter 3 on variables and constants)

We'll be looking at how to define sprites and write programs which make full use of them in the next chapter. All you need know at the moment is that to define one, all you have to do is to give the sprite a name and then sy what you want it to look like. There are several ways of doing this which will be detailed later. For now, if you're interested in the sorts of things you can do with sprites, take a little time to type in the following program:

```
10 SCREEN 2,0,0
20 FOR I=1 TO 8
30 READ B$
40 S$=S$+CHR$(VAL("&B$"+B$))
50 NEXT I
60 SPRITE$(0)=S$
70 X%=INT(RND(1)*256):Y%=INT(RND(1)*192)
80 PUT SPRITE 0,(X%,Y%),15,0:BEEP
90 FOR J=1 TO 300:NEXT J
100 GOTO 70
110 DATA 00011000
120 DATA 00111100
130 DATA 01100110
140 DATA 11011011
150 DATA 11011011
160 DATA 01100110
170 DATA 00111100
180 DATA 00011000
```

Notice how the shape of the sprite is defined by the 1s in the data statements.

VPEEK (<video RAM address>)

The VPEEK command allows you to see exactly what is in a particular part of the computer's video memory. Peeking, and poking (which follows) are the domain of the expert programmer and well beyond the capabilities of the novice. At this stage it's not worth saying any more than that, so we won't!

VPOKE (<video RAM address>),<byte>

VPOKE allows you to insert a byte of information directly into the computer's memory, the address being between 0 and 16383. Unless you know exactly what you're doing, we would strongly recommend you NOT to try out this command (unlike virtually any other command in the language!) If you do try poking and the computer crashes, you can't get any more sense out of it. Switching it off and back on again should solve the problem – we do advise against it though.

In this section we've attempted to demonstrate in detail the most useful, and hence the most widely-used commands available in MSX-BASIC, and we hope that you've found the experience of trying them out an interesting one. Impressive though the commands are in themselves, they don't realise their full potential until they're combined together into programs. We hope that this is obvious from the programs that we've included for explanatory purposes. Although our intention, as we stated at the beginning of the chapter, was to use programs as little as possible, where we have had to use them we hope it's been apparent that the addition of a couple of extra lines to a program can dramatically change the image that appears on the screen.

Well, it's time to take that step from simply using the commands as they're provided, and imposing your own will on the computer, through the art of computer programming!

Starting to program

After looking at so many of the commands and statements available in MSX-BASIC, it's time to look at how we can put them into programs to do useful things. As a break with the tradition

that seems to be a vital part of every book on BASIC programming, we're not going to be asking you to write programs to work out the interest on your bank account (we'll let you off with the example in the introduction), make you guess random numbers for no reason, and so on. Instead we decided to make as full use as possible of the excellent graphics capabilities of the MSX computers, to show how programs can be built up to make the images you see increasingly complex, and so introduce structured programming in a rather more interesting way than is usually the case. We'll then go on to see how the ideas that you learn in this section can be put to more practical use.

Loading and saving programs

But first, some more mundane matters – you've got a few more commands to lern. We've saved these until now as they're needed to get progrms into and out of the computer. As your programs get longer and longer it's frustrating to have to type them in every single time you want to use them, or to have to write them out every time they've been run successfully. If you've got a cassette recorder or disk drive attached to your computer, then the following commands get you round this problem:

SAVE "<device descriptor>.[<file name>]"

The SAVE command saves the BASIC program currently held in memory to the device specified by the device descriptor, e.g. CAS or DISC. For example, SAVE "CAS:JON" would save a file to cassette, and name it JON.

LOAD <"device descriptor:[<filename>]">[,R]

This command is used to load a BASIC program from cassette. LOAD deletes the current program from memory. If the R option is specified then the loaded program starts to run automatically. If the program name is omitted, then the next program which is encountered on the cassette is loaded.

MERGE "[device descriptor:<filename>]"

The MERGE command allows programs to be merged together.

It is important that the programs do not contain conflicting line numbers (for example, two programs both with line number 20), since only the line which is being merged into memory will be saved. The command works by merging a BASIC program on cassette (cassette is the default means of storing programs) into the program currently held in memory, for example MERGE "CAS:JON" would merge a program called JON, stored on cassette, into the one currently being held on the computer. If the program name is omitted, the next program encountered is merged.

BSAVE "<device descriptor>:[<filename>]" [,start address] [,end address] [,execution address]

As you become more proficient in programming you may move away from BASIC programming and on to more complex machine level programming.

BSAVE is used to save a machine code program. It differs from SAVE (which is used to save BASIC programs) in that it can write anything stored in memory (including data), onto cassette. The B in BSAVE stands for "binary", since it is binary data read directly from RAM that is being saved. Its use, though, is really outside the scope of this book.

BLOAD "<device descriptor>:[<filename>]"[,R] [,<offset>]

The BLOAD command loads a machine language program directly into memory and again it is really outside the scope of this book. For completeness, however, [,**R**] causes the program to run as soon as it has been loaded.

Constructing programs

Having sorted out how to store and retrieve programs, it's now time to look at how the programs themselves are built up. We'll start off by looking at how you can structure the programs to make them more efficient and, coincidentally, save yourself a huge amount of typing!

FOR. . .NEXT

You're already familiar with short programs such as the following:

```
10 SCREEN 2
20 CIRCLE (125,100),100
30 GOTO 30
```

This draws a circle with a radius of 100 pixels centred on the point 125,100. But what if you want to draw a series of decreasing circles? Well, one way (although whatever you do don't type this one in!) would be to write the following program:

```
10 SCREEN 2
20 CIRCLE (125,100),100
30 CIRCLE (125,100),90
40 CIRCLE (125,100),80
50 CIRCLE (125,100),70
60 CIRCLE (125,100),60
70 CIRCLE (125,100),50
80 CIRCLE (125,100),40
90 CIRCLE (125,100),30
100 CIRCLE (125,100),20
110 CIRCLE (125,100),10
120 GOTO 120
```

As you can see, what this does is to ask the computer first to draw a circle 100 pixels in radius, then one with a radius of 90 pixels, then 80, and so onuntil line 110 where a circle 10 pixels in radius is drawn. The last line of the program includes the GOTO statement which you've seen in all the programs so far, and which we shall look at in greater detail later. In this particular case, GOTO quite simply tells the computer to keep going to line 120 rather than running off the end of the program. If the GOTO statement hadn't been specified, then the circles would disappear from the screen and the computer would return to the character screen as soon as the last circle had been drawn.

Now, typing in all of the above is an extremely tedious thing to do (it was worse for me to type in that it was for you to read!) and is unnecessary since all that you have to do to achieve the same result

is to put a FOR. . .NEXT loop into your program. We'll be talking about loops quite a lot in the following pages, and essentially what they do is to get the computer to perform the same action a number of times, either in exactly the same, or in a slightly different manner. To demonstrte a FOR. . .NEXT loop, type in the following:

```
10 SCREEN 2
20 FOR N=10 TO 100
30 CIRCLE (125,100),N
40 NEXT N
50 GOTO 50
```

What the computer has done has been to draw a series of circles, starting with a radius of 10 and ending with a radius of 100, centred on the point 125,100. How has it done this? Well, the first time it looks at line 20 the FOR statement tells it to assign the value of 10 to a variable known as N. Whenit executes line 30, therefore, it knows that it has to draw a circle with a radius of 10 pixels. Line 40 tells the computer to take the next value of N. It goes back to line 20 and, since N has been specified as being from 10 to 100, it adds 1 to the existing value, and in line 30 draws a circle with a radius of 11 pixels. This continues until N reaches 100, at which point the computer reads line 50, which tells it to stay where it is.

To return to our original problem, what we wanted was a series of circles, 10 pixels apart. This can be achieved by changing line 20 to read

```
20 FOR N=10 TO 100 STEP 10
```

By introducing STEP into the command, we have told the computer to increment N not by 1, but by a value which we have specified, in this case 10. Thus, the computer now draws circles with radii of 10, 20, 30, ... 100 pixels, the end result being what we wanted the original program to produce.

The way the circles are drawn, however, is still not right. In our original long-winded program, we asked the computer to draw circles with a radius starting at 100 pixels and ending up at 10. The current program does the exact opposite. We can solve the problem by changing line 20 again to the following:

20 FOR N = 100 TO 10 STEP − 10

The circles are now drawn using three program lines insted of the ten we originally looked like needing. If you now look back to the program we used to demonstrate the PAINT command, you should be able to see exactly how the circles get filled with their respective colours. The program we used then was as follows:

```
10 SCREEN 2
20 LET C=2
30 FOR N=100 TO 0 STEP -10
40 CIRCLE (125,100),N,C
50 CIRCLE (125,100),C
60 LET C=C+1
70 NEXT N
80 GOTO 80
```

The structure of the program is the same as the one that we ended up with above, the exception being that the PAINT command has been added in such a way that it also makes use of the FOR. . .NEXT loop. As you can see, the paint colour has been made variable (C). The colour is initially set to 2 (in line 20), and each time the FOR. . .NEXT loop is executed, the colour number is incremented by 1 (line 60). This is achieved using the LET statement. You will also notice that the colour in which the circle is drawn is the same as the colour in which it is about to be filled. Let's return to our circle drawing problem and add a couple more lines to it:

```
10 SCREEN 2
15 FOR M=60 TO 180 STEP 40
20 FOR N=100 TO 10 STEP -10
30 CIRCLE (M,100),N
40 NEXT N
45 NEXT M
50 GOTO 50
```

All we've done here is to add another variable into our CIRCLE and, using another FOR. . .NEXT loop, ask the computer to move the x coordinate for the centre of the circles 40 pixels to the right

each time it plots them. Note that the two loops must not cross each other – the FOR M. . .NEXT M loop encloses the FOR N. . .NEXT N loop. Whenever you write a program, this should always be the case – otherwise the computer will respond with an error message.

Let's add this second FOR. . .NEXT loop to our program for drawing circles and then colouring them in. We haven't asked you to do anything for yourselves so far, so try adding in the second loop yourself before looking at the next program.

```
10 SCREEN 2
20 LET C=2
30 FOR M=60 TO 180 STEP 40
40 FOR N=100 TO 10 STEP -10
50 CIRCLE (M,100),N,C
60 PAINT (M,100),C
70 LET C=C+1
80 NEXT N
90 NEXT M
100 GOTO 100
```

You should have got a program that looks something like the above (we RENUMbered it on the way to stop things getting too crowded). If you did, then congratulations, you should be pleased with yourself. If you didn't, don't worry about it too much – it's just a matter of practice! Now let's run the program and see what happens – the result should be quite colourful after all!

What went wrong? As you can see, the computer got halfway through the second set of circles, stopped, and produced the error message:

Illegal function call in 50

Have a look at line 50 and see if you can sort out what happened.

The answer is that the computer ran out of colours with which to fill in the circles. Colour numbers are only available in the range 0 to 15. At the start of the program we asked the computer to start with colour 2. Fourteen circles later, therefore, it reached colour 15. When line 70 is reached, this is increased to 16, so that when the

computer reaches line 50 again, it is asked to draw a circle in a colour that it does not understand, hence the error message.

So how can we get the program to complete its pattern of circles?

IF. . .THEN

The reason that we got our error and our program "crashed" was that we allowed the colour number to get outside its permitted range, and therefore asked the computer to do something it didn't understand – draw a circle using colour 16, which doesn't exist. The answer, therefore, is to stop the colour number going beyond its limit by checking thenumber and telling the computer what to do when the limit is reached. Adding the following line to the program will achieve precisely this:

```
75 IF C=16 THEN C=2
```

This checks to see if C is 16 and if it is, resets it to 2. If it isn't 16, it continues to line 80.

If you run the program again with this line included, you'll see that this time everything goes to plan. The program runs as it did before, the difference being that this time when line 70 causes the value of C to reach 16, line 75 intercepts it and causes the value of C to return to 2, thereby allowing the process to start all over again. The same thing happens every time C reaches 16, until all the circles have been drawn and filled.

In the above example we used IF. . .THEN to get us out of a problem we'd got ourselves into. There are many uses that IF can be put to, however – it's really up to your imagination how and when you use it. You have two options in terms of what you can ask the computer to do IF something happens.

IF. . .THEN. . .ELSE

To return to our circle drawing program, all that we were interested in testing was whether C was equal to 16 and if it was, to change it back to the value 2. If the condition is "true" then the line is executed, if it is "false" (i.e. C is not equal to 16) then the computer moves on to the next line. With larger programs than this one, it may well be more convenient, if the condition is false, to move to a line other than the next one in the program, and this is

one way in which IF. . .THEN. . .ELSE can be used. For example:

IF C = 16 THEN C = 2 ELSE 80

is an alternative way of writing line 75. The nett result of the line being executed is the same, but insted of letting the computer execute line 80 simply because it is the next line in the program, you have instructed it exactly what to do if the condition is false.
N.B. Line numbers and programming instructions are equally valid things to write after THEN or ELSE, as demonstrated by the above change to line 75.

IF. . .GOTO

You have already seen the GOTO statement in all the programs we have asked you to type in. So far, all we have used GOTO for is to ensure that the result of running a program remains on the screen, by making the last line of the program loop back onto itself until you stop it using the CONTROL and STOP keys. What GOTO does, quite simply, is to cause the program to jump to the line number which follows it.

To demonstrate how GOTO works, type in the following program and run it. Before you read the explanation of what's going on, have a look at the program and see if you can work it out for yourself. It's the most complicated program we've asked you to type in, so don't worry if it's a bit difficult. The program is essentially the same as the circle drawing program; we've just added a few lines to get the computer to ask you if you want it to run or not. When you run the program, answer yes or no as it asks, but try replying with why? or any other such word as well, and see what happens then.

```
10 CLS
20 PRINT "Do you wnat me to draw some
circles?"
30 PRINT "Please answer yes or no"
40 INPUT A$
50 IF A$="yes" GOTO 90
60 IF A$="no" GOTO 170
70 PRINT "Sorry, I don't understand.
Try again"
```

```
80 GOTO 40
90 SCREEN 2
100 LET C=2
110 FOR N=100 TO 10 STEP -10
120 CIRCLE (125,100),N,C
130 CIRCLE (125,100),C
140 LET C=C+1
150 NEXT N
160 GOTO 160
170 END
```

Let's look at what the program does, line by line.

Line 10 clears the screen

Line 20 tells the computer to print on the screen the question "Do you want me to draw some circles?"

Line 30 asks you to respond by typing either "yes" or "no"

Line 40 makes the computer wait for you to give it a word. It assigns a variable called A$ into which your reply will be put and causes the computer to print a ? on the screen to let you know that it is expecting some sort of response.

Line 50 tells the computer that if your answer was "yes" then it must jump stright to line 90, ignoring lines 60, 70 and 80. It then runs through our familiar circle drawing program. If your response wasn't "yes" then the GOTO in line 50 isn't executed so the computer moves on to line 60.

Line 60 tells the computer that if you reply was "no" then it must go to line 170, where the END statement causes the program to cease execution and return the OK prompt to the screen.

Line 70 tells the computer to print on the screen "Sorry, I don't understand. Try again." just in case you didn't reply either "yes" or "no" in line 40.

Line 80 tells the computer to go to line 40 again, where it waits for you to type in an answer that it does understand.

We hope that from this explanation you see how flexible IF. . .GOTO statements can be, and how much typing they can save by allowing you to use the same part of the program over and over again, in much the same way as FOR. . .NEXT loops.

IF. . .GOTO. . .ELSE

IF. . .GOTO. . .ELSE statements work in exactly the same way as IF. . .THEN. . .ELSE, although GOTO restricts the action after IF to a jump to a different line.

Finally, it is perfectly permissible for you to have a command line combining a number of these conditional tests, such as:

IF. . .THEN IF. . .AND. . .GOTO. . .ELSE

or even

IF. . .OR IF. . .AND IF. . .THEN. . .ELSE

Programs can be as complex as you want to make them. We won't attempt to demonstrte a program with these types of statements at this point. It's much more sensible for us to say simply that as your skill in programming increases, and the programs you are able to write become more complex, you'll very quickly start thinking them up for yourself!

As you will have noticed, we sneaked some new commands into the last program, namely PRINT, INPUT, and END, which you hadn't seen before. As we mentioned at the start of the chapter, some instructions only really have meaning when they're used in programs, and these, along with FOR and IF, are just that sort. It's appropriate at this point, now that you've been introduced to the basic elements of program construction, to look at these statements in detail.

Program statements

The statements in this section don't have a syntax as such. Rather, they are followed by a set of information which the statement needs to do its job.

DATA <list of constants>

In all the programs we've looked at so far, we've asked the computer to process things that it already knows about, such as a change in colour or the width of a circle. What do we do though if we want to get some outside information into the computer? For example, how do we tell it the number of miles a car went on a

given amount of petrol in a program to work out the average consumption? One answer is to use a DATA statement. DATA statements don't actually do anything, they simply tell the computer that the numbers or words which follow are data which is to be accessed by a READ command (which we'll look at later). Examples of data statements are:

DATA 1,2,3,4,5,6,7,8,9,10

DATA January, February, March, April, May, June

A DATA statement can contain as many numbers or words as can be fitted onto the line, each separated by a comma, and any number of DATA statements can be used in a program – data is accessed as though it were one long list, so it doesn't matter where the DATA lines are in a program as long as they're in the right order. Any types of numbers are allowed, except for numeric expressions, such as 2+3. If a string constant is included which includes a comma, colon or space, such as 3rd January, then it must be surrounded by " " marks. (A string is a collection of characters such as a name, a telephone number, or a word.)

Finally, the data included in the DATA statement must be the same type as that expected by the READ statement which must access it.

DIM <list of subscripted variables>

The DIM statement brings us on to the subject of arrays, as it is used to specify the size of an array in memory. An array is a special type of variable allowing a number of related variables to be stored and accessed easily.

The simplest type of array consists of a series of variables, grouped together under a single array name, say "A". At the beginning of a program you must define how many variables you wish to store in A. This is called DIMensioning. A can be set to store ten variables using the command DIM A (10).

Each variable is given its own number within the array and can be accessed by giving the array name and the number of the variable. For example, A(3) is the third variable and A(7) the seventh.

Arrays allow numbers to be used for variable names where it

would be difficult to create new variable names. If we wanted to store and print the first ten squared numbers, we could have ten statements of the type LET A = 1*1, LET B = 2*2 and so on. This is impractical in the extreme. An easier way of doing this is to use a FOR. . .NEXT loop and an array, with a program like this:

```
10 DIM A(10)
20 FOR I=1 TO 10
30 A(I)=I*I
40 PRINT I,A(I)
50 NEXT I
```

If you follow the program through, beginning with I = 1, the first statement sets A(1) = 1*1. The next statement sets A(2) = 2*2, and so on up to 10. Array variables can now be manipulated as we want – we could divide each one by 10, for example, by adding the line:

35 A(I) = A(I)/10

END

The END statement quite simply tells the computer that it has reached the end of the program, and instructs it to stop program execution, close any files which the program has asked it to create, and return to command level, thereby causing the OK prompt to be displayed on the screen.

ERROR <integer expression>

MSX-BASIC contains a number of pre-defined error messages, some of which we've already encountered. A full list of these is given in Appendix B. Each error message has a number associated with it, which may be in the range 0 to 255. As you can see, if you look at the appendix, numbers 0 to 60 have already been set aside, which leaves you numbers 61 to 255 to write your own messages. It's best to write your messages starting at 255 and move downwards, so as to ensure compatibility should the error messages contained in MSX-BASIC be expanded. An example of how you might include your own error codes in a program is as follows:

```
10 ON ERROR GOTO 1000
100 IF A$="No" THEN ERROR 250
1000 IF ERROR=250 THEN PRINT "Are you
absolutely sure?"
```

FOR

See page 42.

GOSUB <line number>

We have already seen how the GOTO statement causes the computer to jump to a specified line number. The GOSUB statement is an advanced sort of GOTO in that it tells the computer to go to a subroutine. This is a part of the program which has been written for a specific purpose and may be called a number of times from within the main program. As an example, if we extended the circle drawing program to include the question "Shall I draw some squares?" then this could be used s a response if the answer to the question "Shall I draw some circles?" was "no". Add another option "Shall I play a tune?" and you can see that the program would start to get a bit complicated. To simplify matters, if the answer "yes" was given to any of the questions, then a GOSUB statement causing the computer to move to a subroutine which either drew circles, or drew squares, or played a tune would be advisable.

At the end of a subroutine a RETURN statement must be included. This instructs the computer to return to the line immediately after the GOSUB statement which caused the subroutine to be executed.

Without writing out all of the program lines, the structure of our super-extended circle drawing program, using subroutines, would be as follows:

```
10 SHALL I DRAW SOME CIRCLES
20 IF "yes" GOTO 100
30 SHALL I DRAW SOME BOXES
40 IF "yes" GOTO 120
50 SHALL I PLAY A TUNE
60 IF "yes" GOTO 140
```

```
70 IF "no" GOTO 500
80 PRINT "Sorry I don't understand.
Try again."
90 GOTO 10
100 GOSUB 200
110 GOTO 10
120 GOSUB 300
130 GOTO 10
140 GOSUB 400
150 GOTO 10
200 REM Subroutine to draw some circles
.
.
.
.
.
299 RETURN
300 REM Subroutine to draw some squares
.
.
.
.
.
399 RETURN
400 REM Subroutine to play a tune
.
.
.
.
.
499 RETURN
500 END
```

We shall be looking at REM on page 59.

N.B. It is always advisable to put subroutines at the end of the program, and to enter them via a GOTO command, as this avoids them being entered inadvertently during normal execution of the program. It is necessary, however, to stop the program so that it does not run into the GOSUB routine, but give the message

"Return without GOSUB" as it encounters the RETURN command and finds that it has nowhere to return to. In the last example we used GOTO 10 to stop the program running into line 200.

GOTO <line number>

See page 47.

IF <expression> THEN <expression> ELSE <expression>

See page 46.

