

THE DRAGON PROGRAMMER



S.M.GEE

Other books of interest from Granada:

The Apple II

**APPLE II
PROGRAMMERS
HANDBOOK**

R. C. Vile
0 246 12027 4

ATARI

**GET MORE FROM
THE ATARI**

Ian Sinclair
0 246 12149 1

**THE ATARI
BOOK OF GAMES**

M. James, S. M. Gee
and K. Ewbank
0 246 12277 3

The BBC Micro

**INTRODUCING
THE BBC MICRO**

Ian Sinclair
0 246 12146 7

**THE BBC MICRO –
AN EXPERT GUIDE**

Mike James
0 246 12014 2

**21 GAMES FOR
THE BBC MICRO**

M. James, S. M. Gee
and K. Ewbank
0 246 12103 3

**BBC MICRO
GRAPHICS AND SOUND**

Steve Money
0 246 12156 4

**DISCOVERING
BBC MICRO
MACHINE CODE**

A. P. Stephenson
0 246 12160 2

**ADVANCED
PROGRAMMING FOR
THE BBC MICRO
AND ELECTRON**

Mike James and S. M. Gee
0 246 12158 0

**BBC MICRO AND
ELECTRON
MACHINE CODE –
AN EXPERT GUIDE**

A. P. Stephenson and
D. J. Stephenson
0 246 12227 7

The Colour Genie

**MASTERING THE
COLOUR GENIE**

Ian Sinclair
0 246 12190 4

The Commodore 64

**COMMODORE 64
COMPUTING**

Ian Sinclair
0 246 12030 4

**THE COMMODORE 64
GAMES BOOK**

Owen Bishop
0 246 12258 7

**SOFTWARE 64
Practical Programs for
the Commodore 64**

Owen Bishop
0 246 12266 8

The Dragon 32

**THE DRAGON 32
And How to Make
The Most Of It**

Ian Sinclair
0 246 12114 9

**THE DRAGON 32
BOOK OF GAMES**

M. James, S. M. Gee
and K. Ewbank
0 246 12102 5

**THE DRAGON
PROGRAMMER**

S. M. Gee
0 246 12133 5

**DRAGON GRAPHICS
AND SOUND**

Steve Money
0 246 12147 5

**THE DRAGON 32
How to Use and
Program**

Ian Sinclair
0 586 06103 7

The Electron

**THE ELECTRON
PROGRAMMER**

S. M. Gee and Mike James
0 246 12340 0

**21 GAMES FOR
THE ELECTRON**

Mike James, S. M. Gee
and K. Ewbank
0 246 12344 3

**BBC MICRO AND
ELECTRON
MACHINE CODE –
AN EXPERT GUIDE**

A. P. Stephenson and
D. J. Stephenson
0 246 12227 7

**Learning is Fun!
40 EDUCATIONAL GAMES
FOR THE BBC MICRO
AND ELECTRON**

Vince Apps
0 246 12317 6

**The IBM Personal
Computer**

**THE IBM PERSONAL
COMPUTER**

James Aitken
0 246 12151 3

The Jupiter Ace

THE JUPITER ACE

Owen Bishop
0 246 12197 1

The Lynx

LYNX COMPUTING

Ian Sinclair
0 246 12131 9

The NewBrain

**THE NEWBRAIN
And How To Make
The Most Of It**

Francis Samish
0 246 12232 3

The ORIC-1

**THE ORIC-1
And How To Get
The Most From It**

Ian Sinclair
0 246 12130 0

**THE ORIC-1
BOOK OF GAMES**

M. James, S. M. Gee
and K. Ewbank
0 246 12155 6

The Dragon Programmer

The Dragon Programmer

S. M. Gee

GRANADA

London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

Copyright © 1983 by S. M. Gee

British Library Cataloguing in Publication Data

Gee, S. M.

The Dragon programmer

1. DRAGON (Computer)—Programming

I. Title

001.64'2

QA76.8.D7

ISBN 0-246-12133-5

First published in Great Britain 1983 by Granada Publishing
Reprinted 1983

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

All rights reserved. No part of this publication may be reproduced,
stored in a retrieval system, or transmitted in any form or by any
means, electronic, mechanical, photocopying, recording or otherwise,
without the prior permission of the publishers.

Contents

<i>Preface</i>	<i>vii</i>
1 Before You Switch On	1
What is a computer?	1
Special features of the Dragon	4
Programs and programming	5
2 Getting to Know Your Dragon	7
A new acquaintance	7
Setting up the Dragon	8
Using the keyboard	10
Entering a program	12
Using a tape recorder	13
Editing and our second program	15
As you go along	18
3 First Steps – Variables, PRINT, LET and INPUT	20
Variables	20
Storing things in variables – LET	22
Finding out what's in a variable – PRINT	23
Arithmetic	23
Understanding expressions – the order of evaluation	25
Variables and constants – the full expression	26
A short program	26
Another way of altering variables – INPUT	27
Variables and constants as expressions	27
Describing BASIC	28

INPUT prompting	29
Mixed PRINT	30
Some sample programs	30
4 Looping and Choice – the Flow of Control	35
The flow of control	35
Looping – the GOTO	35
Choice and conditions – the IF statement	38
Using IF	41
The FOR statement	45
Using the FOR loop	48
IF ... THEN and the colon	49
IF ... THEN ... ELSE	51
STOP	51
A final example	52
The flow of control summarised	53
5 Handling Text and Numbers	54
Strings	54
String expressions	56
Arrays	60
A word game	62
Initialising variables – DATA and RESTORE	63
Saving data on tape	65
6 Functions and Subroutines	68
The idea of a function	68
The Dragon's functions	70
Arithmetic functions	70
The trigonometrical functions	72
String functions	73
Special functions	75
User-defined functions – DEF FN and FN	78
Subroutines GOSUB and RETURN	80
Using subroutines	81

7	Low Resolution Graphics and Colour	83
	Controlling PRINT	84
	Print @	85
	Graphics characters	89
	Graphics characters in colour – CLS	91
	SET and RESET	94
	The POINT function	97
	Using joysticks	97
8	Sounds and Games	100
	Simple sounds – SOUND	100
	Programming tunes – PLAY	101
	Advanced PLAYing	106
	Some sound effects	108
	A speaking Dragon	109
	Attack the saucer – a low res game	110
9	High Resolution Graphics	114
	The high resolution modes	114
	Getting into high res – PMODE and SCREEN	116
	The high res commands – COLOR, PCLS, PSET and PRESET	118
	Drawing lines and boxes – LINE	121
	Drawing circles and ellipses – CIRCLE	123
	Finding out what's on the screen – PPOINT	127
	Using high res graphics	127
10	More High Resolution Graphics	131
	The DRAW command	131
	Using the DRAW command – user-defined characters	137
	PAINTing	142
	GET and PUT	143
	Graphics pages – PCLEAR AND PCOPY	145
	The saucer game again!	148

viii *Contents*

11	Logic and Other Topics	151
	Logic and the conditional expression	151
	Inside the Dragon – PEEK, POKE, CLEAR and USR	154
	More INPUT and PRINT	155
	REMark and good programming	157
	Where next?	158
	<i>Index</i>	159

Preface

The Dragon is an exceptionally versatile micro but if it's the first computer you have tried to use, or even if you have had experience of another computer that does things differently, you may be finding that writing programs of your own that produce the effects that you want is really very tricky. If you are at all daunted by the Dragon you are certainly not alone and this book has been written to help you tame it and make it an obedient and useful beast! This is a book for learning Dragon BASIC from first principles right up to some advanced techniques and so has something to offer every Dragon owner who wants to learn to program or to improve his or her programming.