INPUT ["<prompt>";]<list of variables>

As you hopefully saw with the program which asked you if you wanted the computer to draw some circles, the INPUT statement is used to instruct the computer that some input from the keyboard is required before execution of the program can continue (in the program this was either "yes" or "no"). So that you know that the computer is expecting a reply, it displays a question mark on the screen which, if a prompt has been included in the statement, can be preceded by it. The last program showed how to use a primpt with INPUT. In the program, the response was put into the variable called A$ and the computer was told that A$ had to be "yes" or "no", otherwise it should respond that it did not understand what you had typed in and ask you to start again. Whenever an INPUT statement asks for information, your reply must be in the form that the computer is expecting, otherwise the computer will respond with an error message until the input it receives is acceptable.

The error messages associated with incorrect input are as follows.

Responding with the wrong type of data, for example a letter instead of a number will cause the computer to print:

?Redo from start

Responding to input with too many data items, for example 1,2,3 when the computer only expected two numbers will produce:

?Extra ignored

and finally responding to INPUT with too few data items will result in:

??

and the computer will wait for the outstanding data items.

You can get out of INPUT by pressing the CONTROL and STOP keys simultaneously. Typing CONT will cause execution to resume at the INPUT statement.

LINE INPUT ["<prompt>";]**<string variable>**

LINE INPUT works in the same way as INPUT, the exception being that it allows you to assign an entire line of up to 254 characters to a string variable without the use of the normal quote marks. No question mark will appear on the screen unless you specify one in the prompt. Anything you type in following the prompt will be assigned to the string variable. Exit from and re-entry to LINE INPUT is the same as for INPUT.

[**LET**] **<variable>=<expression>**

The LET statement assigns the value of an expression to a variable, eg LET C=2, LET A$="yes" etc. Note that LET is optional – all that you need is the =sign, so that C=2 or A$="yes" would be equally valid.

LPRINT [**<list of expressions>**]
LPRINT USING <string expression>;<list of expressions>

These are used to print data on a line printer. For details of how they work see PRINT and PRINT USING.

MID$ (<string expression 1>,n[,m])=<string expression 2>

MID$ allows you to replace a portion of one string with another. Characters in string 1 are replaced by those in string 2, starting at position n in string 1. m can be used to specify the number of characters from string 2 that are used in the replacement. It is impossible to replace more characters than there are in string 1. As an example,

```
10 LET A$="Sunrise"
20 LET B$="sets up"
30 MID$(A$,4,4)=B$
40 PRINT A$
```

would change A$ from Sunrise to Sunsets.

NEXT <variable>

See page 42.

ON ERROR GOTO

You saw how the ON ERROR GOTO statement worked when we looked at ERROR. ON ERROR GOTO tells the computer what to do if an error occurs. If you do not tell the computer what to do from within the program, or if you include the line ON ERROR GOTO 0, then the computer will halt execution of the program and type the error message on the screen. This allows the programmer more control over any errors which might occur in a program, and even allows him to recover from errors which would normally have cuased a program to stop. It's best to put this statement right at the beginning of your program.

ON <expression> GOTO <list of line numbers>

ON ERROR GOTO is just one example of the ON. . .GOTO statement. It can be more generally used to provide a set of optional GOTOs, the one being chosen depending on the value of the expression. For example, if the value of the expression is 3, then the program will jump to the third line number in thelist following GOTO.

If the value of the expression is zero, or greater than the number of the items in the list, but less than or equal to 255, then the GOTO will be ignored and the next statement executed.

The positions at which items are printed will depend on the punctuation used to separate items in the list, as follows:

, Print lines are divided into zones of 14 spaces each. A comma causes the item following it to be printed at the beginning of the next available zone.

; A semicolon causes the next item to be printed immediately after the one preceding it. Typing one or more spaces between items will have the same effect.

If a comma or semicolon is typed at the end of the <list of expressions>, the next PRINT statement will begin printing on the same line, spaced accordingly. If a comma or semicolon isn't included, the next PRINT statement will start printing on the next line down.

Printed numbers are always followed by a space. Positive numbers are also preceded by a space, while negative numbers are preceded by a − sign.

To save you from having continually to type the word PRINT, you can, if you wish, replace it with a question mark, since MSX-BASIC recognises the two as meaning the same thing in PRINT statements.

To demonstrate the above characteristics of the PRINT statement, and in the finest tradition of computer books, why not try the following:

ON<expression> GOSUB<list of line numbers>

ON. . .GOSUB works in the same way as ON. . .GOTO with each number in the list of line numbers being the first line in the subroutine to be moved to.

POKE<address in memory>, <integer expression>

POKE is used in the same way as the VPOKE command we introduced in the section on graphics commands, and should be used with just as much caution. It is used to put a certain value into a specific memory location.

<address in memory> is the address of the memory location to be poked, and must be in the range −32768 to +65535. If the value is negative, it is taken as being subtracted from 65536, so that, for example, −1 = 65535.

<integer expression> is the data to be poked, and must be in the range 0 to 255. From our introduction, you'll see that the maximum amount of RAM that can be poked to is 64K (64K = 64 × 1024 = 65536 bytes).

PRINT ["<list of expressions>"]

The PRINT statement is used to print information onto the TV screen, as we saw several times in the final version of the circle drawing program. If PRINT is used without a following list of expressions, then a blank line is printed. If an expression is included, then this will be printed, eg PRINT ''Hello'' will cause the word ''Hello'' to appear on the screen.

```
10 CLS
20 PRINT "Hello"
30 PRINT "Hello","Hello"
40 PRINT "What's";"going"
50 PRINT "on"
60 PRINT "here"
70 PRINT "then";
80 PRINT "?"
```

Pretty mind-bending stuff, really, isn't it?!

If you try running this program on either of the graphics screens, you will see that nothing happens. In order to get around this problem, a program of the following type is needed (as you saw when we looked at LOCATE).

```
10 OPEN "GRP:" FOR OUTPUT AS #1
20 SCREEN 2
30 PRINT #1,"Whatever you want the
computer to print"
40 GOTO 40
```

PRINT USING <string expression>;<list of expressions>

Compared to the ease of use of the PRINT statement, PRINT USING is something of a handful. To save confusing you, it has been banished to a later chapter.

READ <list of variables>

We introduced READ earlier on when we spoke about the DATA statement, and said then that the two are always used in conjunction with each other. Take another look at what we said

about DATA, and then run the following program:

```
10 PRINT "The sum of the numbers in";
20 PRINT " the DATA";
30 PRINT " statements is";
40 READ A,B,C,D,E,F
50 G=A+B+C+D+E+F
60 PRINT G
70 DATA 1,2,3
80 DATA 4,5,6
```

Incidentally, we've delibertely made this program a bit longer than it had to be to demonstrate a few of the things we've told you about the PRINT and DATA statements, ie, the use of semicolons, the space that precedes numbers when they're printed, and the fact that data is read sequentially wherever it happens to be. If you didn't spot these, you obviously haven't been paying attention! Note also that although data can appear anywhere in the program, we've put it at the end so that it doesn't get confused with the rest of the program.

Not only can a READ statement access more than one DATA statement, a number of READ statements can access the same data. We shall demonstrate this when we look at the RESTORE command, so don't delete the program!

REM <remark>

REM is used quite simply to allow you to enter explanatory remarks into the program to say what particular parts of the program are doing. We used REM in the super-extended circle drawing program to show which subroutine was doing what. REM statements are not executed by the computer but, as you can see from the way the program was structured, they can be used as entry points to subroutines via GOSUBs, or from a GOTO statement. You can add remarks to the end of a line using ' or :REM, but it is advisable to dedicate a full line to a remark since this makes the program structure easier to follow.

RESTORE [<line number>]

As we noted with the READ statement, for data to be reread, it must be restored. If you kept the program you typed in under READ, then convert it to the following and you'll see what we mean:

```
10 PRINT "The sum of the numbers in";
20 PRINT " the DATA";
30 PRINT " statements is";
40 READ A,B,C,D,E,F
50 G=A+B+C+D+E+F
60 PRINT G
70 RESTORE
80 PRINT "The product of the numbers
is" ;
90 READ A,B,C,D,E,F
100 H=A*B*C*D*E*F
110 PRINT H
120 DATA 1,2,3
130 DATA 4,5,6
```

If you try to run the program without the RESTORE statement, the error message "Out of DATA in 90" will appear. Note also that is is possible to assign a line number to RESTORE, thereby allowing only data which follows that line number to be reread; RESTORE 120 will also allow the above program to run, whilst RESTORE 130 will not.

N.B. You will notice that lines 70 and 90 in the program above are unnecessary. Having assigned values to A,B,C,D,E and F, there's no need to read the data again (line 90). The reason that the program is written this way is to show as simply as possible how RESTORE works. You'll find it embedded in more complex programs later in the book, where it is necessary.

RESUME

The RESUME statement tells the computer what to do after an error recovery has been performed. Any one of four formats can be used for RESUME, depending on where it is that you want execution of the program to resume:

RESUME or RESUME O	causes execution to resume at the statement which caused the error
RESUME NEXT	causes execution to resume at the statement immediately following the one which caused the error
RESUME <line number>	causes execution to resume at the specified line number

RETURN

See GOSUB

STOP

The STOP statement can be used anywhere in a program to terminate execution and return to command level. When a STOP is encountered, the message:

Break in xxxx

(where xxxx is the line number containing the STOP statement) is displayed on the screen. Execution can be resumed by issuing a CONT command. You can demonstrate this for yourself by adding:

```
75 STOP
```

to the program listed under RESTORE. Unlike the END statement, the STOP statement does not close files.

SWAP <variable>,<variable>

SWAP exchanges the values of any two variables, so that if A = 10 and B = 5, then SWAP A,B will make A = 5 and B = 10. Any type of variable may be SWAPped, as long as the one it is swapped with is of the same type.

Summary

Congratulations! You've survived what should be the most difficult chapter of the book! We hope that it hasn't been too painful for you. On the way, you've learnt more than seventy of

MSX-BASIC's most important commands and statements, along with the essentials of how they're strung together into programs. We hope that we've introduced you to the basic concepts of programming in a way that is both visually stimulating and interesting. You should be able to use this section of the book as a reference guide as you read on through the remaining chapters, and the programs that we cover become more and more advanced.

In the next chapter we'll be building on the basis of what you've just learnt, by introducing programs with more complex structures, and describing the few remaining commands, statements and functions that we felt were inappropriate for this chapter. We'll also be looking in a lot more detail at data types, and covering the more useful arithmetic functions contained in MSX-BASIC. Once you've mastered all of these, you should have sufficient knowledge of your computer and its language to be able to write a program to perform virtually any task you can think of. We've included in the following chapters a number of skeleton programs to provide the basis of exam grading applications, games and music applications, leving you to adapt them so as best to serve your particular needs or interests. In the last two chapters we'll be looking at the more esoteric side of programming using the Graphics and Music Macro Languages themselves.

Working with numbers

By now, you are almost certainly aware that computers require some form of data to be supplied to them before they are able to do any useful work. In this chapter we will be looking at the types of data that a computer can use, and which type of data is used when.

MSX-BASIC supports a number of different data types. There are a total of seven types available, namely integer, fixed point, floating point, hexadecimal, octal, binary and character. We'll deal with these a little later on.

Data may be incorporated in programs in one of two ways: as a **constant** or as a **variable**. A constant doesn't change its value when a program is run. A variable can vary during a program's execution – that is, its value can alter if you want it to.

Constants

Firstly, how do you declare constants for each of the different data types? Well, we'll now introduce each of the data types and show how it's done.

An **integer** is simply another word for a whole number – those numbers which don't contain a decimal point anywhere. These have to be within the range of –32768 to 32767. To put these into programs, you just type in the number within an MSX-BASIC statement. For example, if we want to add 10 to 5, the following program will do it.

```
10 PRINT 10+5
```

No matter how many times you run this little program, the end result will always be that the value printed is 10+5 (15). Once part of a program, you're stuck with a constant's value, for better or for worse.

Now let's look at the subject of **fixed point** constants. These are positive or negtive real numbers and can contain a fractional part. If they do have a fractional part, it also goes without sying that they

have a decimal point. The combination of a single decimal point and a series of numbers will declare a fixed point constant in a program.

Acceptable constant values of this type are: −3.645, 123.87, 0.013, −0.56746, .78 and so forth.

Another type of constant which can have a decimal point in it is a **floating point** constant. These are positive or negative numbers, and can be declared in the following ways. One method is to use what is sometimes called scientific notation. The numbers are represented by three components – the **mantissa**, the letter **E** or **D**, and an **exponent**. This is best explained by example. The number 753000 would be represented as 7.53E5. The mantissa in this case is 7.53, and the exponent is 5. What the statement is saying is that the number can be expressed as 7.53 multiplied by 10 to the power 5, or in more simple terms, it is the equivalent of moving the decimal point 5 places to the right. If the exponent is negative, then the decimal point would be moved to the left. The exponent can only be an integer value in the range of −64 to 63.

Substituting a D for an E means that a number will be stored to 14 decimal places of accuracy, as opposed to 6 when E is used. Valid examples of floating point numbers are as follows:

Actual Value	**Floating Point Representation**
12400	12.4E3
0.00683	6.83E-3
−0.00004077032	−4.077032D-5
534500210	5.3450021D8

All the numeric types of data can be stored to different levels of accuracy. Single precision numbers are accurate up to 6 decimal places; double precision numbers are accurate to up to 14 decimal places. You can determine how accurate you want a number to be by using E or D for floating point constants, or the symbols # or ! for all other types of numbers. You've already seen how to declare single precision and double precision for floating point numbers. Single precision for integers and real numbers is obtained by placing an exclamation mark at the end of a list of digits, for example 145! or 89.6!

Double precision is available using the letter D in floating point

numbers, and by placing an optional # symbol at the end of a fixed point number or integer. The # is optional because as soon as the MSX computer is switched on, MSX-BASIC will assume that all numbers will be double precision, unless of course, you tell the computer otherwise. Double precision is the default precision in MSX-BASIC. So 5600 is a double precision value, as are 4573.56, 12.93#, −89.9#, and 12314.

Hexadecimal constants are perhaps totally new to you. Hexadecimal values are numbers represented in base 16. Instead of units, tens, hundreds and so on of the usual system, hexadecimal has units, sixteens, two-hundred-and-fifty-sixths (16×16) and so on. To make this a little clearer, here's how you count up to 16 in hexadecimal:

Decimal value	:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hexadecimal value	:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

Note that above 9 and up to 16, numbers become letters in hexadecimal. It is possible to have a number like "FFFF", or "BCDD" in hexadecimal code (or Hex as it's abbreviated). Hex numbers are often used to program machine code so you don't need to worry about it too much. This is the only place you'll find a reference to them in the book, so this is how you declare them. Quite simply, the prefix, **&H** is placed in front of the number. Examples of Hex numbers are: &H3FA, &H1650, &H2FE.

Octal constants are another oddity left over from the golden days of computing. Just as hexadecimal uses 16 as its base number, and decimal uses base 10, so octal uses base 8. Some people still use octal numbers, but you're not likely to want to use them. If you should, however, here's how you count up to 16 in octal:

Decimal value	:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Octal	:	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20

A prefix is again used to declare an octal constant. Instead of &H, for hexadecimal, we use **&O**. Examples of valid octal constants are &O12, &O5643, &O129.

Binary constants are denoted by the prefix **&B**. Binary is the number base two, as described in Chapter 1. We will be using these constants a little later on. As explained earlier, communicating with

a computer in 1's and 0's is just about as basic as you can get. Examples of binary constants are: &B11110000, &B00101110, &B11001101. You can represent any number from 0 to 255 using one byte.

The final type of data available is the **character string.** A character string is anything enclosed in double quotation marks. The letters A to Z, a to z, 0 to 9 and any other elements of the MSX-BASIC character set are character strings when put in double quotes.

These strings can be up to 255 characters in length. Here are a few examples:

"fred", "1234", "$%^*", "I am a string!", "345.687".

These are commonly used when using the PRINT statement, for example, when printing out a message in a program:

```
10 PRINT "Hello, I am a constant
string"
20 PRINT "I cannot be changed while
running a program."
```

Now that the string constant has had its say, we've covered all the constants available. In the next section, we'll look at variables, how to declare them, and how to use them.

Variables

Variables are names that you can use to represent values in a program. They can be altered during the execution of the program. Like constants, they can be of different types – a total of four in fact.

To begin with, we'll look at how to set up variables. As with everything else in programming, there are some basic rules to learn first. The variable name can be any length you like, providing that you don't exhaust all the available memory! The start of a variable name *must* be a letter.

There are no spaces allowed in the variable name. You can't give the variable a name which is a BASIC reserved word or which contains a BASIC reserved word. As the words are reserved for

MSX-BASIC, the language doesn't take kindly to a programmer pinching them! All these rules may seem a little restricting, but you'll very soon get used to them. For the time being, you can't go far wrong using just the single letters from A through to Z.

Variables may be declared as a particular data type by using a variable declaration character. These are put at the end of the variable name and are as follows:

% Integer variable
! Single precision variable
Double precision variable
$ Character string

Below is a sample of valid and invalid variable names:

Valid : Range% A$ MSX! Trajectory Name10$ PI%
Invalid : 15X NAMELISTS$ 2PAYROLL DIM%

An alternative means of declaring variable type is to use the DEF statement. This has the general form:

DEF<variable type> [**<expression>**],[**expression**],. . .

The variable type may be declared by one of **INT, SNG, DBL,** and **STR**, which are the respective specifiers for integers, single precision, double precision, and string data types. The expression used in the statement is a letter, or range of letters. For example, if the statement DEFINT A is executed, all variable names beginning with the letter A will be integers. If the statement DEFSTR A – Z is executed, all variables in the program will be assumed to be character strings. To change the data type of any particular variable after a DEF statement has been used, all that needs to be done is to use the variable declaration characters. The example below illustrates the DEF statement.

```
10 DEFSTR A-Z:REM DECLARE ALL VARIABLES
AS CHARACTER STRING
20 S="FRED ":R="BLOGGS"
30 PRINT S+R
40 PRINT A
```

```
50 A%=100
60 PRINT A%*10
```

Note that the strings S and R do not require the $ character declaration character. When the variable A is printed out in line 40 it is an empty string, becoming an integer in line 50.

The last type of variable is the array subscript we met earlier. We'll discuss arrays in more detail on page 000.

Once you've decided on your variable names, you can then put them into programs and start giving them values. The process of giving a value to a variable is known as **variable assignment**. Try running the following program:

```
10 REM Variable Assignment
20 PRINT NUMBER%
30 INPUT NUMBER%
40 LET NUMBER%=NUMBER%+1
50 PRINT NUMBER%
```

The only variable in this program is NUMBER%, an integer variable. When printing the value out in line 20, NUMBER% is 0. This is because when MSX-BASIC encounters a new variable name in a program, it initially assumes that its value is zero. In line 30, we have an INPUT statement which asks you to assign a value to NUMBER%. In line 40, the LET statement adds 1 to this value, and this is printed out in line 50.

If you tried typing in a letter or word to the ? prompt, the BASIC would have come up with the message – '?redo from start'. The reason for this (as you should remember) is that you'd told the computer to expect an integer to be typed in. So it'll keep waiting until you give it an integer to work with. Very patient things, computers . . .!

As you should also remember, the LET part of line 30 is optional. The statement "NUMBER% = NUMBER% + 1" would have done just as well. You can assign any variable in this way.

Now try changing the % signs to $ signs, making NUMBER into a string variable. If you run the program now, you'll get another message from MSX-BASIC. This one will be accompanied by a

beep and won't be nearly so pleasant. In fact, what you will have encountered is the error message 'Type mismatch in 30' (dramatic music, please). You will come into contact with many of these infuriating little messages as you program – a full list is included in Appendix A. The reason that the message was printed was because the computer tried to add 1 to whatever word you typed in, and got very confused. It's rather like trying to add apples and oranges together – they are two different sorts of object and cannot be treated in the same way. If you remove the offending line (40), then everything in the garden will be rosy again: whatever word you typed in will be printed out, and the computer will not have to try and add "apples and oranges" together. But try this before going on. Change line 40 to:

```
LET NUMBER$ = NUMBER$ + "1"
```

Putting quotes round the 1 has turned it into a string and the computer is quite happy to add strings together.

It's useful to know how much memory the computer will require for the values of variables. The table below shows how much space each type of variable will need, including array variables.

VARIABLES:

Data Type	No. Of Bytes Required
Integer	2
Single Precision	4
Double Precision	8
Strings	1 per character + 3 e.g. the string "FRED" would require 7 bytes in all

ARRAYS:

Data Type	No. Of Bytes Required
Integer	2 per element
Single Precision	4
Double Precision	8

Having introduced all the different types, we'll now look a little more at the programming side of things, and introduce some simple maths onto the scene.

Mathematical Expressions

Mathematics is not the most popular of subjects. Unfortunately, computers are quite good at maths, so we can't really overlook it. Starting with the very simple stuff first, just type in:

PRINT 10*10

The computer will reply with '100'.

This is the computer behaving as a simple calculator. What you just asked it to do was print out 10 multiplied by 10. As there is no multiply symbol (×) available, the computer uses an asterisk instead. Try the following three as well:

PRINT 10+10
PRINT 10-5
PRINT 10/2

They were addition, subtraction and division operations – everything you need for simple arithmetic. Now let's go about putting these into programs. The following program simply carries out all four arithmetic operations on any two numbers you supply:

```
10 INPUT "Two numbers, please";A,B
20 PRINT "A + B = ";A+B
30 PRINT "A - B = ";A-B
40 PRINT "A * B = ";A*B
50 PRINT "A / B = ";A/B
```

Not a staggeringly complex program, but it works! We'll move onto a more ambitious program now. The aim of this program is to accept words and numbers from the user and then carry out some simple arithmetic. What has to be input is the name of a maths operation, in capitals, and two numbers. For example, a user could type: MULTIPLY,5,4 and the result would be 20. There are a few new things to learn before you can do this.

The program "knows" about the four words associated with arithmetic operations. They are represented as constants in the program. What the program has to do is compare the word supplied by the user with its own constant words. The way MSX-

BASIC compares two character strings is using the equals sign, just as it would compare two numbers.