Chapter One introduces some ideas relevant to all computers and programming in general. It also indicates some of the special features of the Dragon. Chapter Two shows how to get ready to begin learning how to program on your Dragon and includes a program that provides your Dragon with its very own test card for tuning the TV set. We start to learn programming in Chapter Three. This introduces variables and some essential keywords, such as PRINT and INPUT, used to handle them. Chapter Four introduces the idea of the flow of control which is central to computer programming. Chapter Five shows how your Dragon handles both numbers and words and how both can be combined in your programs. The uses of the Dragon's functions and the related idea of subroutines are the main topic in Chapter Six. This is where the very important commands RND, INKEY\$ and TIMER are discussed. The book pays a great deal of attention to the Dragon's graphics and colour, starting in Chapter Seven which deals with low resolution graphics. Sound is added in Chapter Eight, which also presents a flying saucer game using colour graphics and sound to exciting effect. Chapter Nine starts to explore the Dragon's high resolution graphics capabilities. This topic is continued in Chapter Ten which

deals with some of the most powerful, if difficult to master, graphics commands. A high resolution version of the saucer game from Chapter Eight is given in this chapter to enable you to see the difference in the effects created by hi res and low res graphics. Both these chapters include lots of short programs that show off the features being discussed. The final chapter covers the topic of logic and also introduces some commands that allow the programmer to control the Dragon's inner workings in a direct way.

This book takes a very different approach from that adopted by a manual. It does more than simply tell you what commands are there for you to use. Indeed, it shows you *how* to use each command and discusses which command suits which situation. This means that as you gradually evolve your programming style you also gain a good understanding of how BASIC works.

As you will already realise, this book covers a lot of ground. However, it is not meant to be hard going – it is meant to be fun. More importantly, it is not meant as a reading book – it is meant as a try-it-yourself book. There are lots of programs and program snippets included and they are all there for you to use. So stand by your Dragon and enjoy learning BASIC.

My grateful thanks are due to Richard Miles of Granada Publishing for his encouragement throughout this project.

Chapter One

Before You Switch On

The Dragon is one of a new generation of machines that are enabling ordinary people to use computers. Already we are beginning to take such computers for granted. You can go into a high street store and buy a small, easily portable box and take it home. What's so special about that? Well, if you were able to turn the clock back even ten years, such an idea would have been quite ludicrous. Nowadays you can set up your Dragon on a small coffee table in front of your ordinary TV set – but only twenty years ago a computer with a similar memory capacity would have filled the whole of a family house.

You may wonder whether the circuitry inside so slim and compact a case can really do all the things that the older, vastly larger machines could do. In fact it can do much, much more. It can tackle more complicated calculations and produce graphics and emit a variety of sounds – both of which areas were not within the capabilities of even very large machines only ten years ago. Any limitations in today's micro are not really to do with its small exterior – although they are governed by its price. However, the fact that modern microtechnology has enabled so much to be squeezed into so small a space is a fact that the Dragon user can cheerfully ignore, for at the same time as computer design is becoming more sophisticated, computer use is becoming accessible to everyone who wants to join in.

What is a computer?

This question can be answered at many levels. Whole chapters, or even books, can be devoted to it. However, you really do not even need to ask this question in order to use your Dragon. After all, we all watch TV, but few of us ever question what exactly a TV is. If you

2 The Dragon Programmer

want to learn every detail of how the Dragon works then you have no choice but to master electronics. Fortunately it is not difficult to gain an appreciation of what a computer does and roughly how it does it without knowing anything about electronics or the ever present ‘chip’! The fact is, computers would exist if electronics had never been invented. Indeed, the first of them were built using cogs and gears and it was one hundred years before a valve (an early electronic component) found its way into these machines. Although in practical terms the computer seems a product of microprocessor technology, the underlying idea doesn’t depend on the materials you choose to build it from.

Computers are composed of a number of parts, each performing a well identified function. All computers need to communicate with the outside world. The Dragon does this by a keyboard, which you type on, and the TV screen, on which the Dragon shows you what you have typed and anything else you need to know. The keyboard is an *input* device and the TV screen an *output* device. There are other input/output (I/O) devices for use with a computer. You can buy joysticks that can be used with the Dragon, for example, and the Dragon can produce its output on a printer instead of, or as well as, the screen.

A machine would hardly be worth calling a computer if it could only receive information and pass it on unchanged. It might rather be classed as a telephone or a telex! There must be some mechanism for changing or processing information before it is printed out. These days this usually takes the form of complex electronics hidden inside the computer, often referred to as the *central processing unit* or CPU, but also going under a traditional English name, betraying the fact that computers were once made of cogs and gears – the ‘mill’.

In the Dragon the CPU is in a single chip known as a 6809. For programming in BASIC what the 6809 does isn’t of too much importance, and the way it does it certainly isn’t! But in general it performs arithmetic and other operations on information input from the keyboard and stored within the machine. A list of instructions called a *program* controls what operations it does. This aspect is so important it could almost be said, a computer *is* a machine that will obey a list of instructions – but any definition of a machine as complicated as a computer is dangerous! The precise instructions the Dragon can obey will occupy the rest of this book, so for the moment the subject will be set aside.

For a computer to obey instructions concerning what to do with various pieces of information, it must have somewhere to store both

the information *and* the instructions. This part is known as *memory*, but the slightly less general term RAM (standing for *random access memory*) is almost universally used instead. RAM can be thought of as a sort of note pad, on which the CPU can record its instructions and any necessary data. Every memory has a limited capacity – an important measure of a computer's power. The larger the memory the larger the list of instructions it can hold. The most convenient unit of measurement for computer memory capacity is the *byte*, which, roughly speaking, equals one character. (Here the term character means a letter, a digit or any normal punctuation.) So a 400 byte memory could store enough characters to hold a page of this book. The only problem is, this unit of measurement is a little too small, because normal computer memories can store thousands of characters. So it makes good sense to think in terms of thousands of characters, and the unit used for this is the *kilobyte*, often shortened to kbyte or even just K. For various reasons 1 kbyte isn't 1000 bytes as its name suggests, but 1024 bytes. (You may notice that this strange number is the nearest power of two to 1000 and, as you might already know, computers work in binary which is based on 'two' states.) The Dragon has 32K of RAM, which can be extended by external memory units. However, 32K is quite enough for most applications! (Early computers used by the military to calculate missile trajectories, etc., often had less than 16K!)

This book is an introduction to BASIC. BASIC is only one of a number of computer languages, but it is the one that the Dragon has installed in it when you buy it. It is contained in a pre-allocated part of the Dragon's memory. As a computer user you cannot gain access to this part of the memory to change its contents and it is known as ROM (standing for *read only memory*).

This combination of I/O devices, CPU and memory is all there is to a computer: the I/O for communicating with the outside world, the CPU for calculations and general information processing and the memory for storing instructions the machine obeys and data that the CPU acts on. In practice, there is one addition. When you switch your Dragon off its memory is wiped clean. For this not to happen, most computer memory needs a constant supply of electricity. If you switch off, the information is lost. This sort of memory is often known as *volatile*. The memory loss is a problem, for it suggests we have to type in instructions every time the machine has been switched off. To overcome it most computers have a second, *non-volatile* form of memory. In the Dragon this is a standard cassette tape recorder for saving programs and data in a form that can

4 *The Dragon Programmer*

survive power switch off. It also has the advantage of being 'removable' – you can record a program on a cassette, then take it out of the recorder and even send it to someone else. The Dragon is then ready to start on a new program or go back to an old one, which you can do by 'loading' it from an earlier recorded tape.

The Dragon can also use a second but less common form of removable storage – the 'program cartridge'. You can buy cartridges with programs already contained in them which plug into the slot on the right-hand side of the Dragon. There are two main differences between cassette storage and cartridge storage. Firstly, when you plug in your cartridge the program it contains is available almost instantaneously – you don't need to wait for it to load. Secondly, whereas you can use cassette tapes to store your own programs as well as to use prerecorded ones, you cannot use a cartridge to save new programs. The memory used by the cartridge is therefore virtually the same as ROM – in fact it is more properly called EPROM (*erasable programmable read only memory*) and the instructions it contains can be wiped out or changed using special equipment.

The problem with cassette storage is that if you want to save a lot of programs you need to have a large number of tapes, or else it can take a long time to load the particular program you want – though judicious use of the counter on your tape recorder (if it has one) can help with this. An alternative solution is to store your programs on *floppy discs* using a *disc drive*. The Dragon can accommodate disc drives if you first acquire a disc interface that would be plugged into the Dragon's cartridge slot.