So the statement which the program uses to decide whether or not it should multiply, divide, add or subtract is the IF statement introduced earlier. If the word EXIT is typed in instead of the name of a maths operation, then the program prints out a cheery goodbye message, and finishes. Otherwise it compares the word input against the four arithmetic terms. If there is no match at all, then the computer tells the user that it doesn't have a clue what to do, and asks for new information to be put in. The program looks like this:

```
10 REM SIMPLE CALCULATOR
20 REM PRINT INSTRAUCTION PAGE
30 CLS
40 PRINT "CALCULATOR PROGRAM":PRINT "---
---------------"
50 PRINT "Type in the name of the mathem
atical ";
60 PRINT "operation followed by two numb
ers.":PRINT
70 PRINT "The instruction and two number
s should ";
80 PRINT "be separated by   commas, for
 example:"
90 PRINT "MULTIPLY,5,4"
100 PRINT "Type 'EXIT,0,0' when you're f
inished.":PRINT
110 PRINT:INPUT "Input data : ";A$,X,Y
120 IF A$="EXIT" THEN GOTO 180
130 IF A$="ADD" THEN SUM=X+Y:PRINT "The
sum of ";X;" and ";Y;" is ";SUM:GOTO 110
140 IF A$="SUBTRACT" THEN SUM=Y-X:PRINT
"The result of ";X;" from ";Y;" is ";SUM
:GOTO 110
150 IF A$="MULTIPLY" THEN SUM=X*Y:PRINT
"The product of ";X;"multiplied by ";Y;"
 is ";SUM:GOTO 110
160 IF A$="DIVIDE" THEN SUM=X/Y:PRINT "T
```

As you can see, the values are totally different for what seems a totally straightforward statement. In fact, MSX-BASIC would come up with the number 26. This is because it has been instructed that all multiplication should be done before addition. Hence the term "precedence".

Multiplication has precedence over addition. The order of precedence is summarised for all arithmetic operators thus:

Symbol	Operation	Example
^	Exponentiation	A^B
–	Negation	–A
*,/	Multiplication and floating point division	A*B A/B
¥	Integer division	A¥B
MOD	Modulo arithmetic	A MOD B
+,–	Addition and subtraction	A+B A–B

This order can be altered by the use of parentheses. With the example shown above, if we want to carry out addition before multiplication, we would have to write the expression thus:

```
A=(B+C) * D
```

The operation in the brackets is always evaluated first. If B is 2, C is 4 and D is 6 then the result would be 36.

Exponentiation is simply the process of raising a number by a power. For example, if you type:

```
PRINT 10^2
```

the result 100 will be printed out. In other words, the statement you typed in is the same as saying "raise 10 to the power of two". The notation used by MSX-BASIC in this case is equivalent to the following: 102. Other examples and their results are printed below:

Statement	Result	Equivalent Notation
PRINT 10 ^ 4	10000	104
PRINT 2 ^ 8	256	28
PRINT 90 ^ 2.5	76843.347142016	902.5
PRINT 1.37 ^ 40	294321.9730757	1.3740

```
he result of ";Y;" divided by ";X;" is
" ;SUM:GOTO 110
170 PRINT "Sorry, I don't understand."
;A $;" Try again":GOTO 110
180 PRINT "Goodbye then!"
190 END
```

Note the use of the REMark statements. These are quite useful if you leave a program on tape for a while, and can't remember quite what it does. It also enables you to follow the logic (or rather what you thought was logic!) of the program if you want to alter it at a later date. Follow the program through – it's relatively easy to understand.

Addition, subtraction, multiplication and division are not the only mathematical operations possible with MSX-BASIC. The others available are **exponentiation, negation, integer division,** and **modulus arithmetic**. All arithmetic operators obey what is known as the **precedence rule**. When the MSX-BASIC interpreter looks at a line full of these arithmetic operators, it needs to decide which operation to carry out first. Given the example statement:

A = B + C*D

it is difficult for MSX-BASIC to decide whether you want to add B to C first then multiply the result by D, or multiply C and D together first and add B to the result. If you substitute numbers for the letters in the above equation you'll see where the difficulties arise.

Assuming that B is 2, C is 4, and D is 6, then for the first case, add B to C then multiply by D, the steps will look something like this:

First add 2 to 4 = 6
Multiply 6 by 6
Therefore A = 36

And for the second case, where C*D is carried out first, followed by the result being added to B:

First multiply 4 by 6 = 24
Add 24 to 2
Therefore A = 26

As you can see, the values are totally different for what seems a totally straightforward statement. In fact, MSX-BASIC would come up with the number 26. This is because it has been instructed that all multiplication should be done before addition. Hence the term "precedence".

Multiplication has precedence over addition. The order of precedence is summarised for all arithmetic operators thus:

Symbol	Operation	Example
^	Exponentiation	A^B
−	Negation	−A
*,/	Multiplication and floating point division	A*B A/B
¥	Integer division	A¥B
MOD	Modulo arithmetic	A MOD B
+,−	Addition and subtraction	A+B A−B

This order can be altered by the use of parentheses. With the example shown above, if we want to carry out addition before multiplication, we would have to write the expression thus:

A=(B+C) * D

The operation in the brackets is always evaluated first. If B is 2, C is 4 and D is 6 then the result would be 36.

Exponentiation is simply the process of raising a number by a power. For example, if you type:

PRINT 10^2

the result 100 will be printed out. In other words, the statement you typed in is the same as saying "raise 10 to the power of two". The notation used by MSX-BASIC in this case is equivalent to the following: 102. Other examples and their results are printed below:

Statement	Result	Equivalent Notation
PRINT 10^4	10000	104
PRINT 2^8	256	28
PRINT 90^2.5	76843.347142016	902.5
PRINT 1.37^40	294321.9730757	1.3740

Integer division differs from the normal floating point division in the following way. The operands are truncated to integer values, the division takes place, then the result is truncated to an integer. Truncating a number means "cutting off" everything after the decimal point. After truncation, 2.57 would become 2, 9.999999 would become 9 etc. The symbol to denote integer division is perhaps unfamiliar to you. It is the Japanese equivalent of our pound sign and normally represents the YEN, the currency of Japan. If we break the process of integer division up into steps. you'll be able to see exactly how it works. As an example, let's divide 345.978 by 12.866 using integer division. The steps the BASIC uses are outlined as follows:

Statement: PRINT 345.978¥12.866

1	Truncate	:	345.978 to 345
2	Truncate	:	12.866 to 12
3	Divide	:	345 by 12
4	Result	=	28.75
5	Truncate	:	28.74 to 28
6	Result	=	28

If we had used normal division (using the / symbol) the final result would have been quite different (26.890875174879 in fact).

Finally, we come to **modulo arithmetic**. This is denoted by the MOD operator. What this does is to give you the integer remainder from a division. If you type in 'PRINT 7.86 MOD 2'. The result printed out would be 1. The way that MSX-BASIC arrives at this figure is to truncate 7.86 to 7, divide 7 by 2, thus leaving a remainder of one. Again we have a list of examples for you to look at:

Statement	**Result**	
1234.4321 MOD 11.31	2	(1234/11 = 112 with remainder 2)
100 MOD 10	0	(100/10 = 10 with remainder 0)
76 MOD 13	11	(76/13 = 5 with remainder 11)
99.99 MOD 31	6	(99/31 = 3 with remainder 6)

An important point to bear in mind with any type of division is: *Don't divide anything by zero!* The computer will always generate an error message if you try to divide any number by 0. You have to

be especially careful with the MOD operator and integer division. If you have a statement such as 7 MOD 0.45, or 7¥0.45, both these operators will truncate 0.45 to 0, resulting in an error message. Also watch out for it when using variables. You may try and divide a number using a variable which has not been assigned a value.

That's an end to the arithmetic operators (so you can breathe a sigh of relief!), but these are not the only type of operators available. There are two more sets to look at in the next section. Before you throw your hands up in despair, they're very useful and above all, very easy to understand.

Logical and Relational Operators

These types of operators will frequently be used in longer programs and you'll probably use relational operators more than even the arithmetic operators, so it's worthwhile getting to know them.

All that **relational operators** do is to compare things. It's as simple as that. They can see if two things are the same, if one is less or greater than the other, or if they're totally unalike.

Yet another table should be enough to summarise the relational operators. They are *very* simple to use. There are six of them, and here they are along with what they compare:

Operator	**Function**	**Example**
=	Tests for one thing equal to another	X=Y
<>	Tests for inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

These operators are particularly useful in IF. . .THEN statements. For example, if we have a set of class examination marks then we would probably want to work out what grade each student should receive. We'll put this into practice in a full-blown program below. Students' names, their subject and their examination marks for each of three subjects are entered into the program. The program checks or **validates** the data on input so that impossible to enter exam marks such as −34% or 230%. The final

output from the program is the student's name, the grade for each exam, and the average grade over all subjects. After each student has been processed the program ask if any more information is to be supplied.

A new function **INT** is also introduced. What INT does is to round down, or truncate a number. (Refer to Appendix A for a full list of other MSX-BASIC functions). A subroutine is also used to validate the numerical data. As there are three different exam scores to be tested with just one subroutine, before jumping to the subroutine, a variable called TEMP! is assigned to the value of a score. Each time a subject score is tested, the value of TEMP! changes. A subroutine is also used to assign grades to the scores, with the variable TEMP! used in the same manner again.

The steps of the program can be summarised in words. This method of planning a program is known as an **algorithm** (which is essentially the same as the flowchart we introduced earlier on). The algorithm for the program is:

START Input the Student's Name
Input Biology, Chemistry, and Physics marks
Check the data is correct
Convert each of the marks into corresponding grades
Average the three marks and round down the result
Print out name, Biology, Chemistry and Physics grades, plus the examination average.
Ask if there is more data to be processed, if there is then go to the START again, otherwise stop.

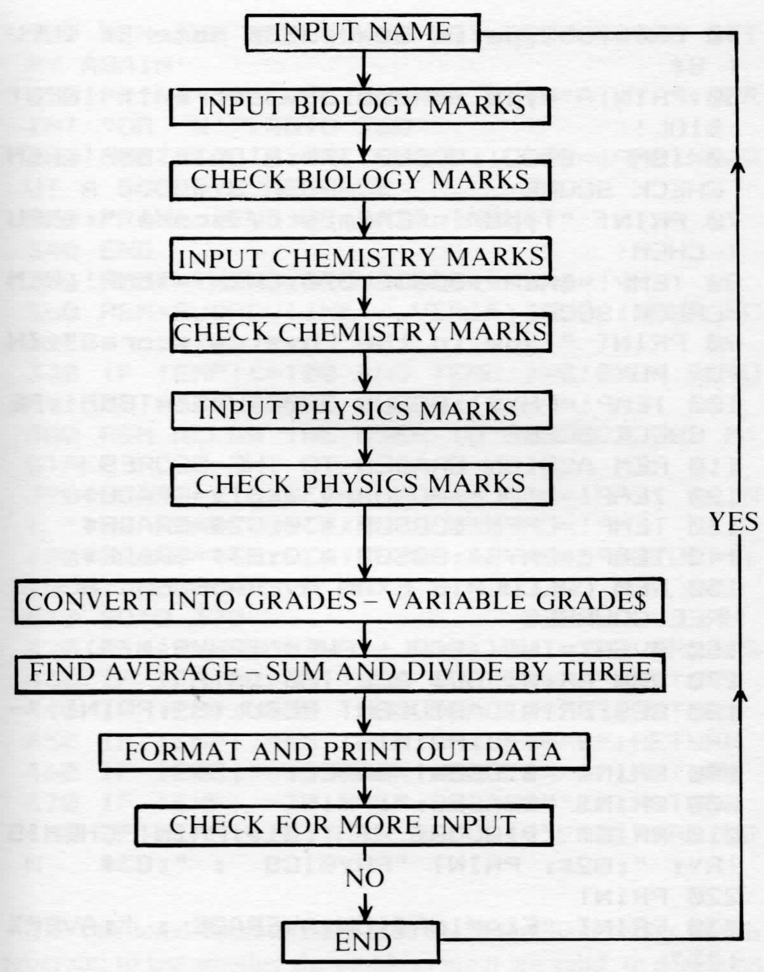


Figure 6 Flow diagram for grade calculation program.

This is shown in the flow diagram in Figure 6. The final program is listed below. It is well commented to show you how each step of the algorithm has been translated to MSX-BASIC.

```
10 REM PROGRAM EXAM SCORES
20 REM CLEAR SCREEN AND INPUT DATA
30 CLS
```

```
40 PRINT "Type in student's name ": INPU
T S$
50 PRINT "Type in Biology score ": INPUT
 BIOL!
60 TEMP!=BIOL!:GOSUB 370:BIOL!=TEMP!:REM
 CHECK SCORE
70 PRINT "Type in Chemistry score ":INPU
T CHEM!
80 TEMP!=CHEM!:GOSUB 370:CHEM!=TEMP!:REM
 CHECK SCORE
90 PRINT "Type in the Physics score ":IN
PUT PHYS!
100 TEMP!=PHYS!:GOSUB 370:PHYS!=TEMP!:RE
M CHECK SCORE
110 REM ASSIGN GRADES TO THE SCORES
120 TEMP!=BIOL!:GOSUB 430:G1$=GRADE$
130 TEMP!=CHEM!:GOSUB 430:G2$=GRADE$
140 TEMP!=PHYS!:GOSUB 430:G3$=GRADE$
150 REM CALCULATE EXAM AVERAGE FOR ALL T
HREE COURSES
160 AVER%=INT((BIOL!+CHEM!+PHYS!)/3)
170 REM PRINT OUT ALL THE DATA
180 CLS:PRINT "STUDENT RESULTS":PRINT "-
---------------":PRINT
190 PRINT "STUDENT NAME : ";S$
200 PRINT "GRADES:":PRINT
210 PRINT "BIOLOGY   : ";G1$:PRINT"CHEMIS
TRY: ";G2$: PRINT "PHYSICS   : ";G3$
220 PRINT
230 PRINT "EXAMINATION AVERAGE : ";AVER%
;"%"
240 PRINT:PRINT
250 REM SEE IF THE USER WANTS TO CARRY O
N OR NOT
260 PRINT "ANY MORE STUDENT DATA" : INPU
T "TO PROCESS (Y OR N)";REPLY$
270 REM IF THE USER TYPES "Y" THEN CARRY
 ON ELSE FINISH
280 IF REPLY$="Y" THEN GOTO 30
290 IF REPLY$="N" THEN GOTO 330
```

```
300 REM THE REPLY WAS NOT RECOGNISED - T
RY AGAIN!
310 PRINT "PLEASE ANSWER WITH A 'Y'": PR
INT "OR 'N'":GOTO 260
320 REM DEAL WITH THE 'N' CASE - PRINT O
UT A GOODBYE MESSAGE
330 PRINT "END OF DATA INPUT"
340 END
350 REM******SUBROUTINES******
360 REM*SUBROUTINE - VERIFY THE INPUT SC
ORES
370 IF TEMP!<=100 AND TEMP!>=0 THEN RETU
RN:REM DATA IS OK
380 REM ALLOW THE USER TO CORRECT HIS MI
STAKE
390 PRINT "THAT VALUE WAS NOT IN" : PRIN
T "THE RANGE OF EXAM SCORES"
400 PRINT "TRY ENTERING A VALID VALUE ":
INPUT TEMP!
410 GOTO 370
420 REM*SUBROUTINE - WORK OUT THE GRADES
430 IF TEMP!>=65 THEN GRADE$="A":RETURN
440 IF TEMP!>=55 THEN GRADE$="B":RETURN
450 IF TEMP!>=45 THEN GRADE$="C":RETURN
460 IF TEMP!>=35 THEN GRADE$="D":RETURN
470 IF TEMP!>=25 THEN GRADE$="E":RETURN
480 IF TEMP!<25 THEN GRADE$="FAIL":RETUR
N
```

The relational operators are used for three main things in this program: to test whether the numbers input are valid, to decide on the grade assigned, and finally to examine the user's reply to the question "ANY MORE STUDENTS TO PROCESS?". In the case of assigning grades, all that the program does is to ask some questions. If the exam mark is GREATER THAN OR EQUAL TO the grade corresponding to 65, which is where the boundary for an A grade starts, then the GRADE$ variable's value becomes "A", and so on for all the possible grades.

You may have noticed something odd about the statement which

checks if the numbers input are in range – line 370. That line contains the word AND which is a **logical operator**. This is the final class of operator we'll discuss in MSX-BASIC.

Logical operators are most often used in IF. . .THEN operations. They perform logical or **Boolean** operations. They are NOT, AND, OR, XOR, and EQV. Taking the AND condition which we used in the previous program:

370 IF TEMP!<=100 AND TEMP!>=0 THEN RETURN

What that line was testing for was both conditions being true. If both were true THEN some sort of action would be taken, in this case a return from the subroutine.

In computer logic 'true' means 'yes' and 'false' means 'no'. As an example, consider the variable X, where X is equal to 10. The following expressions show how true and false are applied to logical operations:

X<20 True
X=10 True
X>10 False

We could extend this with a variable called Y, where Y=50. We can now see how an expression like:

(X=10) AND (Y=40)

can be evaluated. In this case the answer would be false, because one of the conditions has not been met. It is not true to say that X=10 and Y=40, because both expressions must be true. On the other hand, an expression like:

IF X=10 OR Y=40

Will give a true answer because one of the two expressions is true.

Conditional testing is one of the ways in which a computer appears to "think". It can compare two values and seem to make a decision about what to do next based on its comparison. All this uses is the computer's ability to evaluate simple mathematical expressions and conditions.

You can represent the way a logical operator works by using a **truth table**. In this type of table, the two states of the conditions being tested are represented as T (or 1) if the condition is true, and

F (or 0) if it is false. The outcome of the logical operation will depend on what the state of the two conditions were. The table below summarises the logical operators. They are listed in order of precedence.

NOT is TRUE only when a condition is false. It will also be FALSE when a condition is true. For example:

```
10 X%=0
20 X%=NOT(X%)
```

X% will become −1. Another use is demonstrated below

```
10 X%=-1
20 IF NOT X% THEN BEEP
```

The program will generate a beep as NOT X% is false (zero)
The truth table for NOT is:

X	NOT X
T	F
F	T

AND is TRUE only when both conditions are true. For example:

```
10 X=50:Y=100
20 IF (X=50) AND (Y=100) THEN BEEP
```

As both conditions are true, a beep sound will be generated.
The truth table for AND is:

X	Y	X AND Y
T	T	T
T	F	F
F	T	F
F	F	F

OR is TRUE when one or both of the conditions are true, for example:

```
10 X=10:Y=40
20 IF (X<100) OR (Y>40) THEN BEEP
```

The test is true because the value of X matched one of the conditions.

The truth table for OR is

X	Y	X OR Y
T	T	T
T	F	T
F	T	T
F	F	F

XOR (exclusive OR) is TRUE only when one condition is true and the other is false, for example:

```
10 X=10:Y=40
20 IF (X<100) XOR (X>40) THEN BEEP
```

The outcome is true. If Y was 100 though, the test would have failed.

The truth table for XOR is:

X	Y	X XOR Y
T	T	F
T	F	T
F	T	T
F	F	F

EQV is TRUE when both conditions have the same state; that is they are either both false or both true.

Examples, where in both cases the conditional statement is true, are:

```
10 X=10:Y=10
20 IF X=10 EQV Y=10 THEN BEEP
```

or

```
10 X=10:Y=10
20 IF X<>10 EQV Y<>10 THEN BEEP
```

The truth table for EQV is:

X	Y	X EQV Y
T	T	T
T	F	F
F	T	F
F	F	T

IMP (Implication) – A strange one this. The condition is FALSE only if the first condition tested is true and the second is false. For example:

```
10 X=10:Y=50
20 IF X=10 IMP Y=10 THEN BEEP
```

The above program will not beep, as X = 10 is true and Y = 10 is false.

The truth table for IMP is:

X	Y	X IMP Y
T	T	T
T	F	F
F	T	T
F	F	T

This completes our guided tour through numbers and operators that MSX-BASIC can work with. From now on we'll be doing a little more programming.

Using Arrays

In this section, we'll look at the **array** data structure in much more detail giving examples of its use. As you will already know, arrays are defined using the DIM statement. Using the DIM statement, you give the array a name and then the dimensions of the array, that is how many elements it is to contain.

The array name will also define the data type of the elements to be stored in the array. For example, if an array is defined thus:

DIM A$(5)

the array will be able to store a maximum of 5 elements which will

be characters. You can't mix the data types in any one array. The single type of data allowed in an array is known as the array **base type**.

Each element of an array is to be given a unique number, or **indexed**, for easy reference. The number used to index an array is termed the array **subscript**. The following short program is used as an aid to remembering the colours a particular number refers to in MSX-BASIC.

```
10 DIM A$(16)
20 FOR I=0 TO 15
30 READ CL$
40 A$(I)=CL$
50 NEXT I
60 CLS
70 INPUT "COLOUR NUMBER, PLEASE ";N%
80 IF N%<0 OR N%>15 THEN BEEP:GOTO 60
90 PRINT
100 PRINT "COLOUR ";N%;" IS ";A$(N%)
110 REM COLOUR INFORMATION
120 DATA TRANSPARENT,BLACK,MEDIUM GREEN
130 DATA LIGHT GREEN,DARK BLUE
140 DATA LIGHT BLUE,DARK RED,CYAN
150 DATA MEDIUM RED, LIGHT RED,DARK
YELLOW
160 DATA LIGHT YELLOW, DARK GREEN,
MAGENTA,GREY,WHITE
170 END
```

The table below shows the way the array A$ is laid out. When you type in a number, it first checked to see if it is in the range of acceptable numbers. If it is correct, then the number typed in is used to 'look-up' the name of the colour in the array A$.