Special features of the Dragon

Every computer system from the largest to the smallest is some combination of the elements mentioned above – plus extras of course – so why are there so many different versions? This is rather like asking why there are so many different motor cars – the short answer is that different designers have different ideas. Lots of the apparent differences are really to do with styling. The choice of casing and of keyboard are largely governed by cost considerations and what the machine will be used for. A home computer, for example, does not need to be built to withstand extremes of temperature or humidity. Similarly, different applications need different keyboards – a numeric keypad is a must for some data

entry tasks, but can be dispensed with for games playing.

The choice of CPU chip might seem to be one obvious way in which micros differ from one another. At some levels there are some real differences according to which chip has been selected, but as far as the BASIC programmer is concerned the choice of CPU matters hardly at all. Other features of the hardware are much more important. In the case of the Dragon the real bonuses are the colour graphics, the sound generator and the availability of joysticks and program cartridges. In other words, this one portable box contains all you need for many happy hours of programming fun.

Programs and programming

As already mentioned, a computer obeys a list of instructions stored in its memory. These instructions are known as a program. Writing them is known as *programming*. Programming is often thought of as an activity deriving from modern digital computers. It must be remembered, though, that people have been writing instructions for other people to obey since the invention of writing. In this sense, programming is not new, and can be seen as recipes, knitting patterns, etc., in almost every home. Written music is perhaps one of the best examples of traditional programming. Sheet music is a program that will instruct a musician to play a specific tune, very like a computer program in fact, since it relies on using a special language, more precise than ordinary language. Just one note out of place and you have a different tune! A computer program uses a special, equally precise language. In the case of the Dragon this language is BASIC, the most popular programming language. As with written music, even slight changes in a BASIC program can bring about complete changes in meaning, so it is important as you learn BASIC to pay attention from the start to the fine detail. It is not like learning English, where you can first learn words and sentences and then add punctuation – you have to take notice of every comma in a line of BASIC for it to make any sense!

If you find this talk of strict rules worrying, it should be said that the rules are usually very simple and very regular. Unlike English, there are rarely any exceptions to spelling and punctuation rules in BASIC! Also, there are some powerful underlying ideas. Once you have recognised these they make it easy to understand why the rules are there at all. As you read this book there are therefore two types of thing to learn – the fine detail concerning the exact form of each

6 *The Dragon Programmer*

BASIC statement, and the general features shared by all programming languages. The fine detail is important to get a program to work, but understanding the general principles is what makes programming a sensible occupation.

Chapter Two

Getting to Know Your Dragon

There are two problems in using a computer. The first is simply getting it set up and getting used to its idiosyncrasies. The second is writing working programs. This chapter deals with both these problems so that we can get them out of the way before we get down to the main task of learning BASIC.

A new acquaintance

Getting to know a computer is a problem that exists even if you're an expert. For although there is a lot in common between different computers, there are always enough little differences to mean that there has to be a period of adjustment when moving from one machine to another. For example, nearly every computer uses a standard typewriter (or QWERTY) keyboard, but most place extra but very important keys in slightly different places. This can make even the most expert look silly at first! Now if you're an expert, then you know that this early phase soon passes, but if you're a beginner you might panic and think that computing was always going to be this tricky! The trouble is that not being 'at home' with your computer can make easy programming ideas seem difficult.

There is no way to avoid this early barrier to programming because being on friendly terms with your computer is simply a matter of time and a matter of using it. You'll come through this rather frustrating period more easily if you bear in mind the following advice:

Separate in your mind any difficulty that you encounter in using your Dragon from any difficulties that you have with programming.

8 *The Dragon Programmer*

Don't immediately assume that any strange behaviour on the part of your Dragon is its fault – at first the chances are that the mistake is yours!

Don't immediately assume that any unexpected behaviour of a program means that your Dragon is illogical – computers are ruthlessly logical. Try not to confuse typing errors with programming errors.

To try and help you identify this initial difficulty and to help you overcome it, this chapter includes two short programs that you should try to get running on your Dragon before moving on to the rest of the book. They are presented as complete and working programs for you to use to find out about the non-programming problems of using the Dragon. At this stage you are not expected to be able to understand how they work and you might like to return to them as you read through later chapters to see how you're progressing.

Setting up the Dragon

The Dragon is one of the world's easiest computers to get going. All you have to do is to plug the power supply (the solid white box) into the mains and then connect the small rectangular connector into the socket on the back of the Dragon marked 'TRANS SUPPLY'. At this point you can switch the machine on using the on/off switch next to the power socket. If you do this you should be able to hear a faint click inside the case. Don't panic if you cannot hear anything, but this is usually a fair guide to whether you have a working Dragon or not – if you can hear the click there is a good chance things are working. If you can't hear anything then check that the mains power is switched on. If it is then carry on with the setting up procedure – it might just be that your Dragon is a particularly quiet beast!

Connecting the mains to your Dragon and switching on is the only thing that is required to make it start working. However, if you want to see what it is doing you will have to connect it to a working UHF TV set! This operation is a little more difficult than connecting the mains to the Dragon, because it involves a piece of equipment that will vary from house to house – the TV set. When connecting the Dragon to the TV set, it will help to think of it as being an extra channel. Yes, the Dragon is not only a computer, it is a small

television station! When your TV set was first delivered it had to be tuned in to receive the channels that are used in your area. In the same way, when you first use your Dragon you have to tune the TV set to receive it. If your TV set has push button tuning then you will have to decide which button is going to be the 'Dragon channel' and find out from the instruction manual that came with the set how to tune it in. If your set has dial tuning then finding the Dragon channel is exactly the same as finding any other channel. However, before you connect the Dragon to the TV set, either tune the channel button that you are going to use or set the dial to receive BBC 2. (Non-UK readers must consult the tuning information that came with their Dragon to discover their equivalent of BBC 2.) The reason for this is that the Dragon channel is just a little 'higher' than BBC 2 in most areas and rather than start the search from just anywhere it is easier to start from BBC 2 and then tune the set away from the other BBC and ITV channels.

With the set tuned to BBC 2, take the lead that came with your Dragon with the TV aerial plug on one end and plug it into the TV's aerial socket (removing the original aerial's plug first!). Then plug the other end into the socket at the back of the Dragon marked 'TV'. Now with both the Dragon and the TV switched on (you should turn the TV's sound right down at this stage to avoid unpleasant noises) start tuning the TV set. As you get close to the Dragon channel you will begin to see a fuzzy picture. Keep going until you can see the message

© 1982 DRAGON DATA LTD
16K BASIC INTERPRETER 1.0
© 1982 BY MICROSOFT

OK

clearly on the top of the screen. Just as with any TV channel, if you haven't tuned in exactly the picture quality will be poor, so take care and be patient with the fine tuning until you have a nice sharp image.

If you are using a colour set then you may have to adjust the brightness, contrast and colour controls to get the best possible picture, but don't do this until you have finished tuning for the sharpest picture, or things will get hopelessly confused. If you're using a black and white set then you won't be able to see the colours that the Dragon produces, but you still might have to adjust the brightness and contrast on your set to produce a satisfactory picture. A black and white set will work perfectly well with the Dragon, but

instead of seeing its different colours you will see nine different shades of grey ranging from black to white.

If you want to, you can leave any final adjustments until after you have entered the program given later in this chapter, which produces the Dragon's own test card pattern and will help check the fine tuning.

Using the keyboard

After getting your Dragon going the next thing to do is to start to learn your way around the keyboard. The basic layout of the keyboard is that of a typewriter, but there are a few extra keys. If you are not a typist then you will find it difficult at first to find the letter that you are looking for. Once again the best advice is not to panic, it is surprising how quickly you get to know where the most often used letters are. If you just press a few of the keys with capital letters printed on, if your Dragon is working you will see the letters appear on the screen. To rub out any letters that you have typed simply press the '←' key. This is the backspace key and deletes characters one at a time from the screen.

The backspace key is different from the letter and number keys in that instead of producing anything on the screen it is an instruction to do something – i.e. to delete a character on the screen. A key that behaves in this way is usually called a *control key*. The Dragon has a number of control keys and it is worth knowing what they are all for at this early stage.

The key marked 'CLEAR' does what its name suggests, it clears the screen. It is important to realise that it is only the screen that is cleared. Apart from this change your Dragon is unaffected by CLEAR.

The key marked 'BREAK' has no effect unless you are running a program, when it will stop the program. In other words BREAK breaks into a program!

The key marked 'ENTER' is extremely important. If you just push it a few times you will see that all it appears to do is start a new line for you to type on. For this reason it is sometimes called *carriage return*, after the lever on a typewriter that gives you a new line. However ENTER has another function. As well as starting a new line, pressing ENTER signals to the Dragon that you have finished typing something and it is OK for it to look at it and perhaps even obey it. Until you press ENTER what you type on a line is your

business – the Dragon pays no attention to it. You can edit it using the backspace key until you have got it just right and only when you press ENTER will the Dragon take notice of what you have typed. At this stage most of the things that you type at random and then follow by ENTER will not amount to anything that the Dragon can understand and so you will tend to get strange error messages flashed back at you.

As well as the control keys there are a number of other control functions that you have to press two keys to make work. In particular, if you press the SHIFT key and ← you will delete an entire line. Pressing SHIFT and Ø will change the black on green characters that the letter and number keys normally produce to green on black. This is known as inverse video on other computers, but on the Dragon these characters appear in place of lower case and pressing SHIFT and Ø changes from the usual upper case characters to lower case characters. Pressing SHIFT and Ø a second time returns the Dragon to the normal black on green upper case characters. You may find this use of upper and lower case to refer to black on green and green on black characters confusing, but if you use a printer you will see that the black on green characters do come out as upper case and the green on black ones do come out as lower case! Finally, pressing SHIFT and @ together will temporarily stop the Dragon printing on the screen so that you have time to look at it. You can restart output by pressing any other key on the keyboard.

There is one other control key that isn't placed on the keyboard. The RESET button on the left-hand side of the case works like the BREAK key in that it will stop a running program, but the way that it works is different. The only time that you should use the RESET button is when the BREAK key doesn't succeed in stopping a program. In this sense it is a last resort! You may succeed in stopping the program, but there is no guarantee that the program will still be stored in memory!

It is worth summarising the action of the control keys

←	back space
BREAK	stops a program
ENTER	enters the line you have just typed
CLEAR	clears the screen
SHIFT ←	erases a line
SHIFT Ø	change upper/lower case
SHIFT @	pause output to screen
RESET button	panic stop button!

Now you should be ready to tackle the problem of entering a program.

Entering a program

As explained in Chapter One, a program is a list of instructions that a computer can obey. In BASIC this list of instructions is built up by typing in lines of commands, each one beginning with a number – the line number. The easiest way to understand this is to try out a short program. Don't worry if you don't understand how the program works, just concentrate on entering it correctly. Before you begin, switch your Dragon off and then, after waiting a moment, on again. This will produce the familiar copyright message on the screen and ensure that anything you may have typed in while experimenting is cleared out of the machine. Now type the following line, taking care not to make a mistake. If you do make a mistake before you have pressed ENTER then use the backspace key to delete all the characters up to and including the incorrect one and then type the rest of the line again. If you notice the mistake after you have pressed ENTER then simply retype the line again – a better method of correcting ENTERed lines will be described later.

```
10 A=0
```

Notice that the last character is a zero, not a letter 'O' (the zero is on the far right of the top row of keys).

After you have typed this line you have to press the key marked ENTER on the far right of the keyboard. As already explained, the purpose of this is to tell the Dragon that you have finished typing the line and that it can try to incorporate it into any program that it already has. Now enter the following line

```
20 PRINT A
```

Once again press ENTER when you have finished typing it. Next, enter the following line

```
30 A=A+1
```

And finally enter

```
40 GOTO 20
```

You should now have the following program displayed on the screen where you typed it:


```

1Ø A=Ø
2Ø PRINT A
3Ø A=A+1
4Ø GOTO 2Ø

```

If you want to confirm that this program is stored in memory type LIST (remember to press ENTER) and you should see another copy of what you typed in appear on the screen. This is a list of instructions that you can make the Dragon obey by typing RUN and then pressing ENTER. But before trying this it is worth noticing that there are two ways that the Dragon can obey commands. If you type a command without a line number and press ENTER then the Dragon will obey the command at once – this is called *immediate mode*. However, if you precede the command with a line number and press ENTER then the command is added to whatever program already exists, ready to be obeyed at some later time – this is called *deferred mode*. When you type RUN there is no line number so the Dragon obeys the command immediately.

When you do type RUN you will find that a sequence of numbers starting at Ø are printed on the screen. However, it all happens so fast that you might not see it happen! Now is the time to try pressing SHIFT and @ at the same time to pause the display (press any key to continue). When you grow tired of seeing numbers printed on the screen then press the BREAK key and your first program will stop!

Using a tape recorder

The next demonstration program is considerably longer than the first, and as it is likely that you will be loth to simply switch the machine off and so lose it altogether once you have finished entering and running it, now is the time to learn how to use a cassette recorder to save and load programs. The Dragon can use almost any standard tape recorder to store programs, but it is true that the better the tape recorder the more reliable the result. So if you are thinking about buying a cassette recorder for use with your new Dragon, invest at least £20 or so and try to buy a model that uses miniature jack sockets for earphone and microphone connections and has a 'remote' connection that can be used to start and stop the motor. If the tape recorder that you plan to use doesn't use miniature jack sockets then you will have to buy an adaptor from your local hi-fi shop, because miniature jack plugs are all that the Dragon comes equipped with.

To connect the Dragon to your tape recorder simply plug in the two jack plugs on the audio lead that came with your machine (you can easily recognise it because it's the only lead not already in use by this point!) into the socket marked 'TAPE' on the left-hand side of your Dragon. The larger of the two grey plugs should be inserted into the microphone socket on your tape recorder. The smaller should be plugged into the remote socket or just left if the tape doesn't have a remote socket. The remaining black plug should now be connected to the earphone socket. You are now ready to record your first program.

Find a tape with nothing important on it (actually completely fresh blank tapes work better) and rewind it to the beginning. Then wind the tape on by hand with a pencil until you cannot see the leader and recording tape is visible. Set the volume control to about half way (this is unnecessary if your recorder has automatic volume control). To give the Dragon something to save, type in

```
1Ø REM this is a test
2Ø REM this is the second line
3Ø REM this is the third line
```

and then type

```
CSAVE "TEST"
```

press the play and record button on the recorder and press ENTER. If you are using the remote control socket, your Dragon will switch on the recorder's motor at this point. (Otherwise it will already be running.) When the message 'OK' next appears on the screen the program is saved on tape. However, it is still possible that although the program is recorded on tape it may not be good enough to be read back into the Dragon for one of a number of reasons. For example, you may have the record level set too high or there may be a fault in the tape.

To check that the program has been successfully saved, rewind the tape. Now type

```
CLOAD "TEST"
```

set the recorder playing and press ENTER. You will see a flashing 'S' in the top right-hand corner of the screen. This lets you know that the Dragon is Searching for the program. When it finds it the 'S' changes to a flashing 'F', followed by the name of the program that has been Found. If all has gone to plan the next message that you should see on the screen will be another 'OK', meaning that the

program recorded on tape can be read. If everything hasn't gone well you might get the message 'I/O ERROR'. This means that for some reason the Dragon has misread the information on the tape or cannot even find it. If things are really bad then nothing at all will happen, as the Dragon continues to search the tape in vain. In this case the best thing to do is to wind the tape back and listen to it (taking the jack plug out of the earphone socket on the recorder in order to hear what is happening.) At the start of the tape you will hear a steady tone. This should be loud, but not distorted. If this sounds OK then wind the tape on and listen to an unrecorded patch at the same volume. You should hear a very soft hiss. If it sounds like a rain storm then reduce the volume and perhaps alter the tone control and try again. If an unrecorded patch gives absolute silence then you should try again with the playback volume increased.

If you have forgotten the name of the program that you want to load, you can use CLOAD "". This will load the first program encountered on the tape irrespective of its name.