Array Index	Contents
0	"TRANSPARENT"
1	"BLACK"
2	"MEDIUM GREEN"
3	"LIGHT GREEN"
4	"DARK BLUE"
5	"LIGHT BLUE"
6	"DARK RED"
7	"CYAN"
8	"MEDIUM RED"
9	"LIGHT RED"
10	"DARK YELLOW"
11	"LIGHT YELLOW"
12	"DARK GREEN"
13	"MAGENTA"
14	"GREY"
15	"WHITE"

The array data structure can be viewed as a table. The advantages of using arrays is their speed. Try writing the above program using IF. .THEN statements, and you'll see what we mean:

```
80 IF N%=0 THEN PRINT "COLOUR 1 IS
TRANSPARENT"
90 IF N%=1 THEN PRINT "COLOUR2 IS BLACK"
```

etc. .

The array used in the above example was a one-dimensional array. You may also have arrays in two or even three dimensions. The program below uses a two dimensional array to store three students' grades for three subjects, as seen in the earlier examination score program. The array ST is dimensioned 3

elements by 3 elements. (The figures in brackets represent the student number and grade.)

Biology	Chemistry	Physics
ST(1,1)	ST(2,1)	ST(3,1)
ST(2,1)	ST(2,2)	ST(2,3)
ST(3,1)	ST(3,2)	ST(3,3)

Two separate arrays, N$ and SB$ are used to store the students' names, and the subject names. The program stores the students' grades and names, and outputs all the data to the screen as a table.

```
10 DIM ST(3,3),N$(3),SB$(3)
20 FOR N=1 TO 3
30 READ A$
40 SB$(N)=A$:REM SET UP SUBJECT NAME
50 NEXT N
60 DATA BIOLOGY,CHEMISTRY,PHYSICS
70 FOR I=1 TO 3:REM INPUT DATA
80 CLS
90 INPUT "STUDENT NAME ";S$
100 N$(I)=S$
110 FOR J=1 TO 3
120 PRINT SB$(J);" SCORE ";:INPUT SC
130 ST(I,J)=SC
140 NEXT J
150 NEXT I
160 CLS
170 REM OUTPUT INFO. AS A TABLE
180 FOR I=1 TO 3
190 PRINT "NAME ";N$(I)
200 PRINT
210 FOR J= 1 TO 3
220 PRINT SB$(J),ST(I,J)
230 NEXT J
240 PRINT
250 NEXT I
260 END
```

Getting things in order

The tedious process of sorting data is just one of those boring tasks which computers have the speed (and patience!) to accomplish well. Sorting is a subject which has been investigated by mathematicians and computer scientists for many years. Here we'll present a very simple sorting routine without confusing you with too much theory. We'll use arrays to store the data to be sorted in these examples.

There is a very well known method of sorting known as **bubblesort**. First, the program for the sort, followed by an explanation.

```
10 CLS
20 INPUT "NUMBER OF ITEMS TO BE SORTED"
;N
30 DIM ARRAY(N)
40 FOR I=1 TO N
50 INPUT "NUMBER PLEASE ";X
60 ARRAY(I)=X
70 NEXT I
80 CLS
90 PRINT "ORIGINAL SEQUENCE OF NUMBERS":
PRINT
100 FOR I=1 TO N:PRINT ARRAY(I):NEXT I
110 REM SORT THE DATA
120 FOR I=2 TO N
130 FOR J=N TO I STEP -1
140 IF ARRAY(J-1)>ARRAY(J) THEN SWAP
ARRAY(J-1),ARRAY(J)
150 NEXT J
160 NEXT I
170 REM PRINT OUT SORTED DATA
180 PRINT:PRINT "SORTED DATA":PRINT
190 FOR I=1 TO N:PRINT ARRAY(I):NEXT I
200 END
```

This version of bubblesort sorts the data input in ascending order. To sort data in descending order, the greater-than sign (>) is replaced with a less-than sign (<).

Bubblesort scans the elements from the 'highest' end of the array to the second value from the 'bottom', comparing the current value with the one immediately below it. If the value below the current value is higher than the current value, then the two values exhcange places. After scanning through the data once, the lowest element of the array is disregarded, and the process begins again from the highest element. Consider the following set of unsorted data:

100 7 9 1 514

The intermediate stages of the sorting process are shown in the table below:

Value of I	Value of J	Data Order
2	5	1 100 7 9 514
3	5	1 7 100 9 514
4	5	1 7 9 100 514
5	5	1 7 9 100 514

You can see how the value 100 moves its way 'up' the sequence of numbers. Bubblesort is an example of an exchange sort, quite simply because the values in the array are swapped to get them into their correct position. This sorting routine may also be used to sort names, addresses or any character strings – just change the data types of the array used and appropriate variables.

This sorting does have its drawbacks though. Assume you have a list of numbers such as 1 21 3 5 4. Just one scan along this list will put it into order – just exchange the positions of 5 and 4. But the program will keep scanning through the data, even though it is already in order. This method falls down when the numbers are *nearly* in sequence.

So how do you tell the program that everything is in order? We can tell that everything is in the right sequence when no more values need to exchange places. What can be done is to set a variable from 0 to 1 if any exchanges took place when the program last scanned through the list of numbers. If the value of the variable is one after a pass, the program knows that it still has to keep sifting through

the numbers until they are in order. The variable used in this case is commonly known as a **flag** – it signals that an event has taken place. A program would test the value of the flag to see if an exchange had taken place.

To alter the program above to make it more efficient, add the following lines:

```
115 F=0
155 IF F=0 THEN GOTO 170
```

and alter line 140 to read:

```
140 IF ARRAY(J-1)>ARRAY(J) THEN SWAP
ARRAY(J-1),ARRAY(J) :F=1
```

Sorting programs could be used for all kinds of things. Although bubblesort is not super-efficient, you should find it adequate to start with.

Mathematical Functions

MSX-BASIC supplies a number of **functions** specifically for maths operations. It's not worth looking at all of them, so we'll just take a look at the most commonly used ones. A function is supplied with a value and returns a value. The ones we'll look at are the trigonometry functions **SIN, COS** and **TAN**, and one that calculates square roots, **SQR**.

A function needs to be supplied with what is known as an **argument**. An argument for a function is the value you give the function to process. The argument always follows the function name in brackets.

The SIN function means calculate a sine function for angles (it's not one of the seven deadly SINs!). SIN takes an angle in radians and produces the sine of that angle. If you don't understand radians, and would rather use degrees, then use the following equation to convert a number from degrees to radians:

Radians = degrees * PI/180

PI is approximately equal to 3.141593. So to find out the sine of 45 degrees, we convert the value to radians and use the SIN function:

```
10 A=45
20 R=A*(3.141593#/180)
30 PRINT SIN(R)
```

The result of running this program will be the number 0.7071 appearing on your screen. The functions COS and TAN also work on adians in much the same way, and return values for the cosine and tangent of an angle respectively.

The SQR function returns the square root of a number. It will do this so long as you don't give it a negative number or zero to work with. Computers can do a lot of things, but finding out impossible square roots is not one of them. Try the following program:

```
10 PRINT SQR(100)
20 PRINT SQR(4)
30 PRINT SQR(16)
```

The values 10, 2, and 4 will be printed out. Appendix A details all the functions, mathematical and otherwise, available with MSX-BASIC.

User-defined Functions

If you can't find precisely the mathematical function you need from the range available in MSX-BASIC, there is a facility to create your own. This is the DEF FN statement, which allows you to set up a series of instructions and give them a name for later reference. For example, assuming tht you wanted to produce the square of a number from a function (assuming you didn't have the ability to use X 2). The declaration for such a function would be as follows:

10 DEF FNA(X)=X*X

The DEF FN instruction is given by the following syntax:

DEF FN<function name>(<variable>,<variable>,. .)
=<expression>

For our example, the function name is A, and the expression is X*X. The value in the brackets is known as a **dummy variable** for

the function. Dummy variables are replaced by actual values when you use, or **call** the function. Try running the following simple program:

```
10 DEF FNA(X)=X*X
20 INPUT S
30 PRINT FNA(S)
```

The program will produce the square of any number you type in. In this case, X is the dummy variable in the function, and it is replaced by the value of S in line 30. You are not limited to having a single argument for the function. If you simply wanted to multiply two arbitrary values together, then this could be achieved using the following:

```
10 DEF FNA(X,Y)=X*Y
20 INPUT S,T
30 PRINT FNA(S,T)
```

User-defined functions are used in just the same way as the functions supplied by MSX-BASIC.

Remember Pythagoras' Theorem? For those of you who don't, here goes. If you have a right-angled triangle, then the square of the hypotenuse is equal to the sum of the squares of the other two sides. So, if we want to find the length of the hypotenuse (c) of a right-angled triangle with sides (a) and (b), then we would use the following equation:

$$c = \sqrt{a^2 + b^2}$$

A suitable function to calculate the length in this case is contained in the following program:

```
10 DEF FNA(A,B)=SQR((A^2)+(B^2))
20 INPUT "LENGTH OF SIDE A ";X
30 INPUT "LENGTH OF SIDE B ";Y
40 PRINT "THE LENGTH OF SIDE C IS ";
FNA(X,Y)
50 END
```

We'll be using a variation of this function in a later graphics program. There are numerous advantages in using functions. If you're juggling equations to split the atom, then typing out those long equations is going to get very tedious.

Functions are very concise and simple to use once you've defined them. Also they are faster than using the same mathematical expression over and over again. The BASIC interpreter doesn't have to re-process what is essentially the same equation with different variables. Using functions will also save you memory space. Finally, it may make your programs easier to understand, either by yourself a few weeks after you've written them, or y some innocent bystander who marvels at your latest program and asks (always casually!), "Well, how does it do it then?. . ."

Interacting with your programs

In this section, we'll work through a few useful programs and expand more on data validation and how to produce pretty outputs with your programs.

Data Input

First we'll look at a few ways of producing easy-to-use input routines, introducing the LOCATE command, and the INKEY$ variable in the practical examples of creating menu-driven programs.

A menu driven system when applied to computer programs is exactly the same as you would find in a restaurant, in that you use the menu to select the items that you want. Before you ask yourself what food has got to do with computers, let's explain. In a restaurant, you pick out what you want to eat from the menu, and the waiter goes away and gets it for you. With the computer, you can display the options on the screen, pick out the one you want, and get the computer to go away and perform your chosen task. The easy way of doing this is to print a number on the screen next to an option, and get the user to pick the number of the option desired. This simple program displays a list of items on the screen and prompts the user to pick an option number.

```
10 REM SIMPLE MENU SYSTEM
20 REM PRINT OUT LIST OF OPTIONS
30 CLS:PRINT "1. PRINT OUT A MESSAGE"
40 PRINT "2. PLAY SOME NOTES"
50 PRINT "3. DRAW A CIRCLE"
60 PRINT "4. EXIT"
70 LOCATE 1,20
80 PRINT "SELECT OPTION NUMBER ";
90 A$=INKEY$:IF A$="" THEN 90 ELSE PRINT
 A$
```

```
100 OPT%=ASC(A$)-48
110 IF (OPT%<1) OR (OPT%>4) THEN BEEP:
LOCATE 22,20:GOTO 90
120 REM SELECT ACTION TO BE TAKEN
130 ON OPT% GOSUB 500,600,700,800
500 CLS:PRINT "HELLO. A AM AN MSX
COMPUTER!"
510 FOR I=1 TO 800:NEXT I
520 RETURN 30
600 REM PLAY SOME NOTES
610 PLAY "CDEFG"
620 FOR I=1 TO 800:NEXT I
630 RETURN 30
700 REM DRAW CIRCLE
710 SCREEN 2
720 CIRCLE (128,92),70,1
730 PAINT (128,92),1
740 FOR I=1 TO 800
750 SCREEN 1:RETURN 30
800 CLS:END:REM FINISH PROGRAM
```

Line 90 introduces the special variable **INKEY$**. This variable stores the name of a key when one is pressed. The statement on line 90 assigns the current contents of INKEY$ to the variable A$. If A$ is empty (IF A$ = ""), then no key press has taken place, so the program jumps back to the start of the line and keeps looking at INKEY$ until it has a value. Repeatedly looking to see if an event has occurred is known as **polling**. Line 90 **polls** the keyboard.

When a key has been pressed, A$ stores the name of that key. What needs to be done next is to check if the key pressed weas one of "1", "2", "3", or "4". Note that we're treating everything coming in from the keyboard as a character. We could have just used an ordinary input statement, but this leaves the program open to abuse.

You could type in G, for example, which would make the BASIC interpreter produce an error message. Programs should check for errors in input data themselves – the BASIC shouldn't need to do it.

Wha the program does is to convert the character to an integer, and then test to see if it is acceptable data. A BASIC function, ASC, converts characters into integers. All characters in MSX-BASIC have a code number, known as its ASCII code (see Chapter 1). The ASC function produces this code from a character, or character string. The ASCII codes for 1, 2, 3 and 4 are 49, 50, 51 and 52 respectively. So line 100 converts the letter input to an ASCII code, and subtracts 48 from the code number. So if the 1 key is pressed, line 100 will convert 1 to produce the code number 49, subtract 48 which leaves us with what we want – the number 1.

The program is then free to check if the value input is acceptable, i.e. not less than one or greater than 4. If the value was incorrect then a beep is heard, and we see another new feature of the BASIC. The cursor is moved from wherever it was last to a new position given by the LOCATE command. In this case, the cursor is put at the end of the "SELECT OPTION NUMBER" line, ready for the corrected data to be put in. LOCATE 22,20 means "put the cursor at the 22nd character position on the 20th line. In text mode 0, the highest character position can be 40, as this is the maximum width of the screen, in text mode 1, it can only be a maximum of 36. You can locate the cursor in high resolution graphics mode too, the main difference here being that the screen is treated as being 256 characters wide and having a maximum number of 192 lines.

Eventually the correct data will be entered, and the program will then have to decide what the user wants. This is where the ON. . .GOSUB comes in. This is like having a whole string of IF. . .THEN statements all in one line. The BASIC looks at the condition specified, in this case the option the user chose, and jumps to another section of the program, the one being jumped to depending on whether the condition is 1, 2, 3 or 4. Instead of using line 130, the following piece of code could have been used equally well:

```
130 IF OPT%=1 THEN GOSUB 500
132 IF OPT%=2 THEN GOSUB 600
134 IF OPT%=3 THEN GOSUB 700
138 IF OPT%=4 THEN GOSUB 800
```

After the computer has done what the user wants, it waits for a key to be pressed, before setting up the main menu again. This is done by polling the keyboard using INKEY$. When INKEY$ has a value, the program returns to the beginning again, and the process can continue over until the user selects option number 4, which ends the program.

Using a Joystick

An alternative way to use a menu-driven program is to use a joystick to move the cursor around the screen to point to an option. When an option has been pointed to, the space bar is pressed, selecting that option. A simple program can be devised which just prints up a list of words on the screen, allows the user to move the cursor around the screen, place the cursor next to the option desired, and do something when the space bar is pressed. There are two new functions, a special variable and a new statement introduced here, all of them quite useful. To find out how they work, type in this new menu-driven program:

```
10 REM MENU SYSTEM 2
20 REM SET UP SCREEN MODE
30 SCREEN 1:KEY OFF
40 REM INITIALISE X AND Y
50 X=1:Y=1
60 REM PRINT OUT MENU
70 LOCATE 18,12:PRINT "EXIT"
80 LOCATE 18,16
90 PRINT "TUNE"
100 REM ACTIVATE SPACE BAR DETECTION
110 STRIG(0) ON
120 REM PUT CURSOR AT LOCATION DETERMI
NED BY X AND Y
130 LOCATE X,Y,1
135 REM PRINT "X"
140 ON STRIG GOSUB 300:REM SEE IF SPACE
BAR PRESSED
150 REM WORK OUT THE DIRECTION OF JOYS
TICK'S POINTING (IF STICK(0)=3 THEN 200,
ETC.)
```

```
160 A=STICK(0)
170 ON A GOTO 190,195,200,205,210,215,22
0,225
180 GOTO 160
190 Y=Y-1         :GOTO 240:REM UP
195 X=X+1:Y=Y-1:GOTO 240:REM UP AND RIGH
T
200 X=X+1         :GOTO 240:REM RIGHT
205 X=X+1:Y=Y+1:GOTO 240:REM DOWN AND RI
GHT
210 Y=Y+1         :GOTO 240:REM DOWN
215 X=X-1:Y=Y+1:GOTO 240:REM DOWN AND LE
FT
220 X=X-1         :GOTO 240:REM LEFT
225 X=X-1:Y=Y-1:GOTO 240:REM UP AND LEFT
240 REM CHECK THE COORDINATES ARE VALID.
 IF THEY ARE NOT, RESET THEIR VALUE TO A
N ACCEPTABLE NUMBER
250 IF X>40 THEN X=40
260 IF X<1 THEN X=1
270 IF Y>24 THEN Y=24
280 IF Y<1 THEN Y=1
290 GOTO 130:REM LOCATE CURSOR AT NEW PO
SITION
300 REM SUBROUTINE - VALIDATES OPTIONS A
ND CARRIES THEM OUT
310 IF CSRLIN=12 THEN GOTO 500
320 IF CSRLIN=16 THEN PLAY "CDEFGGFEDC":
RETURN
330 REM SIGNAL ERROR WITH A BEP
340 BEEP: RETURN
500 CLS:PRINT "END":END
```

The statement STRIG(0) ON tells the computer to look out for joystick trigger button (0), being pressed. If there is no joystick attached, the space bar is treated as the trigger button. The later statement, ON STRIG GOSUB, tells the computer what to do when the space bar or trigger button is pressed. When the subroutine has dealt with what you wanted it to do when a trigger

button was pressed, it executes another STRIG(0) ON all by itslef, on return from the subroutine. You can also use the statements STRIG(0) OFF and STRIG(0) STOP. The former statement tells the computer that there's no need to look out for a button being pressed, the latter statement tells the computer to take note of the fact that a button has been pressed but not to do anything. The computer will immediately jump to the subroutine which knows what to do when triggers are pressed when a STRIG(0) ON statement is executed if a trigger had been pressed earlier.

The STICK function looks to see in which direction the joystick is being moved. If the value returned by STICK(0) is 0 then the joystick isn't being moved at all. If you don't have a joystick attached to your computer, then the cursor keys may be used. To get the directions of diagonally up and right for example, the Up and Right cursor keys are pressed simultaneously, and so forth. STICK(0) defaults to the cursor keys, unless you plug a joystick into the computer.

POS is a function which returns a number indicating the cursor's current horizontal (column) position on the screen. You can supply this function with any argument at all. As the argument has no bearing on what information you receive back from the function, it is known as a **dummy argument**. CSRLIN is a special variable which keeps track of the cursor's current vertical (row) position on the screen.

Using Interrupts

Interrupts are a class of very useful instructions that are far and few between in most dialects of BASIC. An interrupt, as its name suggests, is a break in the normal flow of a program which is caused by some event. An interrupt is used to **trap** an event – note the occurrence of some event and then do something about it.

This method differs from **polling**, which we used earlier to see if a key had been pressed. To illustrate the differences, here are two programs, both of which are designed to play a note when the spacebar is pressed.

```
10 REM POLLING METHOD
20 PRINT "POLLING"
```

```
30 GOSUB 90
40 PRINT "IS LESS"
50 GOSUB 90
60 PRINT "EFFICIENT"
70 GOSUB 90
80 GOTO 20
90 REM LOOK TO SEE IF SPACEBAR HAS BEEN
PRESSED
100 A$=INKEY$
110 IF A$=CHR$(32) THEN PLAY "C":RETURN
120 RETURN
```

Now the same problem solved using interrupts:

```
10 REM INTERRUPT METHOD
20 STRIG(0) ON
30 ON STRIG GOSUB 80
40 PRINT "INTERRUPTS"
50 PRINT "ARE MORE"
60 PRINT "EFFICIENT"
70 GOTO 40
80 PLAY "CDEFG":RETURN
```

With the polling method, a section of code that checks to see if the spacebar has been pressed has to be written. In the second program, the STRIG(O) ON command tells the computer to look out for the spacebar being pressed, and the ON STRIG GOSUB statement determines which line number the program should branch to when the spacebar has been pressed.

In order to explain the differences between polling and interrupt systems, consider the following little scenario. Imagine that you are cooking a cake in the oven, and at the same time, writing a letter to a friend. If you were behaving like a polling system, you would go over to the cooker every five or ten minutes or so to see if the cake were ready yet, then return to writing the letter. An interrupt system would be provided if your cooker had a timer and alarm bell. You could continue writing your letter without worrying about how the cake was getting along, until the cooker alarm bell sounded, telling you to take the cake out of the oven.

In MSX-BASIC, there are a wide range of interrupts available. They trap events like function key depression, pressing of the STOP key, errors, sprite collision, and so on. To enable the interrupts, there are two commands – <interrupt> ON (that we've just seen) and <interrupt> STOP. What <interrupt> STOP will do is remember the occurrence of an event, but disable a later ON <interrupt> GOSUB statement. This allows you to turn off the event trapping for a time, without missing out on dealing with an event if it has occurred. Executing a <interval> ON command will cause a program to branch if an interrupt has been remembered.

A word of caution about using these interrupt commands. When an ON ERROR statement has been executed, *all* other interrupts are disabled. So if you use the ON ERROR statement with other interrupts, bear in mind that you will have to turn on all the other interrupts again in the interrupt handling routine.

Using these interrupt instructions allows you to write more concise code. Their main disadvantage is that they may make programs slightly more difficult to trace and debug, so their use must be well commented.

Data Output

In previous chapters we looked at the PRINT command and how it can be used wth colons and semicolons to format printouts on the screen. One command we then said we'd cover in more detail was the PRINT USING command.