Once you have the tape system working don't alter the volume controls or tone controls unless it is necessary.

Editing and our second program

At this stage you should feel confident enough to tackle entering a longer program. The program given below will produce the 'test card' pattern shown in Fig. 2.1. Apart from being quite an

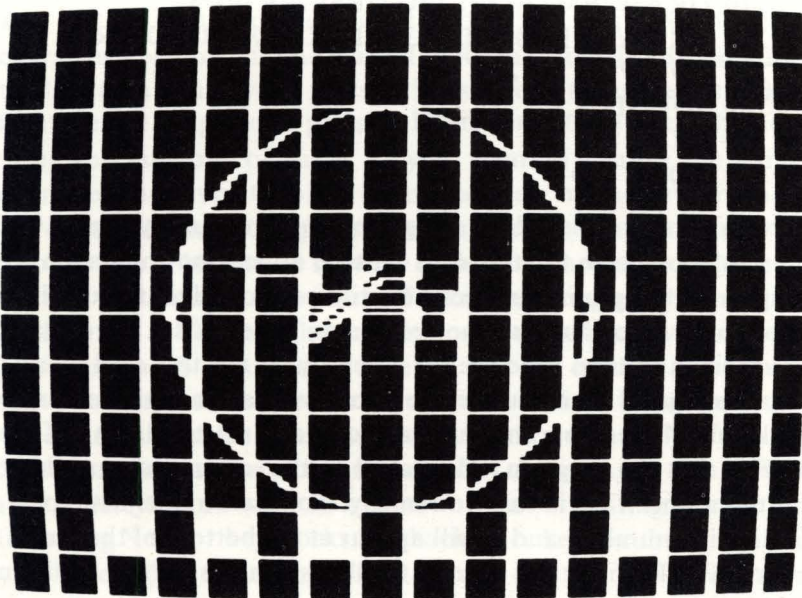


Fig. 2.1. 'Test Card' pattern.

interesting pattern you could use it to adjust the TV set that you are using with your Dragon to produce the sharpest image with the best colour. Take care to type each line in correctly.

```

10 PMODE 3
20 SCREEN 1,0
30 COLOR 4,1
40 PCLS
50 FOR I=0 TO 240 STEP 16
60 LINE (I,0)-(I,191),PSET
70 NEXT I
80 FOR I=0 TO 191 STEP 16
90 LINE (0,I)-(255,I),PSET
100 NEXT I
110 LINE(0,191)-(255,191),PSET
120 LINE(255,0)-(255,191),PSET
130 CIRCLE(120,96),64
140 PAINT(34,165),2,4
150 PAINT(66,165),3,4
160 PAINT(98,165),4,4
170 PAINT(130,165),2,4
180 PAINT(162,165),3,4
190 PAINT(194,165),4,4
200 DRAW "C2BM80,79R20D10R40D15"
210 DRAW"BM130,80L10G25R6E20R10U5"
220 DRAW "BM80,79D2L3R3D3R16D10R35D10R8"
230 FOR P=1 TO 255
240 SOUND P,1
250 NEXT P
260 GOTO 230

```

When you have finished entering the program, simply type RUN and you should see the test card pattern appear on the screen. In addition the program produces a continuous sound that you will be able to hear if you turn up the volume on your TV set. If you can't hear the sound then tune the set carefully. If the test card doesn't look like Fig. 2.1 then you are likely to have made a typing error, so check the whole program very carefully against the listing. Type LIST so that the program is displayed on the screen. You may have omitted a line. This is easy to remedy. To insert a line just type it, with its line number, and it will appear at the bottom of the screen. Press ENTER and it will automatically assume its correct position in the program listing. Numbering in computer programs is

normally done in jumps of ten, so that you can easily insert extra lines if you need to. If you want to delete an entire line from a program just type its number and press ENTER and the line will disappear from listings, so do be careful not to delete program lines you want in this way. If you find an error in a line you do not have to retype the entire line. Instead you can EDIT it.

There are a wide range of editing commands and they are all listed in the Dragon's manual. Part of the problem of editing is in fact remembering all the different commands. After a while you will find that there are only a few that you actually use often. For this reason only a few of the most useful commands will be described below – for the rest, when and if you need them, see the manual. To edit a line type EDIT followed by its line number. This places you in EDIT mode and the line in question is printed in full so that you can see what is happening. On the line below you will see the line number and the flashing cursor. If you now press the space bar you will be surprised to see the line appear, one character for each press of the bar. The character that would appear next if you pressed the space bar is called the *current character* and all the characters to the left of the current character cannot be altered by editing. If you want to return to the beginning of the line press L and you will see the line printed in full and on the next line the line number, just as it was when you first started editing. If you want to move the current character back by one or a number of characters you can press the ← key. Notice that this doesn't delete anything, it just moves the current character position left one place. If you want to delete something in the line then you have to press D. This deletes the current character from the line. Repeatedly pressing D will continue to delete characters until you run out of them. If you want to change the current character then type C followed immediately by the new character.

Perhaps the most complicated editing action is inserting new text into the line. This can be done by typing I immediately followed by the text that you want to insert. To stop inserting you have to press SHIFT and ↑. To move the current position to a letter within the line you can type S followed by a letter and the current position will move to the first occurrence of that letter. If all these commands weren't enough there is one last, very useful command, X. If you type X the current position moves to the end of the line and you are free to type whatever you want to extend it. It is interesting to note that following X the ← key works as a backspace key to rub out anything you have typed or anything that already exists. Finally, when you have finished editing the line press ENTER to indicate, as always,

that you have a line to be added to the program.

The editing described above allows you to change a single line of BASIC, but what about the problem of changing whole chunks of a program? If you want to see the whole program, type LIST, but if you want to see part of the program you can use LIST 'start'-'finish' where 'start' and 'finish' are line numbers. This command, not surprisingly, will list the part of the program between and including line number 'start' and line number 'finish'. If you leave out 'start' then the first line of the program is assumed and if you leave out 'finish' the last line of the program is assumed. For example, LIST 20-50 will list lines 20 and 50 and all the lines between. LIST -50 lists the program from the first line to line 50 and LIST 20- lists the program from line 20 to the end. The DEL command will allow you to delete blocks of program lines specified in the same format as LIST. So DEL 20-50 deletes the same lines that LIST 20-50 will list on the screen and so on. DEL is a very dangerous command. Make a small mistake and you can lose more than you expected! If you do want to remove a complete program from memory and return the Dragon to the state it was in when you first switched on, just type NEW. Like DEL, NEW should be used with care as you cannot restore a program that you have lost by using NEW.

To summarise:

To see all or any part of a program use LIST 'line number'-'line number' and press SHIFT @ to pause the listing.

To insert a line simply type it with an appropriate line number.

To change the contents of a single line use EDIT and the editing commands. The most often used are L(ist),C(hange),D(etele), S(earch),I(nsert) and X for eXtend. If the change is major then it is often quicker to retype the line completely!

To delete a single line just type its line number followed by ENTER.

To delete a block of lines use DEL 'line number'-'line number'.

To delete an entire program and get the Dragon ready for a new program, type NEW.

As you go along

As you learn BASIC and the special features of the Dragon from the rest of this book you are bound to improve, both in your use of the

keyboard and your understanding of the editing features, until you cannot understand what all the fuss was about! However, until then it is wise to recall the advice given at the start of this chapter and try not to let the frustration produced by typing errors interfere with your understanding of computing in general and BASIC in particular.

Chapter Three

First Steps – Variables, PRINT, LET and INPUT

A program is a list of instructions that your computer can carry out. The question that this poses is what sort of instructions can you use in a computer program? It is clear that instructions like 'go and make a cup of tea' are too vague for anything other than a human to cope with! Instructions used in a computer program must be precise. They have to specify exactly what must be done and perhaps less obviously they have to specify what it has to be done to. In other words, a computer instruction tells the computer what to do and what to do it to. In this chapter we will look at the simplest 'objects' in BASIC and some very simple things that you can do with them.

Variables

The idea of a *variable* is the most important single idea in programming. A variable is an area of computer memory that is used to store information. This sounds like an easy idea but it has one or two subtle points. If you are going to store information in an area of memory you are going to need some way of referring to it. You're going to have to give it a name! This is not such an unusual idea if you think about other, more traditional, ways of storing data. For example, each file in a filing cabinet is normally given a name that identifies it and it alone. Just think of the confusion of asking for a file if two files had the same name! It is just the same with BASIC. An area of memory that is used to store information, a variable, must be given a unique name that can be used to refer to it. The only additional difficulty with a BASIC variable is that you must also define what sort of objects you are going to store in the memory area. One reason for this (we will meet others later) is that the amount of memory set aside to store the information depends on its type. For the time being the only sort of information that we will

store in memory will be numbers of any type. A variable that is used to store a number is called a *numeric variable* or a *simple variable*.

You cannot give a variable any name that takes your fancy, because this would lead to confusion on the computer's part. For example, suppose you gave the name 'I' to a variable. How would the Dragon know the difference between the variable I and the number 1? In the case of the Dragon you can give a simple variable a name of any length as long as it starts with a letter and thereafter uses only letters and digits. You can also insert spaces anywhere in the name to make it more readable, but the Dragon ignores them. The Dragon will also ignore everything after the first two letters! In other words as far as the Dragon is concerned SU, SUM and SUMMER are all the same name! So remember – the first two letters are for the Dragon and the rest for you to read. There is another restriction on variable names that is more difficult to describe. If you use a variable name LIST, for example, how is the Dragon to tell the difference between LIST the variable and LIST the command to list a program? To avoid this confusion the Dragon will not allow you to use any name that starts with one of its command words. The trouble is, of course, that you do not know all of the Dragon's command words at this stage! In practice all you can do is to look out for any variable names that the Dragon objects to by giving you an error message and change them to something else. Here are some examples of simple variable names that are allowed:

```
SUM
SCORE, SCORETOTAL, SCORE TOTAL
THIS IS A VERY LONG NAME
COUNT1
DAY2MONTH3YEAR83
```

Notice that both the versions of SCORETOTAL are treated as the same name. It is often difficult to think up names for variables that suggest the nature of the information to be stored in them, but it is well worth doing. If you come back to read a program after a long time clear, obvious variable names can make it a lot easier to re-understand your own program! Even so you should try to avoid very long variable names – they can be very boring to type out over and over again in a program. Some examples of names that the Dragon would not allow are:

<i>Name</i>	<i>Reason for rejection</i>
IDAY	starts with a number
*DATE	starts with an * which is not a letter
DATE*	contains * which is not a letter and is not a digit
ANSWER?	? is not a letter or a digit
OVER-TIME	- is not a letter or a digit
TOTAL	TO is a Dragon command

Storing things in variables – LET

Now that we know about variables and how to give them names it is time to discover how to store information in them. This can be done using the BASIC command LET. For example,

```
10 LET SUM=56
```

will store the number 56 in an area of memory called 'SUM'. If you recall, lines of a program are entered with line numbers that control their order in the list of instructions that make up the program.

This example is in fact our first program! If you enter it exactly as written, nothing will happen until you type RUN, when the Dragon will start obeying the list of commands. In this case there is only one command and this is very easy to obey – the number 56 is stored in an area of memory called 'SUM'. If you think about it just a little, more than this has happened. Before the program was run there was no area of memory called 'SUM' to store 56 in! When the Dragon comes across the name of a variable that you wish to use to store something in, it checks to see if it already exists, and if doesn't it sets aside an area of memory of the right size and remembers its new name. So this innocent single-line program has two effects – it creates the variable called 'SUM' and then it stores the number 56 in it.

The LET statement is such a common statement in BASIC that to save space and typing you can leave the word LET out. So the line of the program could equally well have been written

```
10 SUM=56
```

In future examples the word LET will be omitted from LET statements!

Finding out what's in a variable – PRINT

The one-line example in the previous section is a little disappointing because we have to take on trust that the Dragon has actually stored the number 56 in a variable called 'SUM'. What we need is a command that will make the Dragon find the variable and print its contents on the TV screen. The BASIC command with this effect is, most reasonably, called PRINT. If you add a new line, numbered line 20, to the previous example you will have the following two-line program:

```
10 SUM=56  
20 PRINT SUM
```

If you RUN this program you will be pleased to find that the number 56 is printed on your TV screen on the next free line.

It is important that at this point you understand exactly what is happening as a result of this two-line program. Later on, when you have absorbed BASIC almost as a second language, you understand what is going on without even thinking about it, but for now it is all too easy to read this two-line program and think you understand it because it *sounds* OK! So, to recap what we have already learnt, the first line creates a variable called 'SUM' and stores 56 in it. The second line finds the area of memory with the name 'SUM' and prints what is stored in it on the screen. (Notice that it is easy for a beginner to think that PRINT SUM would print the word SUM on the screen. So, if you already understand why this interpretation is incorrect you are no longer a beginner!)

For the PRINT statement to work it has to be possible for the Dragon to find the variable that it refers to. If for some reason you try to print a variable that hasn't been created then the Dragon will obligingly create it for you and store zero in it! To see this, remove line 10 (by simply typing 10 and ENTER) and RUN the program again. Although the Dragon looks after you in this way if you forget to create a variable, it is better not to fall into the habit of relying on it – other machines and other versions of BASIC are not so forgiving!

Arithmetic

Our programs are slowly becoming more interesting, but they are still a long way from being useful. We can now store numbers in

variables and print out what is stored in any variable, but so what! To be of any use we have to be able to change what is stored in a variable and print out something that we regard as an answer. The key to doing this lies in the idea of an *arithmetic expression*. An arithmetic expression is nothing more than a piece of arithmetic that you haven't yet worked out. For example, $3+6$ is an arithmetic expression that works out or *evaluates* to 9. You can write an arithmetic expression on the right-hand side of the equals sign in a LET statement with the effect that the Dragon will evaluate the expression and store the result in the variable. For example try

```
10 SUM=3+6
20 PRINT SUM
```

You will see 9 printed on the screen. As promised, the Dragon has evaluated the expression and stored the result in 'SUM'.

As with most things to do with computers, there are rules governing what makes a correct expression. You can use the four operations that you should be familiar with from simple arithmetic. Addition and subtraction are indicated by the usual symbols $+$ and $-$, but multiplication and division use the symbols $*$ and $/$. The reason for using $*$ to mean multiply instead of a cross is that the traditional symbol is too easy to confuse with the letter X. Some examples of correct arithmetic expressions are:

<i>Expression</i>	<i>Evaluates to</i>
$3+2$	5
$3*2$	6
$6/2$	3
$3+2-4$	1
$2.1+3.3$	5.4

Apart from the four usual operations of arithmetic there are two others that can be used on the Dragon – the *unary minus* and *raise to a power*. The unary minus sounds rather grand, but it is simply the normal subtraction sign used in front of a single number. For example, the $-$ in $3-2$ is the normal subtraction sign, but the $-$ used in -3 is the unary minus. Although the same sign is used in both cases, as we will see later they are treated slightly differently. The raise to a power sign is $\uparrow$. For example, $2\uparrow 2$ is read as 'two raised to the power of two', i.e. two squared or four. The raise to a power sign is not used very often and it is mentioned here more for completeness than for its importance.

Understanding expressions – the order of evaluation

Although the idea of an arithmetic expression seems straightforward, there is a hidden complication. For example, if you write the innocent looking expression $3+2*4$, does it mean ‘three plus two’ (i.e. five) ‘times four’ (answer twenty), or does it mean ‘three plus’ the answer to ‘two times four’ (i.e. ‘three plus eight’, answer eleven)? It may seem strange to you that there are two possible ways to work out this expression, because you may feel that one of the two methods is obviously correct and the other is equally obviously incorrect. However, even in arithmetic, there are no absolute answers! The correct interpretation is a matter of convention and isn’t something that is handed down from high. The question of whether we do the $+$ or the $*$ first in an expression like $3+2*4$ is settled by a general agreement that multiplication is more important than addition, and so it should be done first, making the correct answer eleven. This agreement that multiplication is more important than addition can be formalised in terms of assigning *priorities* to each operation and carrying out the operation with the highest priority first. The assignment of priorities can be extended to every operation that can be used in an expression (even some that we haven’t met as yet). The priorities that the Dragon uses to sort out the order in which arithmetic should be carried out are:

<i>Operation</i>	<i>Priority</i>	
↑	4	highest
Unary –	3	
*,/	2	
+,–	1	lowest

To evaluate an expression you should always work out the operators with the highest priority first. If two operators in an expression have the same priority then you should do the one furthest to the left first (i.e. in the absence of any other preference, you work from left to right).