PRINT USING is an altogether different kettle of fish to PRINT. With this statement you can specify an output to be formatted in a certain way. To do this, there are a set of special formatting characters involved. For outputting character data there are three formatting characters, !, &, and a. First the exclamation mark. See what happens when you run the following program:

```
10 INPUT A$
20 PRINT USING "!";A$
```

Only the first character of the string is printed out. This would be particularly useful, for example, when printing out a list of names. The above program could be used to cut first names down to

initials. The following program takes names as input and produces the formatted output of initial and surname:

```
10 INPUT "FIRST NAME ";C$
20 INPUT "SURNAME ";S$
30 PRINT USING "!";C$;
40 PRINT ". ";S$
50 GOTO 10
```

If the following data were entered: "GRAHAM BLAND", "FRED BLOGGS", "ERIC SPROTE", the output from the program would be:

G. BLAND
F. BLOGGS
E. SPROTE

The second formatting character for strings is the & sign. You specify this formatter rather differently, thus:

```
10 PRINT USING "&&";"FRED";"ERIC"
20 PRINT USING "&   &";"FRED";"ERIC"
```

The output will appear like this:

FRED
FRED ERIC

The rule for this formatter is that it prints out at least two characters from a string, plus as many characters as there are spaces between the two &s. So in the first example, only the first two characters were produced from both "FRED" and "ERIC" resulting in "FRER". In the second example, there were three spaces between the &s resulting in the other two characters from "ERIC" and "FRED" being printed plus one space. If we had specified the following format:

20 PRINT USING "& &";"FRED";"ERIC"

We see that the string "FREERI" is produced. The final character formatter is the a symbol. This specified that a character string should be printed out exactly as it is. For example, the

following program takes a name as input, and puts it inot a message which is printed out:

```
10 INPUT "WHAT IS YOUR NAME ";N$
20 PRINT USING "HELLO @, HOW DO YOU DO";
N$
```

As a result, if you type in "FRED BLOGGS", the computer will respond with:

"HELLO FRED BLOGGS HOW DO YOU DO".

There are also special formatters for numbers. These are #, +, –, **, ¥¥, **¥, and ^^^^. Some of these are of little use, especially the one which will print out the Yen signs. However, here's a brief look at what some of them can do.

The # symbol denotes a number. If you want to print out a series of numbers in a very neat manner, you would use this formatting character. As an example, type in the following:

10 PRINT USING "###. ##";1.646,134.5,.45,6.91

The program will produce the following output:

1.65 134.50 0.45 6.91

The numbers are rounded up if necessary with this formatting character. If a decimal point is specified, and the number is a fraction, a 0 will always be put in front of the decimal point. Note also that spaces are inserted in front of the number if it is shorter than that given by the PRINT USING statement.

The plus and minus signs will cause a + or – symbol to be placed before or after the number:

```
10 PRINT USING "+###.##";12.86,-12.86
20 PRINT USING "###.##+";12.86,-12.86
30 PRINT USING "###.##-";12.86,-12.86
```

The output produced will look like this:

+12.86 –12.86

12.86   12.86 –
12.86   12.86 –

The "**" formatter fills in the leading spaces of a number with asterisks:

```
10 PRINT USING "**#.##";12.86
20 PRINT USING "**#.##";1.83
30 PRINT USING "**#.##";123.67
```

The output for the above program will look like this:

*12.86
**1.83
123.67

The final formatting characters insert the Japanese Yen symbol in front of a number. Unless you deal with a lot of Japanese currency, it's not likely you'll want to use these formatters.

There are versions of PRINT USING and PRINT available for use with a printer. Called LPRINT USING, and LPRINT, they are used in exactly the same way, but the output goes to a printer (if attached to your system) instead of appearing on the screen.

Input and Output with Files

A file is basically collection of data. MSX-BASIC provides a number of special instructions and variables especially for use with files. There are four types of device which may be used. Only one of these devices may be used for both the storage and retrieval of files, namely the cassette recorder. The other three devices are output only. The table below outlines the devices available, the way that MSX-BASIC names them (their descriptor) and the way they may be used:

Device Name	Device Descriptor	Input/Output
Cassette Recorder	CAS:	Input and output
Line Printer	LPT:	Output only
TV/Monitor	CRT:	Output only
Graphics Screen	GRP:	Output only

The main device we'll consider is the cassette recorder, as it is

essential for saving programs, and you probably have one in the house anyway.

To use a file in MSX-BASIC, you first have to **open** the file. The following program simply requests three words from the user and stores them to tape:

```
10 OPEN "CAS:FRED" FOR OUTPUT AS #1
20 INPUT A$
30 INPUT B$
40 INPUT C$
50 PRINT #1,A$
60 PRINT #1,B$
70 PRINT #1,C$
80 CLOSE #1
```

Line 10 actually opens a cassette-based file called "FRED" and declares that it is to be used as an output file by the program. The number with the # sign in front of it is the **file number**. The file number # 1 is used to refer to the file "FRED" throughout the program. After line 10 has been executed, you will be prompted to press play and record on the tape recorder.

The output to the file is carried out by the statement in line 30. The PRINT # statement simply writes the character strings you input to tape. This statement is in fact just about the same as a normal PRINT statement. In addition, there is also a PRINT # USING statement available in MSX-BASIC. The last statement of the program closes the file. If you don't specify a file number, then all open files will be closed. The END statement also has this effect. What the program does when it closes a file, is to write a character to the tape which marks the end of the file. This character corresponds to typing the CTRL and Z keys simultaneously.

Now you've saved your character strings to tape, you may wish to retrieve them at a later date. The following program will input character strings from the tape file "FRED".

```
10 OPEN "CAS:FRED" FOR INPUT AS #1
20 IF EOF(1) THEN GOTO 100
30 INPUT #1,A$
40 PRINT A$
```

```
50 GOTO 20
60 CLOSE
```

The OPEN statement is altered so that data will be **input** this time. **EOF** is a special variable in MSX-BASIC. When a file is being read from tape, the presence of the CTRL-Z character is monitored. When the CTRL-Z character is encountered, then the value of the variable EOF becomes −1. EOF is the **End-Of-File** variable. If a program doesn't check for the value of EOF changing to −1, then an 'Input Past End' error will be encountered.

The simple cases we have shown above use two of the three possible file modes. Aside from INPUT and OUTPUT modes, APPEND mode is also available. If you open a file in APPEND mode, then the file will be read from tape until the CTRL-Z character is encountered, then new information may be added onto the end of the file.

The very simple program below is designed to set up a cassette file for names and telephone numbers, and retrieve the phone number for a particular name.

```
10 CLS
20 PRINT "1. SET UP FILE"
30 PRINT "2. RETRIEVE INFORMATION"
40 PRINT:PRINT
50 INPUT "SELECT OPTION (1 OR 2)";A$
55 PRINT "ENTER TAPE COUNTER NO.":INPUT
"NO : ";C
60 IF A$="1" THEN GOTO 100
70 IF A$="2" THEN GOTO 240
80 BEEP:GOTO 50
90 REM SET UP/UPDATE FILE
100 OPEN "CAS:TEL" FOR OUTPUT AS #1
110 CLS
120 PRINT "FILE SET UP"
130 PRINT "-----------"
140 FOR I=1 TO 800:NEXT I
150 CLS
160 PRINT "TO EXIT TYPE * INSTEAD OF NAM
E"
```

```
170 PRINT
180 INPUT "TYPE IN NAME : ";N$
190 IF N$="*" THEN PRINT "SETUP FINISHED
": PRINT"REWIND TAPE TO ";C:GOTO 340
200 INPUT "TYPE IN NO. : ";T$
210 PRINT:INPUT "IS THIS OK? (Y/N) ";R$
220 IF R$="Y" THEN PRINT #1,N$,T$
230 GOTO 150
240 REM INFO RETRIEVAL
250 OPEN "CAS:TEL" FOR INPUT AS #1
260 CLS:PRINT "START TAPE AT ";C
270 INPUT "TYPE IN NAME: ";N$
280 PRINT "SEARCHING FOR ";N$;"'S NO."
290 IF EOF(1) THEN 330
300 INPUT #1,A$,B$
305 PRINT A$,B$
310 IF N$=A$ THEN PRINT "THE NUMBER IS "
;B$:F=1
320 GOTO 290
330 IF F=0 THEN PRINT "NO LISTED N0. FOR
";N$
340 CLOSE
350 END
```

The problem with using cassette based filing systems is that they work incredibly S-L-O-W-L-Y and are fairly inflexible. MSX-DOS provides much faster and much more useful file handling options in the BASIC, taking full advantage of the speed and capacity of floppy disks.

Please refer to Appendix A for other formatting functions such as SPACE$ and TAB.

In the next chapter, we'll look at the more dynamic aspects of programming, how to exploit the musical capabilities of an MSX computer (or how to become *very* unpopular with your family and friends *very* quickly).

MSX music and sound

Sound from a computer can be generated in different ways. Years ago, programmers working on mainframes and minicomputers used to generate music from their line printers in a very devious way. By sending certain character sequences to the printer, they found that they could produce slight changes in pitch which could loosely be called musical notes. Some inventive, if not altogether pleasant tunes, could be played in this unorthodox manner.

With the much later arrival of the home micro, musical notes could be played by sending electrical pulses to a speaker. If the pulses are sent fast enough, then a tone will be produced. In most cases, this can be controlled by BASIC using POKE. This puts numbers into a memory location, which is then used by an assembly language routine to pulse a speaker very quickly and produce recognisable tones.

MSX micros have a more sophisticated means of producing their sounds. There is a chip dedicated to this purpose, which you can access through two statements and one command: SOUND, PLAY, and BEEP. The chip is capable of producing three different sounds simultaneously, with a variety of sounds ranging from the pings, crashes and screeches familiar to those who play arcade games, to musical chords and notes.

You've already met the BEEP command before. Just to remind yourself what it sounds like, try typing BEEP. Thrilling, isn't it? If Beethoven had been stuck with the BEEP command, we'd be sadly lacking in symphonies. The really useful statements are SOUND and PLAY.

The SOUND statement is the harder of the two to learn to use, but it does give you more flexibility in the types of sound you can produce. What the sound statement does is to send numbers directly to the sound chip. This chip has 13 **registers**, all of which you can get at. Each register stores a number which controls a specific function of the chip. These are shown in the table overleaf.

So, depending on which register you send a number to, and the value of that number, certain things will happen. Confused? Well, to help make everything a little clearer, please welcome the **programming model** of the MSX sound chip, the General Instruments' AY-3-8910 (thunderous round of applause please).

REGISTER NO.	FUNCTION
0	Controls the fine pitch tuning for sound channel A. All 8 bits are used so that any number between 0 to 255 can be placed in this register.
1	Controls the course pitch tuning of channel A. There are only four bits used here, so the maximum value you can put into this register is 15.
2	Fine pitch tuning for channel B.
3	Coarse tuning for channel B.
4	Fine pitch tuning for channel C.
5	Coarse pitch tuning for channel C.
6	This modulates the noise channel. 5 bits are used here, so the values that can be placed here range from 0 to 63.
7	This register can change the output from a sound channel from a tone to a noise rather like hissing. Values up to 7 will enable a tone to be produced from the channels. Values above 7 will create noise from the three channels.
8	Sets the volume of channel A.
9	Sets the volume of channel B.
10	Sets the volume of channel C.
11	8-bit fine tune.
12	8-bit coarse tune.
13	Envelope controller.

Sound registers

The programming model of a piece of hardware shows the programmer what each register of the chip controls. This is demonstrated in the program below.

```
10 REM SOUND DEMO
20 REM INITIALISE SOUND CHANNEL A. SET C
OARSE TUNE TO 1 AND FINE TUNE TO 0
30 SOUND 0,1
40 REM START UP THE SOUND FOR CHANNEL A
50 SOUND 8,5
60 END
```

If you run the program, you'll notice two things happen. One is an awful whine coming from the loudspeaker of your television set, and the other is that the "OK" prompt is seen, indicating that your program has finished running. "Ah!", I hear you ask, "Why didn't that awful whine stop when the program did?". The reason it is still driving you potty is because the sound chip has been told to produce that whine by the program, but it hasn't been told to stop making a noise. There are two ways you can get rid of this awful noise. One is to type BEEP – as BEEP uses the sound chip, it resets everything, and turns off the sound.

The other way of removing this potential headache is by typing SOUND 8,0. This turns the volume of channel A down to zero. Now, how did the program produce that horrible sound? By putting the values of 0 and 1 into register 0 and 1 of the sound chip, the program was setting the pitch of sound channel A. By putting the value of 5 into register 8, the program was turning the volume of channel A up. Try the following program and see what happens:

```
10 SOUND 0,1:SOUND 8,5
20 FOR I=1 TO 14:FOR J=1 TO 100:NEXT J
30 SOUND 1,I
40 NEXT I
50 SOUND 8,0
60 END
```

This program produces 14 different pitches. You'll notice that we turned off the sound at the end of the program.

Let's get more adventurous and turn on all the sound channels at once. This will produce a chord:

```
10 SOUND 0,1:SOUND 1,1
20 SOUND 2,1:SOUND 3,3
30 SOUND 4,1:SOUND 5,6
40 FOR I=8 TO 10
50 SOUND I,5
60 NEXT I
```

These are the basic ways of producing musical tones. What is more exciting about SOUND is that you can produce lots of unusual sounds. The simplest way of doing this is by using the fine tune registers.

Sample SOUND Programs

The best way to get to grips with the sound chip is through practice, so we've given you a number of different programs to try. We strongly advise you to make changes to the programs for yourself, to see what effects they have.

This section of the book lapses into strange noises. Warn everybody beforehand. . .

```
10 REM WEIRD NOISES
20 INPUT N
30 SOUND 0,1:SOUND 1,1:SOUND 8,5
40 FOR I=255 TO 0 STEP-N
50 SOUND 0,I
60 NEXT I
70 GOTO 40
```

Try values for N like 5, 15, .5 etc and see what you get.

The noise channel has immense possibilities. Register 7 is used to start the noise channel up. Here are two programs, both of which purport to sound like helicopters.

```
10 SOUND 7,5
20 SOUND 8,7
30 FOR I=63 TO 1 STEP-1
40 SOUND 6,I
50 NEXT I
60 GOTO 30
```

And now another helicopter noise complete with the whine of the engine!

Value of S	**Envelope produced**
0,1,2,3,9	
4,5,6,7,15	
8	
10	
11	
12	
13	
14	

Sound envelopes available from the "S" command

```
10 SOUND 0,20:SOUND 1,0:SOUND 2,30
20 SOUND 3,0:SOUND 4,0:SOUND 5,9
30 SOUND 6,0:SOUND 7,48:SOUND 8,16
40 SOUND 9,4:SOUND 10,6:SOUND 11,100
50 SOUND 12,2:SOUND 13,12
60 GOTO 30
```

This little program uses the envelope capability of the sound generator. An envelope determines the shape of the waveform of the sound generated. There are 8 different envelopes that can be produced. The diagram below shows these waveforms and the value you need to put into register 13 to produce them.

SOUND is one of those statements you have to play with for a while to get used to it. If it's music you're after, the laborious

methods involved with SOUND are too time-consuming, and require you to think in terms of numbers rather than notes. Wouldn't it be easier to ask the computer to play a note such as C or E?

The Music Macro Language

The PLAY command allows you to do just that and quite a bit more. You can specify notes to be played, their lengths, waveforms and volume. All the instructions to the sound chip are presented as character strings. As a simple example, the following statement will produce a chord.

```
PLAY "C", "A", "E"
```

Channel A will play the note C, channel B will play A, and channel C will play E. The pitch of the notes produced can range over 8 octaves from C to C. The character strings contain instructions which are recognised by the PLAY command. This constitutes a small language within BASIC, and is known as the **Music Macro Language**. The instructions for the Music Macro Language (or **MML**) are summarised below.

MML Command	Function
A to G	Plays the indicated note in the current octave. The options #, + and – are also allowed, where # or + indicates a sharp, and – indicates a flat. These options are only allowed, however, when they correspond to a black key on the piano. B ^ is therefore an invalid note.
O<n>	Sets the current octave, where n can be from 1 to 8. EAch octave goes from C to B, and octave 4 is the octave that the MML will use unless you use the O instruction.
N<n>	Provides an alternative to specifying notes and octaves. n can be between 0 and 96, where 0 means rest, 1 means the note C of the lowest octave, and so on.

L<n>	Sets the length of the notes which follow it, where the length of the note is l/n, thus: **L1** whole note **L2** half note **L3** one of a triplet of three half notes (1/3 of a four beat measure) **L4** quarter note **L5** one of a quintuplet (1/5 of a measure) **L6** one of a quarter note triplet **L64** sixty-fourth note The length may also follow a note if you only want the one note changed. For example L2A and A2 mean the same thing. A length of 4 is the default.
R<n>	Sets the length of a pause (or rest) where n can be between 1 and 64, and works in exactly the same way as for L. The default is 4. A . after a note causes it to be played as a dotted note, i.e. 3/2 times its normal length. More than one dot can appear after the note, and the length will be adjusted accordingly so that, for example, A. . . will play for 27/8 as long as usual. Dots may also appear after a pause to adjust its length in a similar fashion.
T<n>	Sets the tempo for the tune by specifying the number of quarter notes to be played in a minute, where n can range between 32 and 255. The default is 120.
V<n>	Sets the volume, where n may be in the range 0 to 15, with default 8.
M<n>	Sets the period of envelope, with n ranging from 1 to 65535 and a default of 255.
S<n>	Sets the shape of envelope. n may range between 1 and 15, with a default of 1.
X<string variable>	executes a specified string of notes, pauses etc.

In MML commands, the value of n can be a constant or a variable. If you use variables, you have to assign them in the string of commands. For example try the following short program:

```
10 INPUT X
20 PLAY "o=X;C"
30 GOTO 10
```

Note the semi-colon after the variable name. If you forget to put this in, the basic will give you an "ILLEGAL FUNCTION CALL" error message. The program will play the note C in the octave you supply.

Here is a little melodic piece to twiddle your thumbs to:

```
10 PLAY "L4O5AECO4","L4O3AEBA","L8AECO3A
O4AECO3A"
20 PLAY "L4O5BECO4G+","L4O2G+","L8O3G+O4
CEBBECO3G+"
30 PLAY "L4O6CO5ECO4G","L4O2G","L8O4GO5C
EO6CCO5ECO4G"
40 PLAY "L4O5F+DO4AD","L4O3D","L8O5F+DO4
ADO5F+DO4AD"
50 GOTO 10
```

As you can see, all three voices are brought into play (forgive the pun). The length of the notes are also varied. The following program generates pseudo random notes using the **RND** function.

```
10 REM PSEUDO-RANDOM NOTE GENERATOR
20 PITCH=INT(RND(10)*100)
30 IF (PITCH>96) THEN GOTO 10 ELSE PLAY
"S8M1500L24N=PITCH;":GOTO 20
```

The RND function is like a wheel of fortune. When you use the function, some value between 0 and 1 is produced. I say the program is pseudo-random, because it will produce the same sequence of random numbers each time the program is run. This time, the S and M instructions are used in the PLAY string. S

specifies the type of sound envelope produced, and M specifies how long the envelope will shape the notes played.

The next program uses all three voices to produce random chords. The joystick is used to increase the speed at which the notes play by increasing or decreasing the note length.

```
10 REM PSEUDO-RANDOM NOTE GENERATOR WITH
 THREE VOICES AND SPEED CONTROL
20 REM DEFAULT NOTE LENGTH
30 L=12
40 REM POLL JOYSTICK
50 IF STICK(0)=1 THEN L=L+1:IF L>64 THEN
 L=64
60 IF STICK(0)=5 THEN L=L-1:IF L<1 THEN
L=1
70 REM CALCULATE RANDOM NUMBERS AND OFFS
ETS FOR VOICES TWO AND THREE
80 PIT1=INT(RND(10)*100):IF (PIT1>86) OR
 (PIT1<11) THEN GOTO 80 ELSE PIT2=PIT1-1
0:PIT3=PIT1+10
90 REM PLAY RANDOM CHORD
100 PLAY "S8M1500L=L;N=PIT1;","S8M1500L=
L;N=PIT3;"
110 GOTO 50
```

The final program in this section uses the X instruction. You'll notice that there are three string variables declared at the beginning of the program. These strings are eventually executed using the X instruction.