All this may seem a little over-complicated just to carry out a little arithmetic, but it is necessary if you want to write unambiguous expressions. However, there is another way of specifying the order of evaluation that can be used to override the usual priorities – brackets (). It is a longstanding convention that any parts of an expression enclosed in brackets are carried out first. For example, although $3+2*4$ is 11, $(3+2)*4$ is 20. If you’re ever in any doubt

about how the Dragon will evaluate an expression, then put brackets around the parts that you want worked out first. Brackets sometimes waste time and effort but they can never cause trouble!

Variables and constants – the full expression

So far we have looked at arithmetic expressions involving only numbers, but there is no reason why we cannot use variables in expressions. If you write an expression such as SUM+3 the Dragon will find the variable called SUM and retrieve the number stored in it. It will then add three to this number. For example, if SUM had 32 stored in it, the expression SUM+3 would evaluate to 35. Notice that there is no suggestion that what is stored in the variable SUM is in any way altered. Its contents are simply used in the evaluation of an expression. A number such as 32 is known as a *constant* (because its value never changes!) and now we can see that an expression can be made up of variables and constants, with the arithmetic operators + - / * ↑. An expression always evaluates to a constant and it is this constant that is stored in a variable by a LET statement.

A short program

Using all that we have found out so far about constants, variables and expressions we can now write a short program that adds two numbers together:

```
10 N1=23.34
20 N2=44.32
30 ANSWER=N1+N2
40 PRINT ANSWER
```

If you enter and RUN this program you will see that the sum of the two numbers in lines 10 and 20 are printed by line 40. Although this is another simple example it demonstrates a wide variety of programming ideas. In lines 10 and 20 the by now familiar LET statement is used to store two constants in two variables. In line 30 the arithmetic expression N1+N2 is evaluated and the result is stored in a third variable ANSWER. Line 40 prints the contents of ANSWER on the screen. If you think this is easy, so far so good! Try changing lines 10 and 20 to add different numbers together and change line 30 to give you different arithmetic expressions.

Another way of altering variables – INPUT

In the previous example the two variables N1 and N2 had numbers stored in them by use of the LET statement. This is convenient unless we want to use the program many times with different values. As suggested, the only way that it is possible to change the values stored in the variables is to edit each line before running the program. Obviously what we need is a statement that will allow us to enter any value into the variable *while the program is running*. This is what the BASIC statement INPUT is for. For example try the following program:

```
10 N1=5
20 INPUT N2
30 ANSWER=N1+N2
40 PRINT ANSWER
```

When you run this you might be surprised to find that nothing much happens! Don't panic! What has happened is that line 10 was carried out and 5 was stored in the variable N1. Then the Dragon moved on to line 20 where it obeyed the command INPUT by waiting for you to type a number, and this is why the question mark was displayed. The Dragon is waiting for you to type a number and then press ENTER to signal that you have finished typing/correcting the number. It then stores the number that you have typed in the variable N2 and proceeds to the next instruction. So if you haven't already done so, run the program and type in a number of your choice. You will be pleased to see your number with five added to it printed in the usual place.

We now have two ways of storing numbers in a variable – LET and INPUT. It is important to understand the difference between the way LET and INPUT work. As in the case of LET, if a variable doesn't exist before its use in an INPUT statement the Dragon will create it. If you always want to store the same value or the result of an expression in a variable then use a LET statement. If you want to store a different value in a variable each time the program is run, then use an INPUT statement.

Variables and constants as expressions

One of the most powerful features of BASIC is the way that almost anywhere that you can use a constant or a variable you can use an

expression as well. For example, in the PRINT statement you can write

```
30 PRINT N1+N2
```

and the Dragon will evaluate the expression and print the result.

It is also true that the simplest forms of an expression are the constant and the variable. For example, the number 3 can be thought of either as a constant or as an extremely simple expression. Similarly, the variable SUM can also be thought of as an expression. So not only can you use an expression wherever you might use a variable or a constant, you can use a variable or a constant anywhere that you can use an expression! For example

```
LET N1=N2
```

and

```
PRINT 3
```

are both valid BASIC statements.

Describing BASIC

It is difficult to describe any language, and BASIC is no different. The trouble is that while it's easy to give an example of what is correct, it is difficult to explain all the possible correct variations. For example, at the start of this chapter the LET statement was introduced by

```
LET SUM=56
```

but this gave no hint that you could write things like

```
LET SUM=NUMBER
```

or

```
LET SUM=N1+N2
```

To try to overcome this difficulty it is usual to give a definition involving the general types of things that a statement allows. For example, the general form of the LET statement can be written as

```
LET 'simple variable' = 'arithmetic expression'
```

where the things between the single quotes are not to be taken literally but replaced by an example of the stated type. So in a real

LET statement ‘simple variable’ would be replaced by a variable name such as COUNT, SUM, NUMBER, etc.

Throughout the rest of this book BASIC statements will be introduced by examples and then defined in the same way as the LET statement above. As we learn more about a statement it may prove necessary to redefine it to include a wider range of features. So far the two other BASIC statements that we have introduced, PRINT and INPUT, can be defined as follows:

PRINT ‘arithmetic expression’

and

INPUT ‘simple variable’

but as we shall see later these are not the final definitions!

INPUT prompting

Although we can now use INPUT to store information in variables, the way the Dragon just stops and waits for someone to type a number in is a little unsatisfactory. What is required is the ability to print a message on the screen saying something like ‘TYPE IN A NUMBER NOW’ or ‘WHAT IS YOUR NUMBER?’. Such a message is often called an *input prompt* and BASIC provides two similar ways to print such messages. Try the following short program

```
10 PRINT "THIS IS A PROMPT"
20 INPUT "WHAT IS YOUR NUMBER";N1
```

Line 10 will display ‘THIS IS A PROMPT’ on the screen and line 20 displays ‘WHAT IS YOUR NUMBER ?’ just below it, and then waits for you to type a number. In both cases the characters printed on the screen are the ones inside the double quotation marks. A set of characters in double quotes is known as a *literal string*, or simply as a *string*. You can use either PRINT or INPUT to produce a prompt message on the screen, depending on which is more convenient. As an example of the use of both, consider the following version of the number addition program given earlier.

```
10 PRINT "WHAT IS YOUR FIRST NUMBER ?"
20 INPUT N1
30 INPUT "WHAT IS YOUR SECOND NUMBER";N2
40 PRINT N1+N2
```

Mixed PRINT

The ability to print messages on the screen is clearly a very useful facility for other things than just printing prompts. For example, in the last program it would have been better to print a message saying that the number about to be printed was the sum of the two numbers. You can in fact use a single PRINT statement to print more than one thing at a time. For example, change line 40 in the previous program to

```
40 PRINT N1;" + ";N2;" = ";N1+N2
```

and you will see that the contents of 'N1' are printed, then a space and a plus sign followed by another space, then the contents of 'N2' followed by an equals sign with a space on either side of it, and the answer. You can consider this PRINT statement as a list of items to be printed, each item in the list being separated by a semi-colon and printed in the next free printing position. Our general definition of PRINT can now be updated to read

PRINT 'print list'

where 'print list' is a list of items separated by semi-colons. The items can be either expressions or strings. Each PRINT statement starts printing on a new line. In Chapter Five we will return to the definition of the 'print list' and expand it to include ways of formatting the information on the screen, but this simple version of the 'print list' will satisfy all our requirements until then.

Some sample programs

Even with so little BASIC it is possible to write some useful, if simple, programs. For example, if you want regularly to work out how many dollars you would buy for a number of pounds then a currency conversion program would be useful. The overall outline of this program is easy to explain, as follows:

Ask the user for the conversion rate.