The X instruction allows music strings to be executed within a PLAY statement. So, if you have sequences of notes which are used repetitively in a piece of music, there is no need to write out the note sequence each time it occurs. For example, assuming that you wish to repeat the following sequence of notes: "O4BGBO5CEG", you would declare assign this string value to a variable, such as A$, and call the name of this string using the X instruction in later PLAY statements. The following short program shows how the X instruction is used.

```
10 A$="O4BG05CEG"
20 PLAY "XA$;"
30 END
```

Note that a semi-colon is required after the string variable name.

```
10 A$="L24o5bgbo6cego5bgbo6cego5bgbo6ceg
"
20 B$="L24o4bgbo5cego4bgbo5cego4bgbo5ceg
"
30 C$="L24o3bgbo5cego4bgbo5cego4bebo4ceg
"
40 FOR I=1 TO 4
50 PLAY "v4XA$;","v4XB$;","v4XC$;"
60 NEXT I
70 FOR I=1 TO 2
80 PLAY "v10L103d","v9L24XA$;","v10L1o2a
"
90 PLAY "o3D",A$,"o2g"
100 PLAY "o3C",A$,"o2e"
110 PLAY "o3c",A$,"o2a"
120 NEXT I
130 FOR I=1 TO 2
140 PLAY "v12L12o3daco2go3ga","v1012o4c"
,"v1012o5c"
150 PLAY "l4o4dg","o3b","o5g"
160 PLAY "L24o4cdefgbo5cdefgb","o3g","o5
g"
170 PLAY "o4bgbo5cego4bgbo5ceg","o3f","o
5f"
180 PLAY "sm1500o3ccccccco5cccccc","o4c",
"o5c"
190 PLAY "l4o4dg","o3b","l2o5d"
200 PLAY "l12cdefgb","l8o6dco5af","o5g"
210 PLAY "o4bgbo5ceg","sm15000l24o7ccccg
gggdddd","f"
220 NEXT I
230 PLAY "L24o4cdefgbo5cdefgb"
240 PLAY "L24o3cdefgbo4cdefgb"
250 PLAY "L24o2cdefgbo3cdefgb"
```

```
260 PLAY "v10l12o3g","v10l12o4b","v10l12
o4d"
270 PLAY "o3a","o4e","o5c"
280 END
```

The following program plays our attempt at the introduction to the ''Hallelujah Chorus''. In some note sequences (lines 120–190), you will be able to hear an echo.

```
10 A$ = "R8L40T255DR24DR24T120L8ER24D
20 B$ = "T120R18L12DR24ER24D"
30 C$ = "R8L6O3GR18AR18BO4L6CR24O3GO4C"
40 PLAY "T120L6G.","T120L6G.","T120L6G."
50 PLAY B$,"R18XB$;","R16XB$;"
60 PLAY A$,"R18XA$;",A$
70 PLAY A$,"R18XA$;",A$
80 PLAY "R6","R6","R6"
90 PLAY "T120L6G.","T120L6G.","T120L6G."
100 PLAY B$,"R18XB$;","R16XB$;"
110 PLAY A$,"R18XA$;",A$
120 PLAY A$,"R18XA$;",A$
130 PLAY C$,"R18XC$;",C$
140 PLAY A$,"R18XA$;",A$
150 PLAY A$,"R18XA$;",A$
160 PLAY C$,"R18XC$;",C$
170 PLAY A$,"R18XA$;",A$
180 PLAY A$,"R18XA$;",A$
```

The following music piece is a simple 12-bar blues chord sequence based on the key of C:

```
10 FOR I=1 TO 4
20 PLAY "L12O3D","O4F+","O4A'
30 PLAY "O3D","O4G","O4B"
40 PLAY "O3D","O4A","O4C"
50 PLAY "O3D","O4G","O4B"
60 NEXT I
70 FOR I=1 TO 2
80 PLAY "O3G","O4B","O5D"
90 PLAY "O3G","O5C","O5E"
```

```
100 PLAY "O3G","O5D","O5F"
110 PLAY "O3G","O5C","O5E"
120 NEXT I
130 FOR I=1 TO 2
140 PLAY "O3D","O4F+","O4A"
150 PLAY "O3D","O4G","O4B"
160 PLAY "O3D","O4A","O4C"
170 PLAY "O3D","O4G","O4B"
180 NEXT I
190 FOR I=1 TO 2
200 FOR J=1 TO 4
210 PLAY "L12O5EC+O4A","O6E","O5C+"
220 NEXT J
230 FOR J=1 TO 4
240 PLAY "O5DO4BG","O6D","O5B"
250 NEXT J
260 NEXT I
270 PLAY "O3D","O4F+","O4A"
280 PLAY "O3D","O4G","O4B"
290 PLAY "O3D","O4A","O4C"
300 PLAY "O3D","O4G","O4B"
```

The appeal of the MML lies in its similarity to normal musical notation. Those of you who have a musical training will get to grips with the language very quickly, and those of you who don't will find music making comes to you very easily.

Graphics using MSX-BASIC

One of the more attractive qualities of a computer is its ability to draw pictures. One picture drawing method beloved of programmers everywhere is to produce a file containing text laid out so that it resembles a picture. Many pictures of Snoopy the dog have been produced in computer installations throughout the world. Using the printer to overprint, different shadings can be produced. The most famous example of this form of computer art must be the version of Leonardo Da Vinci's *Mona Lisa*. If you have a printer you can experiment with this form of computer drawing. It does have the disadvantage of wearing out your printer though! There is a much simpler way of producing pictures.

All MSX computers have a chip dedicated to producing graphics images on the screen. This can be accessed in a number of ways. One method is to send data directly to the video display processor using VPOKE (see Chapter 2). Generally this is a time-consuming process and definitely not for the weak at heart. A much more amenable way to produce pictures on the screen is by using MSX-BASIC's built-in graphics commands.

As you'll already know, there are two graphics modes: low and high resolution. The one we'll be using most in this chapter is the high resolution screen: Screen 2. This screen is made up of a total of 49152 tiny points called pixels. These are arranged to form a grid 256 pixels across by 192 down. Initially, all the pixels are set to one colour (the background colour) when you enter the high resolution screen mode.

Graphics Using PSET and PRESET

The simplest graphics commands are PSET and PRESET. PSET turns a pixel on, PRESET turns a pixel off. You can specify the colour of the pixel by using the colour option of the PSET and PRESET commands. The short program listed below turns all the pixels on.

```
10 SCREEN 2
20 FOR I=0 TO 256
30 FOR J=0 TO 192
40 PSET(I,J),15
50 NEXT J
60 NEXT I
```

Each pixel's colour will be white (colour 15). You'll notice that a pixel's position on the screen is given by two coordinates. Using PSET and PRESET, we can select individual coordinates to turn on, and in so doing draw lines and shapes as we want. An easy way to select which pixels we want to turn on, is by using the joystick. The program below demonstrates how individual pixels may be switched on, using the joystick in effect, as a pencil on the screen.

```
10 REM DOODLE PROGRAM
20 SCREEN 2
30 REM SET INITIAL COORINATES
40 X=128 : Y = 96
50 STRIG(0) ON
60 D = 1 : E = 0
70 ON STRIG GOSUB 200
80 IF D = 1 THEN PSET (X,Y),15 ELSE LOCA
TE X,Y
90 REM POLL JOYSTICK
100 IF STICK(0)= 1 THEN Y=Y-1
105 REM 6-158
110 IF STICK(0)=2 THEN Y=Y-1:X=X+1
120 IF STICK(0)=3 THEN X=X+1
130 IF STICK(0)=4 THEN X=X+1:Y=Y+1
140 IF STICK(0)=5 THEN Y=Y+1
150 IF STICK(0)=6 THEN Y=Y+1:X=X-1
160 IF STICK(0)=7 THEN X=X-1
170 IF STICK(0)=8 THEN X=X-1:Y=Y-1
180 GOTO 80
200 REM REVERSE VALUE OF D
210 SWAP D,E : RETURN
```

The variable D allows the program user to move about the screen without drawing anything. When the space bar is pressed the value of D is changed from either 0 to 1 or 1 to 0, using the SWAP statement. If the current value of D is 1 then pixels are lit up; if it is 0 no pixels are lit up but the cursor position is moved.

Having demonstrated PSET and PRESET, we are now in a position to construct our first 'animation' program. We will briefly look at how motion can be simulated on the screen using these commands.

The principle behind all animation is to show a picture, change it in some way and show the new picture – all so fast that the eye believes it is seeing movement. Using PSET and PRESET we can simulate movement in the same way.

Firstly, a point on the screen is set (say PSET (10,10)). Then an adjacent point is set in the same way, while the original point is turned off (using PRESET). What happens in a program therefore, is that the process of turning a point on and turning its partner off, is repeated for as long as movement is required. The continuation of movement is supplied through a FOR. . .NEXT loop.

```
10 SCREEN 2
20 FOR I=10 TO 250
30 PRESET (I-1,100)
40 PSET (I,100)
50 NEXT I
60 FOR I=250 TO 10 STEP -1
70 PRESET (I+1,100)
80 PSET (I,100)
90 NEXT I
100 GOTO 20
```

This will send a point across your screen from left to right and back again. Notice how the left and right motions are described in different FOR. . .NEXT loops. Look also at the PRESET syntax. In moving from left to right, it is the point behind the PSET point that needs to be turned off – hence PRESET is for I-1,100.

As with all programs, try experimenting. For example, try and make the computer beep when the point changes direction. Try and make the point move onto a different line each time it changes

direction (you may have to introduce a new variable here!)

LINE is another useful graphics statement. With LINE, you specify a pair of start coordinates and a pair of destination coordinates, and a line will be drawn to link the two together, for example:

```
10 SCREEN 2
20 LINE(50,50)-(100,50),15
30 LINE(100,50)-(100,100),15
40 LINE(100,100)-(50,100),15
50 LINE(50,100)-(50,50),15
60 GOTO 60
```

The above program will draw a white box. There is an easier way of drawing a box using LINE which only requires one LINE statement. As you saw before, if you specify the top left-hand coordinate of the box as the start coordinate, and the bottom right-hand corner of the box as the destination coordinate, you would expect a diagonal line to be drawn. This is normally true, unless you use the B option on the LINE statement (see Chapter 2). Type in and run this program:

```
10 SCREEN 2
20 LINE(20,20)-(40,40),,B
30 GOTO 30
```

The B at the end of the LINE statement indicates that you want a box to be drawn instead of a straight line. LINE will also allow you to colour the box in once it is drawn. Instead of using B at the end of a LINE statement, use BF. This instructs the computer to draw the box then "paint" the box.

The listing below is a program which draws boxes of random sizes and paints them in random colours.

```
10 SCREEN 2
20 DEFINT A-Z
30 V=INT(RND(1)*256):W=INT(RND(1)*192)
40 X=INT(RND(1)*192):Y=INT(RND(1)*192)
50 C=INT(RND(1)*15)
```

```
60 LINE(V,W)-(X,Y),C,BF
70 GOTO 30
```

So much for box-like objects. We can also draw circles, as you saw earlier. Here are some programs which draw some interesting patterns using the CIRCLE statement.

```
10 REM WALLPAPER
20 REM INPUT VALUES
30 INPUT "X SPACING ";X
40 INPUT "Y SPACING ";Y
50 INPUT "RADIUS DECREMENT ";Z
55 SCREEN 2
60 FOR I=1 TO 256 STEP X
70 FOR J=1 TO 192 STEP Y
80 FOR K=100 TO 5 STEP -Z
90 CIRCLE(I,J),K,15
100 NEXT K
110 NEXT J
120 NEXT I
130 GOTO 130
```

For starters, try inputting 50, 30 and 10. You'll get some nice wallpaper effects with the program. Here's another doodle:

```
10 REM CIRCLE DOODLE
20 SCREEN 2
30 FOR I=1 TO 192
40 CIRCLE(128,I),I,15
50 NEXT I
60 GOTO 60
```

You're not limited to circles of course. You can specify an ellipse using the aspect ratio option of CIRCLE (again as you saw before). You specify the ratio of the circle's height to its width. If for example, you choose the number 1/3, the result will be a flattened ellipse, and if you choose a value of 3/2 the ellipse will be taller than it is wide. The program below demonstrates CIRCLE being used with different aspect ratios:

```
10 SCREEN 2
20 FOR I=1 TO 32
30 CIRCLE(128,96),70,15,,,I/8
40 NEXT I
50 GOTO 50
```

The final thing you can do with CIRCLE is draw an arc. What you need to give the CIRCLE statement is a start angle and an end angle in radians for the arc. Try out this program:

```
10 SCREEN 2
20 CIRCLE(128,92),70,15,5,4
30 CIRCLE(128,92),50,15,4,3
40 CIRCLE(128,96),40,15,3,2
50 GOTO 50
```

What you'll see are three incomplete circles. You'll notice that they're incomplete in different places, so to speak. This is because the start and end angles were different in each case.

When you've drawn a circle, ellipse, or box, you can colour it in using the PAINT statement. The only rule here is that in high resolution mode, when painting a shape, the colour used *must* be the same colour that was used to draw the shape in the first place. This program draws a circle and paints it black.

```
10 SCREEN 2
20 CIRCLE(128,96),70,1
30 PAINT(128,96),1
40 GOTO 40
50 GOTO 50
```

Try painting the circle in white instead of black. You should see the whole screen being obliterated by white. This is because the computer expected to stop painting when it reached a white boundary. In low resolution mode, PAINT will paint in any shape, regardless of what colour its border is.

On the subject of colour, you can find out what colour a specific pixel is on the screen by use of the POINT function. This function returns the colour number of a pixel given by screen coordinates.

POINT is useful when programming simple games. For example, you could use it to see if a missile had landed on a target. All the targets could be painted in a unique colour, black for example. When the missile's coordinates matched those of a black pixel, then the missile has hit a target. Here's a simple demonstration of POINT in action:

```
10 SCREEN 2
20 CIRCLE(50,50),25,15
30 CIRCLE(100,100),25,8
40 CIRCLE(150,150),25,1
50 PAINT(50,50),15:REM WHITE
60 PAINT(100,100),8:REM RED
70 PAINT(150,150),1:REM BLACK
80 FOR I=1 TO 5000:NEXT I:REM DELAY FOR
A WHILE
90 X=POINT(50,50)
100 Y=POINT(100,100)
110 Z=POINT(150,150)
120 SCREEN 1
130 PRINT X,Y,Z
```

The numbers 15, 8, and 1 will be displayed on the screen.

Sprites

Sprites are impish little graphics shapes which you can define and move about on the two graphics screens. They seem to exist totally independently of all other shapes on the screen drawn by the likes of CIRCLE, LINE, PSET, etc. Sprites, for reasons that will be explained shortly, lend themselves to applications in games programs, where they may define missiles, bombs, spaceships, and all the other curious shapes which are to be found lurking in many arcade games.

First, some sprite-related terminology to get used to. You can view the graphics screens as a tabletop where you can arrange your pictures using the normal graphics commands. The graphics screen can be viewed as a plane or surface where you can draw pictures. This plane is known as **plane zero**. Now imagine that the graphics tabletop has tiers above it – more planes where graphics shapes may

be drawn. The extra planes are available for sprites to use. In effect, if you put a sprite on plane zero, and one on plane one, the sprite on plane one will appear to be above the sprite on plane zero. Clever eh! But what do sprites look like and how do you define them?

Sprites are made up of 8 binary numbers, a set of 64 1s or 0s. If you arrange these numbers as an 8×8 grid, you have your basic sketch pad, as shown in Figure 7, with which to draw sprites.

Now, if you were to put a '1' in one of the grid boxes, this would set that part of the sprite to on, i.e. that element of the sprite would appear on the screen. If you put a zero in the box, then that particular element of the sprite will not appear on the screen. Let's look at this a little more closely with a worked example. The following short program will put a square box into the centre of the high resolution screen. The sprite will appear as a solid box as all the values in the data statements are one – each of the 64 elements that make up the sprite will be 'lit up'.

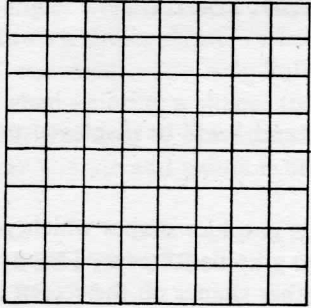


Figure 7 Sprite definition grid

```
10 SCREEN 2,0,0
20 FOR I=1 TO 8
30 READ B$
40 SHAPE$=SHAPE$+CHR$(VAL("&B"+B$))
50 NEXT I
60 SPRITE$(0)=SHAPE$
70 PUT SPRITE 0,(128,96),15,0
80 GOTO 80
90 DATA 11111111
100 DATA 11111111
```

```
110 DATA 11111111
120 DATA 11111111
130 DATA 11111111
140 DATA 11111111
150 DATA 11111111
160 DATA 11111111
```

SPRITE$ in line 60 is a very special MSX-BASIC variable. It is expressly used to define sprites. What the FOR loop is doing, is taking the line of numbers one at a time from the DATA statements, and adding them to the string variable. As you have probably noticed, there is quite a lot of conversion going on in line 40. The **VAL** function encountered here converts characters representing numbers into their numeric equivalent, for example PRINT VAL("1232") would yield the result 1232 which would be a single precision number.

What line 30 does is to read in the numbers in each DATA statement as a character string, then line 40 adds the &B binary number prefix, converts this string to a true binary value, then converts the entire binary number to a character string using the CHR$() function. When the value of the string variable named SHAPE$ has been built up in the FOR statement, it is ready to be assigned to the special SPRITE$ variable. You have to give the sprite a number at this stage. You have 32 options open to you, the numbers 0 to 31. The sprite number is used to reference that particular sprite from that point on. What line 60 does is to assign the reference number 0 to the sprite shape that has just been built up.

Once you've defined your sprite, you'll want to put it on the graphics screen. This is done using the PUT SPRITE statement. Not only does this determine *where* the sprite will appear on the screen, it also defines which sprite will be put on the screen, its colour, and on which plane of the screen the sprite will exist. For the example above, we have said that sprite 0 will be put at the centre of the screen on plane zero, and it will be coloured white. In words, the PUT SPRITE statement may be described thus:

PUT a SPRITE on PLANE NUMBER XX at position (X,Y) with a COLOUR YY, and the sprite used will be SPRITE NUMBER ZZ.

The top left hand corner of a sprite is used to reference where the sprite will appear on the screen. It is this corner of the box which is placed at the centre of the screen, and not the centre of the sprite. This is a fairly useful thing to remember!

An interesting thing happens when you try to put two sprites on the same plane at once. The two sprites will flicker on and off. This is because only one sprite can exist on a plane at any one time. What the video chip does is to put one sprite on a specified plane. If it encounters another sprite that wants to appear on that plane, it erases the original sprite, and replaces it with the new one. The next step is to try to move a sprite around the screen.

We'll use the joystick routine as before to move the sprite. There is one stationary sprite on the screen at the centre of plane 1. You'll see the black cross appear to move *over* the white cross. If you put the black cross on plane 1, and the white cross on plane 0, you will be able to move the black cross under the white one.

If you look at the data statements in the program in just the right light, and in a very blurry manner you'll be able to see the pattern of ones appear as a cross. Now to the program:

```
20 SCREEN 2,0,0
30 FOR I=1 TO 8
40 READ B$
50 SHAPE$=SHAPE$+CHR$(VAL("&B"+B$))
60 NEXT I
70 SPRITE$(0)=SHAPE$
80 SPRITE$(1)=SHAPE$
90 PUT SPRITE 0,(128,96),15,0
100 X=128:Y=92
110 DIR=STICK(0):IF DIR=0 THEN 110
120 GOSUB 150
130 PUT SPRITE 1,(X,Y),1,0
140 GOTO 110
150 IF DIR=1 THEN Y=Y-1
160 IF DIR=2 THEN Y=Y-1:X=X+1
170 IF DIR=3 THEN X=X+1
180 IF DIR=4 THEN X=X+1:Y=Y+1
190 IF DIR=5 THEN Y=Y+1
200 IF DIR=6 THEN Y=Y+1:X=X-1
```

```
210 IF DIR=7 THEN X=X-1
220 IF DIR=8 THEN X=X-1:Y=Y-1
230 IF X>256 THEN X=256
240 IF X<1 THEN X=1
250 IF Y>192 THEN Y=192
260 IF Y<1 THEN Y=1
270 RETURN
280 DATA 10000001
290 DATA 01000010
300 DATA 00100100
310 DATA 00011000
320 DATA 00011000
330 DATA 00100100
340 DATA 01000010
350 DATA 10000001
```

By setting an option in the SCREEN command, you can enlarge sprites to 16×16 pixels. Replace line 10 in the above program to read SCREEN 2,1,0. This will enlarge or magnify the crosses in the program.

You can actually define your own 16×16 sprites without having to magnify 8×8 sprites. First, you have to select a screen mode that will allow you to use 16×16 sprites. The statement that will do the trick is SCREEN 2,2,0. If you're wondering why the keyclick is turned off each time SCREEN is used, it's because you'll probably find that the constant clicking produced when using a joystick extensively will irritate you beyond belief!

To define 16×16 sprites, you need 32 DATA statements. The first 16 DATA statements define the shape of the left half of the sprite, the remaining 16 define the right half.

Otherwise, the setup procedure is handled in much the same way as for 8×8 sprites. You can also magnify these sprites so that they are dimensioned to 32×32 pixels by using the SCREEN command: SCREEN 2,3,0.

Here are some very primitive games to give you some ideas. The first of these uses a 16×16 sprite to represent a spaceship, an 8×8 sprite is used as a missile, and the targets are rows of red boxes. The game will go on until you've hit the targets five times. It's very simple, and provides a reasonable basis to expand on ideas for

games of your own. Some odd little features are included in the program. When the joystick is not being moved in any direction, the spaceship will move about in random directions on its own.

```
10 REM DEMO GAME
20 DEFINT A-Z
25 SCREEN 2,2:COLOR 15,15,15:CLS
30 REM INITIALISE VARIABLE
35 ON STRIG GOSUB 860:REM HANDLE MISSILE
 DROP
40 SC=0:REM NUMBER OF TARGET HITS
50 X=128:Y=10:REM INITIAL SPACESHIP COOR
DINATES
60 REM DRAW BOXES
70 LINE(40,180)-(50,190),4,BF
80 LINE(80,180)-(90,190),4,BF
90 LINE(120,180)-(130,190),4,BF
100 LINE(160,180)-(170,190),4,BF
110 LINE(200,180)-(210,190),4,BF
115 LINE (0,96)-(256,96),1
120 REM SET UP SPACESHIP SPRITE
130 FOR I=1 TO 32
140 READ A$
150 S$=S$+CHR$(VAL("&B"+A$))
160 NEXT I
170 SPRITE$(0)=S$
180 REM SET UP MISSILE SPRITE
190 REM  RESTORE 1260
200 FOR K=1 TO 8
210 READ B$
220 T$=T$+CHR$(VAL(("&B")+B$))
230 NEXT K
240 SPRITE$(1)=T$
250 REM SET UP SPACE BAR POLLNG
260 STRIG(0) ON
270 REM PLACE SPACESHIP POSITION
280 PUT SPRITE 0,(X,Y),1,0
300 REM SEE IF STICK HAS MOVED
310 REM IF IT HAS NOT, CALCULATE RANDOM
DIRECTION
```

```
320 N=STICK(0):IF N=0 THEN N=INT(RND(1)*
8)
330 REM CALCULATE OFFSETS
340 IF N=0 THEN GOTO 280
350 ON N GOSUB 760,770,780,790,800,810,8
20,830
360 REM CHECK THE VALUES OF X AND Y ARE
LEGAL
370 IF X>256 THEN X=256
380 IF X<1 THEN X=1
390 IF Y>104 THEN Y=104
400 IF Y<1 THEN Y=1
410 GOTO 280
420 REM SUBROUTINE TO WORK OUT OFFSETS
760 Y=Y-1:RETURN
770 Y=Y-1:X=X+1:RETURN
780 X=X+1:RETURN
790 X=X+1:Y=Y+1:RETURN
800 Y=Y+1:RETURN
810 Y=Y+1:X=X-1:RETURN
820 X=X-1:RETURN
830 X=X-1:Y=Y-1:RETURN
840 REM SUBROUTINE TO HANDLE TRIGGER BUT
TON DEPRESSION
850 REM DROP BOMB AND SEE IF IT HITS TAR
GET
860 V=Y+1
870 PUT SPRITE 1,(X+4,V),1,1
880 IF POINT((X+12),(V+8))=4 THEN PLAY"S
M1500L2406CDEFG":SC=SC+1:IF SC>4 THEN SC
REEN 1:COLOR 15,4,7:CLS:END ELSE :RETURN
890 IF V=192 THEN PLAY "C","F+","B":RETU
RN
900 V=V+1:GOTO 870
910 REM SPRITE DEFINITION DATA STATEMENT
S
920 REM 1. SPACESHIP
930 DATA 11110000
940 DATA 11110000
950 DATA 11000000
```

```
960 DATA 01000010
970 DATA 01000111
980 DATA 01001010
990 DATA 01111111
1000 DATA 01101011
1010 DATA 01101010
1020 DATA 01111111
1030 DATA 01001010
1040 DATA 01000111
1050 DATA 01000000
1060 DATA 11000000
1070 DATA 11110000
1080 DATA 11110000
1090 DATA 00001111
1100 DATA 00001111
1110 DATA 00000011
1120 DATA 10000010
1130 DATA 11100010
1140 DATA 01010010
1150 DATA 11111110
1160 DATA 01010110
1170 DATA 01010110
1180 DATA 11111110
1190 DATA 01010010
1200 DATA 11100010
1210 DATA 10000010
1220 DATA 00000011
1230 DATA 00001111
1240 DATA 00001111
1250 REM 2. SPRITE DEFINITION OF BOMB
1260 DATA 00011000
1270 DATA 11111111
1280 DATA 11111111
1290 DATA 10000001
1300 DATA 01000010
1310 DATA 00100100
1320 DATA 00111100
1330 DATA 00111100
```

The next games program is a moon-lander game where the object is to keep landing on three white landing pads. Pressing the spacebar will slow down your descent, but will gradually deplete your fuel. There are hazards, of course. You have to avoid stationary and moving "blobs". If you hit one of these you'll blow up. Likewise, if you crash into the rocks, or run out of fuel, you will destroy your ship. When you do land safely, a little man jumps into a box at the bottom of the screen. The program makes use of another ON. . .GOSUB command similar to the ON STRIG GOSUB statement we've already seen. This one allows you to watch out for sprites colliding with each other, in this case, blob sprites colliding with the rocket ship. The command SPRITE ON has to be given before the BASIC is able to detect a sprite collision. SPRITE OFF and SPRITE STOP are like the respective STRIG() ON and STRIG OFF commands. The outline of how to use these statements in a program are given below:

```
10 SPRITE ON:REM START COLLISION DETECTI
ON
.
.
.
70 ON SPRITE GOSUB 100:REM SEE IF SPRITE
S HAVE COLLIDED
.
.
.
100 REM COLLISION HANDLING SUBROUTINE
110 SPRITE OFF:REM TURN OFF COLLISION DE
TECTION
.
.
.
180 SPRITE ON:REM TURN COLLISION DETECTI
ON BACK ON
190 RETURN
```

Here's the program as promised:

```
10 REM LANDER GAME
20 REM
30 SCREEN 2,0,0
40 CLEAR
50 FLAG=1
60 COL=15
70 DIR=4:K=63
80 STRIG(0) ON
90 ON STRIG GOSUB 1500
100 INC=1
110 FUEL=100
120 FFLAG=0
130 K=100
140 F=0
150 ACC=3
160 X=128:Y=10
170 COLOR 15,4,1
180 REM SET UP SPRITES
190 N=0
200 GOSUB 820
210 N=1:RESTORE 1050:OBJ$="":GOSUB 820
220 N=2:RESTORE 1140:OBJ$="":GOSUB 820
230 GOSUB 350
240 X=128:Y=10:PUT SPRITE 0,(X,Y),8,1:TL
=0:SPRITE ON
250 ON SPRITE GOSUB 1570
260 REM poll joystick
270 SOUND 6,K:S=STICK(0):IF S=0 OR S=1 O
R S=8 THEN S=DIR
280 GOSUB 880
290 GOSUB 1230
300 GOSUB 1270
310 GOSUB 1440
320 REM
330 GOTO 270
340 REM DRAW LANDSCAPE
350 LINE(0,160)-(24,184),1
360 LINE(24,184)-(32,144),1
```

```
370 LINE(32,144)-(40,160),1
380 LINE(40,160)-(48,152),1
390 LINE(48,152)-(56,168),1
400 LINE(56,168)-(64,152),1
410 LINE(64,152)-(56,144),1
420 LINE(56,144)-(80,144),1
430 LINE(80,144)-(72,152),1
440 LINE(72,152)-(88,160),1
450 LINE(88,160)-(96,104),1
460 LINE(96,104)-(104,128),1
470 LINE(104,128)-(120,128),1
480 LINE(120,128)-(112,144),1
490 LINE(112,144)-(128,168),1
500 LINE(128,168)-(136,152),1
510 LINE(136,152)-(144,168),1
520 LINE(144,168)-(160,136),1
530 LINE(160,136)-(160,160),1
540 LINE(160,160)-(176,120),1
550 LINE(176,120)-(184,144),1
560 LINE(184,144)-(200,128),1
570 LINE(200,128)-(216,128),1
580 LINE(216,128)-(232,144),1
590 LINE(232,144)-(236,128),1
600 LINE(236,128)-(208,96),1
610 LINE(208,96)-(224,80),1
620 LINE(224,80)-(256,136),1
630 LINE(56,142)-(80,142),15
640 LINE(104,126)-(120,126),15
650 LINE(200,126)-(216,126),15
660 REM PLACE STATIONARY BOULDERS
670 PUT SPRITE 10,(32,56),3,2
680 PUT SPRITE 12,(40,92),3,2
690 PUT SPRITE 13,(148,54),3,2
700 PUT SPRITE 14,(104,50),3,2
710 PUT SPRITE 15,(208,50),3,2
720 PUT SPRITE 16,(112,92),3,2
730 PUT SPRITE 17,(184,90),3,2
740 PUT SPRITE 18,(160,20),3,2
750 CIRCLE (0,0),70,8:PAINT (1,1),8
760 PAINT (128,191),1
```

```
770 LINE(200,184)-(224,192),15,B
780 LINE(208,184)-(208,192),15
790 LINE(216,184)-(216,192),15
800 RETURN
810 REM SPRITE SETUP ROUTINE
820 FOR I=1 TO 8
830 READ A$
840 OBJ$=OBJ$+CHR$(VAL("&B"+A$))
850 NEXT I
860 SPRITE$(N)=OBJ$:RETURN
870 REM JOYSTICK OFFSET CALCULATION
880 IF S=4 THEN 900
890 DIR = S
900 IF S=3 THEN X=X+2:RETURN
910 IF S=4 THEN X=X+1:Y=Y+ACC:RETURN
920 IF S=5 THEN Y=Y+ACC:RETURN
930 IF S=6 THEN Y=Y+ACC:X=X=1:RETURN
940 IF S=7 THEN X=X-1:Y=Y+1:RETURN
950 REM SPACESHIP PATTERN
960 DATA 00011000
970 DATA 00011000
980 DATA 00111100
990 DATA 00111100
1000 DATA 00111100
1010 DATA 11111111
1020 DATA 10000001
1030 DATA 10000001
1040 REM MAN PATTERN
1050 DATA 00000000
1060 DATA 00011000
1070 DATA 00011000
1080 DATA 01111110
1090 DATA 00011000
1100 DATA 00111100
1110 DATA 00111100
1120 DATA 00000000
1130 REM BLOB PATTERN
1140 DATA 00111000
1150 DATA 01111101
1160 DATA 00111110
```

```
1170 DATA 01111111
1180 DATA 11111111
1190 DATA 01111100
1200 DATA 00111110
1210 DATA 00011111
1220 REM RANGE CHECKING AND PLACIN OF SP
ACESHIP AND BLOBS
1230 IF X<0 THEN X=0:RETURN
1240 IF Y<0 THEN Y=0:RETURN
1250 IF X>256 THEN X=256:RETURN
1260 IF Y>192 THEN Y=192:RETURN
1270 REM PLACE MOVING BLOBS
1280 PUT SPRITE 0,(X,Y),8,0
1290 PUT SPRITE 2,(F,80),3,2:F=F+1
1300 PUT SPRITE 3,(F+100,100),3,2
1310 PUT SPRITE 4,(F+10,40),3,2
1320 PUT SPRITE 5,(F+60,128),3,2
1330 PUT SPRITE 6,(F+112,116),3,2
1340 PUT SPRITE 7,(F+224,124),3,2
1350 PUT SPRITE 8,(F+72,72),3,2
1360 IF FLAG=1 THEN RETURN
1370 GOTO 1400
1380 PUT SPRITE 1,(X,Y+4),15,1
1390 PUT SPRITE 1,(X,Y+4),0,1
1400 FUEL=FUEL-2:K=K-4:IF K<0 THEN K = 6
3
1410 IF FUEL=0 THEN GOTO 1570
1420 REM RETURN
1430 REM SEE IF THE SHIP HAS LANDED YET
1440 IF (POINT(X,Y+9)=15) AND POINT(X+9,
Y+9)=15 THEN BEEP:PLAY"L20O6CDEFG":X=128
:Y=10:FFLAG=0:FLAG=1:FUEL=100:SC=SC+1
1450 IF SC=1 THEN PUT SPRITE 23,(200,184
),15,1
1460 IF SC=2 THEN PUT SPRITE 23,(208,182
),15,1
1470 IF SC=3 THEN PUT SPRITE 23,(216,184
),15,1:SC=0
1480 IF POINT(X,Y+9)=1 OR POINT(X+9,Y+9)
=1 THEN GOTO 1570
```

```
1490 RETURN 330
1500 SWAP FLAG,FFLAG:SOUND 7,5:SOUND 8,7
1510 ACC=1:IF FFLAG=0 THEN PUT SPRITE 1,
(0,0),0,1
1520 IF FFLAG=0 THEN ACC=3:SOUND 8,0
1530 RETURN
1540 END
1550 REM COLLISION ROUTINE
1560 TL=1
1570 BEEP
1580 FOR I=1 TO 4
1590 PUT SPRITE 0,(X,Y),1,0
1600 PUT SPRITE 0,(X,Y),15,0
1610 FOR J=1 TO 255 STEP 5:SOUND 8,9:SOU
ND 0,J:SOUND 8,0:NEXT J
1620 NEXT
1630 IF TL=1 THEN GOTO 240
1640 X=128:Y=10:FUEL=100:FLAG=1:FFLAG=0
1650 RETURN
```

Draw

The DRAW command is used to draw figures using the Graphics Macro Language (or GML). This works in the same way as the MML we saw in the previous chapter. Drawing instructions are presented as character strings containing GML commands.

Try the following example:

```
10 SCREEN 2
20 DRAW "R100D100L100U100"
30 GOTO 30
```

As you can see, this program drives a box on the screen. Where the box is positioned depends upon whether or not you have used any of the graphics modes before. If you enter this program just after turning on the machine, the box will be drawn at the top left-hand corner of the screen. However, if you have used any of the graphics screens before, then the top, left hand corner of the box will be positioned at the last pixel addressed.

What the computer has done in the DRAW command is to draw

a series of lines; first right 100 pixels then down 100 pixels and then back 100 pixels, thereby ending up back where the line started and producing the box – it's really as simple as that! Now try replacing line 20 with:

```
20 DRAW "D100F100E100H100G100"
```

This first draws a vertical line down 100 dots, and then diagonally down-and-right, up-and-right, up-and-left, and finally down-and-left, using four more elements of the GML notation, namely F,E,H and G. The full list is as follows:

U<n> Move up
D<n> Move down
L<n> Move left
R<n> Move right
E<n> Move diagonally up and right
F<n> Move diagonally down and right
G<n> Move diagonally down and left
H<n> Move diagonally up and left

In each of the above commands, movement begins at the last point that was referenced. This means that movement starts at the top left-hand corner of the screen, and thereafter from the point at which the previous command stopped drawing. <n> indicates the distance to be moved, so that R100 means move right 100 pixels, F100 means move down 100 pixels and right 100 pixels, and so on.

One way of starting to draw at a point other than the top left-hand corner is by using:

M<x,y>

where x is a horizontal movement and y is a vertical movement. Try changing line 20 in the program again to read:

```
20 DRAW "M100,10R100D100L100U100"
```

As you can see, the point at which drawing starts has been moved 100 pixels across the screen and 10 pixels down it.

Any of the above commands can be prefixed with either B or N.

B moves but doesn't plot any points, N moves but returns to the point that it started from when it has finished. The other commands available are as follows:

A<n> allows you to set the angle at which a line is to be drawn. n can be between 0 and 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270 degrees.

```
    0
1-- + --3
    2
```

C<n> sets the colour of the line to be drawn, and in line with the COLOR command can be in the range 0 to 15.

S<n> sets a scale factor which may be used to modify the distance moved using the U,D,L,R,E,F,G,H and M commands.

X<string variable> allows you to execute a draw command from within another DRAW command. This can be very useful in that, for example, you can define part of the object that you're drawing as being separate from its whole, or can execute DRAW commands that are more than 255 characters long.

The Paintbox program

The next program is named paintbox and it is a simple general purpose drawing program. The graphics screen is split up into two areas: a drawing area and a colour selection area. A joystick (or the cursor keys) is used to control a cursor.

The colour selection area contains the paintbox which gives the program its name. It is a palette of fourteen colours, any one of which may be selected by positioning the cursor over the desired colour and pressing the spacebar. The colours not available from the palette are colours 0 – transparent, and 15 – white.

The drawing instruments you may use are a paintbrush, a

straight edge (to draw lines), a pair of compasses (to draw circles), a box shape (to draw boxes) and an eraser. There are also four, textured 'fillers' to choose from – an aerosol can, vertical lines, horizontal lines and a stipple pattern, as well as the standard PAINT option. The cursor will change to the shape of the selected drawing instrument (or texture pattern) when it is moved within the boundaries of the drawing area. As a point of interest, a term for these various cursor shapes is an **icon** – the cursor shape represents the currently selected function.

Various drawing functions (20 in all) are selected at the keyboard. The full range of command letters is given below (note that they are all upper case). Wherever possible, a command letter has been chosen to match the command's function – although after a while we ran out of letters!

Using the paintbrush

D selects the paintbrush and puts it 'down'. Moving the down cursor then draws a painted line. Initially, the paintbrush will draw a very narrow line. Pressing D once more will allow painting across the whole width of the paintbrush.

U picks 'up' the paintbrush so that you can move it without leaving a paint trail.

Using the aerosol

A selects the aerosol can (spray paint!)

SPACE BAR makes the aerosol spray paint.

Using the compass to draw a circle

C selects the compass and defines the centre of a circle to be drawn at current cursor position.

R defines the radius of the circle as the line from where the cursor was when you pressed C to the current cursor position – the circle is then drawn.

Using the straight edge to draw a line

L selects the straight edge and defines the current cursor position as the starting point for a line.

E defines the end point for a line – and draws the line.

Using the box to draw a box

B selects the box and defines the current cursor position as the start point for one corner of a box to be drawn.
X defines the diagonally opposite corner of the box, and draws the box.

Painting a box or other enclosed area

P paints the whole of the enclosed area within which the cursor has been put. Note that if there is no enclosed area the whole screen will be painted in the current colour.

Local shading using squares of texture/pattern

Pressing one of these commands selects a small square textured box. Pressing the space bar draws the box at the current cursor position in the selected texture/pattern.
V vertical lines – selects vertical line fill-in pattern.
Y selects horizontal line fill-in pattern.
K selects stipple texture fill-in pattern.

Using the eraser

W selects eraser. Moving the cursor then erases (by painting in white pixels).

Other commands

Q selects colour white (while in drawing area).
T toggles cursor – switches the cursor colour from white to black or vice versa.
H home – jumps to colour selection area of the screen.
S screen – jumps to the centre of the drawing section of the screen.
Z zero option! – erases the drawing area of the screen to white. To avoid inadvertently destroying your masterpiece, the Z command has to be used twice before it has any effect.

F exits from the program. Like the Z command, this also has to be used twice before the program is ended.

This program makes full use of sprites to define various cursor shapes, a total of ten in all. In all cases, the reference point for all the cursor shapes is the top left-hand corner – the start point for a box will be defined from where the top left-hand corner of the cursor is positioned.

One user-defined function is used to calculate the radius of a circle using Pythagoras' Theorem.

When the cursor is moved into the colour selection area, it changes into an arrow. When it is moved back into the drawing area, the cursor shape changes back into the shape of the most recently used drawing instrument. The default drawing instrument is the paintbrush.

Quite a bit of testing is done in the program to ensure that objects are only drawn within the drawing area of the screen. In most cases, command letters are only accepted while the cursor is within the drawing area boundary.

When painting-in an area of the screen, if the cursor is lying on top of even a single pixel of the same colour as the paint colour, then you won't see anything being painted. So, before painting-in, lift the cursor, and move it to pixels of a different colour.

When filling an area, the program will only recognise lines that are in the current colour as boundaries. Therefore you have to be sure that the boundary of the shape to be painted is the same colour as the currently selected colour. To avoid the program filling beyond the drawing area, the drawing boundary is automatically redrawn in the current colour before painting any area. Using (P)aint before drawing anything will change the whole drawing area to your chosen colour, so allowing colours other than white to be used as your 'canvas'.

```
10 REM PAINTBOX
20 DEFINT A-Z
30 REM SET UP JOYSTICK OFFSETS
40 DIM Z(8,2)
50 FOR I= 1 TO 8
60 FOR J= 1 TO 2
```

```
70 READ A:Z(I,J)=A
80 NEXT J
90 NEXT I
100 DATA 0,-1,1,-1,1,0,1,1,0,1,-1,1,-1,0
,-1,-1
110 SCREEN 2,0,0
120 COLOR15,15,15:CLS
130 STRIG(0) ON
140 REM DEFINE PYTHAGORUS FUNCTION
150 DEFFNC(M,N,O,P)=SQR((ABS((M-O)^2))+(
ABS((N-P)^2)))
160 F=1:M=2:N=1:Q=1:P1=0:P2=1
170 ON STRIG GOSUB 1400
180 X=40:Y=172
190 GOSUB 320:REM SET UP SCREEN
200 GOSUB 410:REM SET UP ALL THE SPRITES
210 W=15
220 REM PLACE CURSOR
230 C5=1
240 PUT SPRITE 1,(X,Y),Q,N
250 IF N=3 THEN PSET(X,Y),15
260 IF X>57 AND FLG=1 AND N=2 THEN GOSUB
 2090
270 IF STICK(0)=0 THEN GOTO 290
280 GOSUB 1470
290 B$=INKEY$:IF B$= "" THEN GOTO 240
300 GOSUB 1560
305 GOTO 240
310 REM SCREEN DRAWING ROUTINE
320 C=1
330 FOR I=8 TO 32 STEP 24
340 FOR J=8 TO 152 STEP 24
350 LINE (I,J)-(I+16,J+16),C,BF:C=C+1
360 NEXT J
370 NEXT I
380 LINE (56,8)-(248,184),1,B
390 RETURN
400 REM SPRITE SET UP ROUTINE
410 FOR I=1 TO 10
420 S$=""
```

```
430 FOR I=1 TO 8
440 READ A$:S$=S$+CHR$(VAL("&B"+A$))
450 NEXT I
460 SPRITE$(I)=S$
470 NEXT I
480 RETURN
490 REM ARROW SPRITE
500 DATA 11110000
510 DATA 11000000
520 DATA 10100000
530 DATA 10010000
540 DATA 00001000
550 DATA 00000100
560 DATA 00000010
570 DATA 00000001
580 REM PAINT BRUSH SPRITE
590 DATA 11110000
600 DATA 11110000
610 DATA 11110000
620 DATA 00000000
630 DATA 11110000
640 DATA 01100000
650 DATA 01100000
660 DATA 01100000
670 REM ERASER SPRITE
680 DATA 11110000
690 DATA 10011000
700 DATA 10010100
710 DATA 11110010
720 DATA 01000001
730 DATA 00100001
740 DATA 00010001
750 DATA 00001111
760 REM COMPASS SPRITE
770 DATA 11000000
780 DATA 00100000
790 DATA 00010010
800 DATA 00001111
810 DATA 00001111
820 DATA 00010010
```

```
830 DATA 00100000
840 DATA 11000000
850 REM AEROSOL CAN
860 DATA 11100000
870 DATA 00111110
880 DATA 00100010
890 DATA 11111010
900 DATA 10001000
910 DATA 10001000
920 DATA 10001000
930 DATA 11111000
940 REM LINE SPRITE
950 DATA 10000000
960 DATA 10000000
970 DATA 10000000
980 DATA 10000000
990 DATA 10000000
1000 DATA 10000000
1010 DATA 10000000
1020 DATA 10000000
1030 REM BOX SPRITE
1040 DATA 11111111
1050 DATA 10000001
1060 DATA 10000001
1070 DATA 10000001
1080 DATA 10000001
1090 DATA 10000001
1100 DATA 10000001
1110 DATA 11111111
1120 REM HORIZONTAL BAR
1130 DATA 00000000
1140 DATA 11111111
1150 DATA 00000000
1160 DATA 11111111
1170 DATA 00000000
1180 DATA 11111111
1190 DATA 00000000
1200 DATA 11111111
1210 REM VERTICAL BAR
1220 DATA 10101010
```

```
1230 DATA 10101010
1240 DATA 10101010
1250 DATA 10101010
1260 DATA 10101010
1270 DATA 10101010
1280 DATA 10101010
1290 DATA 10101010
1300 REM STIPPLE CURSOR
1310 DATA 10101010
1320 DATA 01010101
1330 DATA 10101010
1340 DATA 01010101
1350 DATA 10101010
1360 DATA 01010101
1370 DATA 10101010
1380 DATA 01010101
1390 REM SPACEBAR INTERRUPT ROUTINE
1400 IF X>57 AND N=5 THEN GOSUB 1930:RET
URN
1410 IF X>57 AND N=8 THEN V1=1:V2=2:GOSU
B 2000:RETURN
1420 IF X>57 AND N=9 THEN V1=2:V2=1:GOSU
B 2000:RETURN
1430 IF X>57 AND N=10 THEN V1=2:V2=2:GOS
UB 2000:RETURN
1440 IF X>57 THEN RETURN
1450 IF POINT(X,Y)=15 THEN PLAY "L1402C"
:RETURN
1460 C5=POINT(X,Y):PLAY "L2405C06C":RETU
RN
1470 X=X+Z(STICK(0),1):Y=Y+Z(STICK(0),2)
1480 IF X>246 THEN X=246
1490 IF Y>183 THEN Y=183
1500 IF Y<9 THEN Y=9
1510 IF X<9 THEN X=9
1520 IF X<56 THEN N=1
1530 IF X>56 THEN N=M:F=0
1540 RETURN
1550 REM COMMAND LETTER ROUTINE
1560 IF (ASC(B$)<65) THEN RETURN
```

```
1570 IF B$="F" THEN BEEP:E1=E1+1:IF E1=2
 THEN SCREEN 1:COLOR 15,5,4:END
1580 IF X<56 AND B$="S" THEN X=152:Y=96:
N=M:RETURN
1590 IF B$="T" THEN SWAP Q,W:RETURN
1600 IF X<56 THEN RETURN
1610 CL=CW
1620 IF B$="Q" THEN C5=15:RETURN
1630 IF B$="U" THEN FLG=0:N=2:M=2:SWAP P
1,P2:RETURN
1640 IF B$="D" THEN FLG=1:N=2:M=2:SWAP P
1,P2:RETURN
1650 IF B$="H" THEN X=40:Y=176:N=1:RETUR
N
1660 IF B$="R" AND F2=1 THEN GOSUB 1810:
F2=0:RETURN
1670 IF B$="P" THEN LINE(56,8)-(248,184)
,C5,B:PAINT (X,Y),C5:RETURN
1680 IF B$="Z" THEN BEEP:Z=Z+1:IF Z>1 TH
EN LINE(56,8)-(248,184),15,BF:LINE (56,8
)-(248,184),1,B:Z=0:RETURN
1690 IF B$="C" THEN C1=X:C2=Y:M=4:N=4:F2
=1:RETURN
1700 IF B$="Y" THEN M=8:N=8:RETURN
1710 IF B$="V" THEN M=9:N=9:RETURN
1720 IF B$="K" THEN M=10:N=10:RETURN
1730 IF B$="W" THEN N=3:M=3:CW=CL:CL=15:
FLG=1:RETURN
1740 IF B$="L" THEN L=1:L1=X:L2=Y:N=6:M=
6:RETURN
1750 IF B$="B" THEN B=1:B1=X:B2=Y:N=7:M=
7:RETURN
1760 IF B$="X" AND B=1 THEN LINE(B1,B2)-
(X,Y),C5,B:B=0:RETURN
1770 IF B$="E" AND L=1 THEN LINE(L1,L2)-
(X,Y),C5:L=0:RETURN
1780 IF B$="A" THEN N=5:M=5
1790 RETURN
1800 REM RADIUS CALCULATION
1810 R = FNC(X,Y,C1,C2)
```

```
1820 D1=C1:D2=C2
1830 GOSUB 1860
1840 CIRCLE (C1,C2),R,C5,,,8/7
1850 RETURN
1860 REM CIRCLE RANGE CALCULATION
1870 IF D1+R>246 THEN BEEP:RETURN
1880 IF D1-R<57 THEN BEEP:RETURN
1890 IF D2+R>183 THEN BEEP:RETURN
1900 IF D2-R<9 THEN BEEP:RETURN
1910 RETURN 1850
1920 REM AEROSOL DRAWING ROUTINE
1930 IF (X-8<57)OR(Y-8<9) THEN RETURN
1940 RV=5*RND(1):IF RV<2 THEN 1940
1950 FOR I=1 TO RV
1960 PSET(X-(8*RND(1)),Y-(8*RND(1))),C5
1970 NEXT I
1980 RETURN
1990 REM HORX. VERT. AND STIPPLE ROUTINE
2000 IF (X+8)>246 OR (Y+8)>183 THEN BEEP
:RETURN
2010 FOR I=X TO X+7 STEP V1
2020 FOR J=Y+1 TO Y+7 STEP V2
2030 PSET(I,J),C5
2040 NEXT J
2050 NEXT I
2060 RETURN
2070 REM PAINTBRUSH PAINTING
2080 IF P1=1 THEN PSET(X,Y),C5:RETURN
2090 WL=3
2100 IF X+WL>246 THEN WL=WL-1:GOTO 2120
2110 FOR I=0 TO WL
2120 PSET(X+I,Y),C5
2130 NEXT I
2140 RETURN
```

Suggested Improvements to Paintbox

Using the (C)ircle command, only complete circles within the drawing area are permitted. A crude, but effective way of clipping circles at the edge of the drawing boundary is to redraw everything

outside the boundary after a circle is drawn. The following subroutine and other simple alterations will achieve this:

```
3000 REM MASKING ROUTINE
3010 LINE (0,0)-(56,192),15,BF
3020 LINE (56,0)-(256,8),15,BF
3030 LINE (56,184)-(256,192),15,BF
3040 LINE (248,0)-(256,192),15,BF
3050 GOSUB 320
3060 RETURN
```

The circle range checking routine (lines 1860–9000) should be altered thus:

```
1860 REM CIRCLE RANGE CALCULATION
1870 IF D1+R>246 THEN FC=1:RETURN
1880 IF D1-R<57 THEN FC=1:RETURN
1890 IF D2+R>183 THEN FC=1:RETURN
1900 IF D2-R<9 THEN FC=1:RETURN
```

Line 1910 should be deleted. The final alteration includes the addition of line 1845 which calls the subroutine:

1845 IF FC = 1 THEN GOSUB 3000 : FC = 0

Although quite a lot of graphics work is being done to carry out the screen redrawing, the subroutine still works very quickly.

Another improvement could be in error checking – instead of merely 'beeping', the program could indicate what you had done wrong in some visual way. A range of error sprites could be defined for each error. An appropriate sprite could then be displayed in the bottom left-hand corner of the screen when you had made a mistake. For example, when trying to paint in an area of the screen while positioned over a point of matching colour, a right-pointing arrow symbol could be shown, indicating that the cursor should be moved.

A more advanced painting option could be introduced allowed circles and boxes to be painted in, using vertical or horizontal lines or stipples. Whilst this is quite easy for the case of filling in bozes,

circles present greater difficulties. Here is a suggested method for filling in boxes. First alter line 1760:

1760 IF B$ = "X" AND B = 1 THEN LINE (B1,B2) – (X,Y),C5,B: B = 0: GOSUB 4000: RETURN

```
4000 C$=INKEY$:  IF  C$=""  THEN 4000
4010  IF  ASC(C$)<48 OR ASC(C$)>51 THEN BE
EP:RETURN
4020  IF  C$  =  "0"  THEN RETURN:REM NO TEXT
URING WANTED
4030  IF  C$  =  "1"  THEN V1=1:V2=2:GOSUB 50
00:RETURN
4040  IF  C$  =  "2"  THEN V1=2:V2=1:GOSUB 50
00:RETURN
4050  IF  C$  =  "3"  THEN V1=2:V2=2:GOSUB 50
00:RETURN
4060 RETURN
 .
 .
 .
5000  IF  B1>X  THEN  V1=-V1
5010  IF  B2>Y  THEN  V2=-V2
5020 FOR I= B1  TO  X  STEP  V1
5030 FOR J= B2  TO  Y  STEP  V2
5040 PSET  (I,J),C5
5050 NEXT  J
5060 NEXT  I
5070 RETURN
```

After the X command has been given, the program will wait for a number between 0 and 3 to be typed in. These correspond to:

0 Don't paint box.
1 fill in box with horizontal lines.
2 fill in box with vertical lines.
3 fill in box with stipple texture.

The subroutine lines numbered 5020–5070 are virtually the same as another routine in the program (lines 2010–2060). To avoid duplicating these lines, try altering lines which call the subroutine

(lines 1410, 1420 and 1430) and the subroutine starting at line 4000. You will have to keep lines 5000 and 5010 of the new subroutine though.

Filling-in circles in this manner causes immense problems. There are some quite hefty maths involved, so if you feel daring enough – start working at it!

As you will no doubt have noticed, it is possible to expand this program a great deal. We hope there is enough here for you to start with. As this is the final (and longest!) program in the book, it seems apt to end at this point.

Appendix A

MSX-BASIC functions

ABS(X)
Returns the absolute value of the expression X.

ASC(X$)
Returns a value that is the ASCII code of the first character of the string X$. If X$ is an empty string, an ''Illegal function call'' error is returned.

ATN(X)
Returns the arctangent of X in radians. The result is in the range –pi/2 to pi/2. The expression X may be any numeric type. This function is always performed in double precision.

BIN$(X)
Returns a string which is the binary value of a decimal number. X must be a numeric expression in the range –32768 to 65535. If n is negative, a two's complement form is used – BIN$(–n) is the same as BIN$!(65536–X).

CDBL(X)
Converts X to a double precision number.

CHR$(X)
Returns a string which represents the ASCII code for X.

CINT(X)
Converts X to an integer number. An ''Overflow'' error occurs if X is outside the range –32768 to 32767.

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

CSNG(X)
Converts X to a single precision number.

CSRLIN

Returns the vertical (column) coordinate of the cursor.

ERL/ERR

Error variables. When an error is trapped, ERR has the value of the error code, and ERL contains the line number of the line in which the error was trapped. If the statement that caused the error ws a direct mode statement, ERL will be 65535.

EXP(X)

Returns e to the power of X.

FIX(X)

Returns the integer part of X (fraction truncated).

FRE("")

This returns the number of free bytes in memory which can be used for BASIC programs, variables etc.

HEX$(X)

Returns a string which represents the hex value of X. X must be a number in the range −32768 to 65535.

INKEY$

Returns either a single character read from the keyboard or an empty string.

INPUT$(X)

Returns a string X characters long read from the keyboard.

INSTR([X,]A$,B$)

Searches for the first occurrence of string B$ in A$ and returns the position in A$ at which a match is found. The offset X may be used to set the starting position for the search. X must be in the range 0 to 255.

INT(X)

Returns the largest integer <=X.

LEFT$(A$,X)

Returns a string composed of the leftmost X characters of the string A$. X must be in the range 0 to 255.

LEN(X$)

Returns the number of characters in X$. All non-printable character and spaces are counted.

LOG(X)
Returns the natural logarithm of X. X must be greater than zero.

LPOS(X)
Returns the current position of the line printer print head within the printer buffer. However, this function does not necessarily give the actual position of the print head. X is a dummy argument.

MID$(A$,X[,Y])
Returns a string Y characters long from X$ beginning from the Xth character character position. Both X and Y must be in the range 1 to 255. If Y is omitted or if there are less than Y characters to the right of the Xth character, all rightmost characters returned.

OCT$(X)
Returns a string representing the octal value of the decimal argument X. X must be a numeric expression in the range −32768 to 65535.

PEEK(X)
Returns a byte (an integer in the range 0 to 255) read from memory location X. X must be in the range −32768 to 65535.

POS(X)
Returns the current horizontal position of the cursor. The leftmost position is 0. X is a dummy argument.

RIGHT$(A$,X)
Returns the rightmost X characters of string X$. This function is used like LEFT$.

RND(X)
Returns a random number between 0 and 1. The same sequence of random number will be generated each time a program is run.

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

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

SPACE$(X)
Returns the string of spaces of length X. X must be in the range 0 to 255.

SPC(X)
Prints X blanks on the screen. SPC may only be used with PRINT and LPRINT statements. X must be in the range 0 to 255.

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

STR$(X)
Returns a string representation of the value of X.

STRING$(X,Y) and STRING$(X,A$)
Returns a string X characters long that all have ASCII code Y or the first character of the string X$.

TAB(X)
Prints Spaces to position X on the console. If the current print position is already beyond position X, TAB does nothing. X must be in the range 0 to 255. TAB may only be used with PRINT and LPRINT statements.

TAN(X)
Returns the tangent of X in radians. TAN(X) is calculated to double precision.

USR[<digit>] (X)
Calls the user's assembly language subroutine with the argument X. <digit> is in the range 0 to 9 and corresponds to the digit supplied with an earlier DEFUSR statement for that routine. If <digit> is omitted, then USR(0) is assumed.

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

VARPTR (<variable name>) and **VARPTR (£<file number>)**
Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> before execution of VARPTR, else an "illegal function call" error results. The value returned will be an integer in the range −32768 to 32767. If a negative address is returned, adding this value to 65536 will give the actual address.

Appendix B

Error codes and messages

1 NEXT without FOR

You omitted to start a FOR. . .NEXT loop with a FOR. A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.

2 Syntax error

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

3 RETURN without GOSUB

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

4 Out of DATA

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

5 Illegal function call

A parameter that is out of the range is passed to a math or string function. An FC error may also occur as the result of:

1. a negative or unreasonably large subscript.
2. a negative or zero argument with LOG.
3. a negative argument to SQR.
4. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR$, or ON. . .GOTO.

6 Overflow

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

7 Out of memory

A program is too large, has too many files, has too many FOR

loops or GOSUBS, too many variables, or expressions that are too complicated.

8 Undefined line number
A line reference in a GOTO, GOSUB, IF. . .THEN. . .ELSE is to a nonexistent line.

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

10 Redimensioned array
Two DIM statements are given for the same array or DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero
A division by zero is encountered in an expression, or you attempted to raise zero to a negative power.

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

13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

14 Out of string space
String variables have caused MSX-BASIC to exceed the amount of free memory remaining. MSX-BASIC will allocate string space dynamically, until it runs out of memory.

15 String too long
An attempt is made to create a string more than 255 characters long.

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

17 Can't continue
An attempt is made to continue a program that:
1. has halted due to an error,
2. has been modified during a break in execution or
3. does not exist.

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

19 Device I/O error
An input/output error occurred on a cassette, printer, or CRT operation. It is a fatal error, i.e., BASIC cannot recover from the error.

20 Verify error
The current program is different from the program saved on the cassette.

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

22 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

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

24 Missing operand
An expression contained an operator with no operand following it.

25 Line buffer overflow
An entered line has too many characters.

26–49 Unprintable Errors
These codes have no definitions. Should be reserved for future expansion in BASIC.

50 FIELD overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or the end of the FIELD buffer is encountered while doing sequential I/O (PRINT #, INPUT#) to a random file.

51 Internal error
An internal malfunction has occurred – the chances are your machine needs looking at by a specialist.

52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified by the MAXFILE statement.

53 File not found

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

54 File already open

A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

55 Input past end

An INPUT statement is executed after all the data in the file has been INPUT, or for null (empty) file. To avoid this error, use the EOF function to detect the end of file.

56 Bad file name

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

57 Direct statement in file

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

58 Sequential I/O only

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

59 File not OPEN

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

60–255 Unprintable Errors

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

Appendix C

Full screen editor control keys

CONTROL KEY	SPECIAL KEY	FUNCTION
A		Ignored
*B		Move cursor to start of previous word
*C		Break when MSX BASIC is waiting for input
*D		Ignored
*E		Truncate line (clear text to end of logical line)
*F		Move cursor to start of next word
*G		Beep
H	Back space	Backspace, deleting characters passed over
I	Tab	Tab (moves to next TAB stop)
*J		Line feed
*K	Home	Move cursor to home position
*L	CLS	Clear screen
*M	Return	Carriage return (enter current logical line)
*N		Append to end of line
*O		Ignored
*P		Ignored
*Q		Ignored
*R	INS	Toggle insert/typeover mode
*S		Ignored
*T		Ignored
*U		Clear logical line
*V		Ignored
*W		Ignored
*X	Select	Ignored

***Y**		Ignored
***Z**		Ignored
[	ESC	Ignored
***Y**	Right arrow	Cursor right (Y is Yen sign)
*]	Left arrow	Cursor left
*^	Up arrow	Cursor up
*–	Down arrow	Cursor down
***DEL**	DEL	Delete character at cursor

All keys marked with asterisk (*) cancel insert mode when editor is in insert mode.

Appendix D

Differences between SV-BASIC and MSX-BASIC

The Spectravideo SV318 and SV328 computers include a language specification that is very close to that of MSX-BASIC. Indeed, there are only eight differences between the two. SV-BASIC handles screen modes, screen width, function keys, the sound channel, key click and printing slightly differently, and includes two graphics commands not included in MSX-BASIC, namely GET and PUT. We shall look at these in order:

SCREEN <mode>[,<option>]

Whilst MSX-BASIC allows four different screen modes, SV-BASIC only allows three. These are specified as follows:

SV-BASIC	MSZ-BASIC	Mode
SCREEN 0	SCREEN 0	40×24 Text mode
SCREEN 1	SCREEN 2	High resolution graphics mode
SCREEN 2	SCREEN 3	Low resolution graphics mode

As you can see, there is only one option available for the SCREEN command in SV-BASIC, as compared with the range of options in MSX-BASIC. This option varies according to which screen mode is in use.

In text mode, if the option is set to zero, then the function key display will disappear. If it is made non-zero, then the function key will reappear. The latter (surprise, surprise!) is the default.

In either of the graphics modes, the option is used to select the sizes of sprites to be displayed.

0 selects 8×8 unmagnified sprites
1 selects 8×8 magnified sprites
2 selects 16×16 unmagnified sprites
3 selects 16×16 magnified sprites

The precise meanings of sprites, 8×8, 16×16, magnified and unmagnified are all given in the chapter on graphics.

WIDTH [39][40]

In MSX-BASIC, any screen width between one and forty characters is permitted for the text screens. In SV-BASIC, width is limited to either thirty-nine or forty characters.

KEY [ON][OFF]

In MSX-BASIC, the KEY command is used to switch the function key display at the bottom of the screen on and off. In SV-BASIC, the screen option described above is used.

SOUND [ON][OFF]

The SOUND channel can be turned on and off in SV-BASIC, though this is not possible in MSX-BASIC. In the latter, SOUND is limited to making weird and wonderful noises!

CLICK [ON][OFF]

In MSX-BASIC, turning the key click on and off is one of the options available with the SCREEN command. In SV-BASIC, key click has its own command – CLICK ON makes the keys click and CLICK OFF makes them silent.

PRINT

In SV-BASIC, you can quite happily print things on any of the screens, simply by using the PRINT statement. In MSX-BASIC, PRINT works on the text screens, but on the graphics screens it does not, and a rather more convoluted method is required (see the description of the PRINT statement in Chapter 2 for a full description).

GET <x1,y1>)–(<x2,y2>),<array name>

In SV-BASIC, when you have drawn a picture on the graphics screen, the GET command allows you to save part of the picture into an array. When used in conjunction with PUT, you can then

put the piece of the picture that you've stored back onto the screen, anywhere you like. By far the easiest way of demonstrating this is with a short program:

```
10 SCREEN 1
20 DIM A(1000)
30 FOR X = 10 TO 100 STEP 10
40 CIRCLE (X,X),X
50 NEXT X
60 GET (10,10)-(100,100),A
70 PUT (125,100),A,PRESET
80 GOTO 80
```

Line 10 selects the high resolution graphics screen. Line 20 dimensions an array called A to be big enough to store the portion of the screen that is to be 'got'. Lines 30 to 50 instruct the computer to draw a series of circles starting with a radius of 10 pixels and centred on the point (10,10); then with a radius of 20 pixels centred on (20,20); and so on up to a circle of radius 100 pixels centred on (100,100). Line 60 asks it to GET a portion of the picture starting at the point 10,10 and ending at the point 100,100 and to store it in A. Then, in line 70 it takes A and PUTs it back onto the screen with its top left-hand corner at position (125,100). PSET is one of the options that can be used with PUT to alter the way that the pattern you've saved is re-drawn onto the screen.

PUT (<x,y>,<array name>[,<option>]

And so to the PUT command which copies the part of the picture specified by GET, to another position on the screen. The syntax for PUT is fairly simple – x and y specify the point on the screen to which the top left-hand corner of the pattern is to be copied, and the array name is the name that was given to the pattern to enable GET to be used. The options that may be specified when the pattern is drawn back onto the screen are as follows:

PSET

Causes the pattern to be put back onto the screen in exactly the same way as it was saved into the array.

PRESET

Causes the pattern to be drawn in the same way as by PSET, but with the foreground and background colour reversed.

AND

Combines the pattern colour with the screen colour, in such a way that the pattern that is being superimposed onto the screen is only drawn when the two coincide.

OR

The pattern that is being superimposed overlaps the pattern that is already on the screen, so that both patterns can be seen.

XOR

If a pixel from the pattern that is being superimposed onto the screen matches the one that is already there, then it is displayed in the background colour. XOR is the default option.

Whilst this explains in words what the different options do, the best way of understanding them is to try them out for yourself. The easiest way of doing this is to turn back to the program we used to demonstrate GET, and replace the PSET in line 70 with each of the other options in turn.

MSX is the first standard for home computers. Software written for one MSX computer will run on every other – they all understand the same MSX BASIC.

This book takes you from the simplest concepts of MSX BASIC to the complex techniques of using its stunning graphics and music capabilities; the MSX Macro languages. With drawing and painting programs, sprites, tunes to play, easy reference guides to key words and many useful hints and tips, this book is perfect for both beginners and advanced programmers wishing to find out more about the revolutionary MSX.

MSX was developed by Microsoft, the company responsible for the industry-standard MS-BASIC and MS-DOS operating system. MSX has been adopted by all the major Japanese electronics manufacturers, including Sanyo, JVC, Sony, Canon, Hitachi, NEC, Pioneer, Yamaha, Toshiba, Mitsubishi and Matsushita and Spectravideo in America.

ISBN [illegible]
CENTURY COMMUNICATIONS LTD
[illegible]