Ask for the number of pounds to be used to buy dollars.

Print number of pounds times conversion rate.

By now you should realise that this program is a simple one for your Dragon, so before you look at the version given below, try to

write your own version. Remember to include input prompting and explanatory messages. There is no one perfect way to write any program, so don't worry if your version is different to the one given. It could well turn out to be better!

```
10 PRINT "POUNDS TO DOLLAR CONVERSION"
20 INPUT "WHAT IS THE CONVERSION RATE";RATE
30 INPUT "HOW MANY POUNDS DO YOU WANT TO
    SPEND";AMOUNT
40 PRINT
50 PRINT "FOR ";AMOUNT;" POUNDS "
60 PRINT "YOU CAN BUY ";RATE*AMOUNT;
    " DOLLARS"
```

Line 10 simply prints a title for the program. Lines 20 and 30 prompt for and accept the necessary input. Notice how line 40 is used to print a blank line to space the output into an input and a result section. Lines 50 to 60 print the answer with some explanation.

As another example, consider the problem of calculating the stopping distance of a car travelling at any speed in miles per hour. The main problem in writing this program is knowing how to calculate the stopping distance. It is important to realise at this stage that no computer can calculate something unless you can explain to it *how* to do the calculation. Looking at the highway code reveals that the stopping distance is made up from two components – a thinking distance and a braking distance. The thinking distance in feet is roughly the same as the speed in mph. The braking distance is a little more complicated and is given by the square of the speed divided by 20. However, both of these quantities are very easy for the Dragon to calculate, so to add to the usefulness of this short program it is reasonable to print out the thinking distance, the braking distance, the overall distance and how many car lengths it takes to come to a stop. This last calculation is based on the fact that the average (UK) car is 14 ft long. The program is now easy to write.

```
10 PRINT "STOPPING DISTANCE"
20 PRINT
30 INPUT "SPEED IN MPH=";SPEED
40 PRINT "AT ";SPEED;"MPH"
50 PRINT
60 PRINT "THINKING DISTANCE=";SPEED;" FEET"
70 BRAKDIST=SPEED*SPEED/20
80 PRINT "BRAKING DISTANCE=";BRAKDIST;
    " FEET"
```



```

90 PRINT "OVERALL DISTANCE=";SPEED+
  BRAKDIST;" FEET"
100 PRINT
110 PRINT "WHICH IS ";(SPEED+BRAKDIST)/14;
  " CAR LENGTHS"

```

Line 10 simply prints a title for the program. Line 20 prompts for the only information needed by this program – the speed in mph. This is stored in the variable SPEED. Lines 30 to 40 start giving the answer to the user by printing the speed that the calculation is for and leaving some space. Line 60 prints the thinking distance, which requires no calculation as it is numerically the same as the speed in mph. The braking distance is calculated by line 70 and printed by line 80. The overall distance is calculated and printed by line 90. Notice that this is an example of using an expression in a PRINT statement. Line 100 calculates and prints the overall stopping distance in terms of car lengths. Notice that you have to put brackets around the addition to make sure that it is carried out first.

The final example in this chapter is a sizeable and really useful program if you are interested in DIY. A very common problem is that of estimating how much sand, aggregate and cement you should buy to cast a slab of concrete. To help with this difficult task the following program is a ‘Concrete Calculator’. Once again our first problem is knowing how to do the calculation before we begin writing the program. Looking up the relevant information in a book on building reveals the following – 1 cubic metre of concrete made up from a 1 part cement, x parts of sand and y parts of aggregate (i.e. a 1: x : y) mix needs

$$\frac{1}{0.025*(1+x+y)}$$

bags of cement and

$$\frac{x*1.5}{(1+x+y)}$$

cubic metres of sand and

$$\frac{y*1.5}{(1+x+y)}$$

cubic metres of aggregate. (It’s not too important to understand why these equations work, it is often the case that a program is written from information that the programmer doesn’t fully understand –

and why should it be otherwise!)

Converting this information into a program is once again relatively easy. The first part of the program should ask for the dimensions of the concrete slab and then calculate its volume. The program should then ask for the mixture ratio, and using the equations given above work out the number of bags of cement and the volume of sand and aggregate required. Finally, the results should be printed out in a form that the user will find acceptable. The details of the program are:

```

10 PRINT "CONCRETE CALCULATOR"
20 PRINT
30 INPUT "WHAT IS THE THICKNESS IN MM";THICK
40 INPUT "WHAT IS THE LENGTH IN M";MLEN
50 INPUT "WHAT IS THE WIDTH IN M";WIDTH
60 PRINT
70 VOL=THICK*.001*MLEN*WIDTH
80 PRINT "TOTAL VOLUME = ";VOL;" CUBIC M"
90 PRINT
100 INPUT "HOW MANY PARTS OF SAND TO ONE OF
    CEMENT";SANDPART
110 INPUT "HOW MANY PARTS OF AGGREGATE TO
    ONE OF CEMENT";AGGPART
120 SUM=1+SANDPART+AGGPART
130 CEMENT=VOL/SUM/.025
140 TSAND=VOL*SANDPART*1.5/SUM
150 TAGG=VOL*AGGPART*1.5/SUM
160 PRINT "USING A 1:";SANDPART;":";AGGPART;
    " MIX"
170 PRINT "YOU NEED -"
180 PRINT CEMENT;" BAGS OF CEMENT"
190 PRINT TSAND;" CUBIC M OF SAND"
200 PRINT "AND ";TAGG;" CUBIC M OF AGGREGATE"

```

Lines 30–80 ask for the dimensions of the slab and both calculate and print the total volume. Lines 100 and 110 ask for the ratio of the mix and line 120 calculates $1+x+y$ used in all of the following calculations. The most interesting lines of the whole program are 130–200. The first few lines (130–150) carry out the main calculations and the last section (160–200) print the results. If you look carefully at the calculations you will see that an arithmetic expression in BASIC doesn't always look like the equation that it

comes from. For example, you might fall into the trap of writing

$$\frac{1}{0.025*(1+x+y)}$$

as

$$1/0.025*(1+x+y)$$

but the result of this BASIC arithmetic expression is given by dividing 1 by 0.025 and then multiplying the answer by $(1+x+y)$, i.e. it works out

$$\frac{1}{0.025} * (1+x+y)$$

rather than the equation that we are interested in. The correct expression is

$$1/0.025/(1+x+y)$$

or if you want to use extra brackets to clarify matters

$$1/(0.025*(1+x+y))$$

In the section that prints the results notice the way that each answer is *embedded* in the printed message. Remember that you do not have to write programs that produce results in a standard way – experiment with what looks good and seems natural.

Chapter Four

Looping and Choice – the Flow of Control

So far we have written programs that are lists of instructions carried out one at a time from the top to the bottom. Although it is possible to write useful programs using nothing more, programming becomes really interesting only when you can change the order in which instructions are carried out.

The flow of control

If you look at any of the example programs at the end of Chapter Three it should be possible for you to follow through with your finger the order in which the instructions would be obeyed by the Dragon. You can think of this as tracing the *flow of control* through the program. Each instruction has its turn at governing or controlling what the Dragon is doing and it then passes control to the next instruction. In the absence of any other information the next instruction is taken to be the next line down. So the *default* flow of control is a line starting at the top of the program and finishing at the bottom. The following is a very simple program that demonstrates this default condition.

```
1Ø INPUT A
2Ø PRINT A
3Ø INPUT B
4Ø PRINT B
```



Fig. 4.1.

Looping - the GOTO

BASIC provides a single statement for changing the default flow

of control – the GOTO statement. Try the following program.

```
10 TEST=0
20 PRINT TEST
30 GOTO 20
```

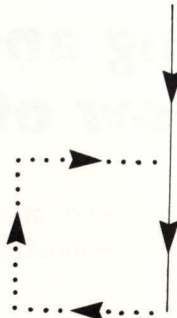


Fig. 4.2.

If you run this program you will see the screen fill with zeros (one to each line). After this you may not be able to see the zeros being printed but they still are! The only way that you can stop the program is to press the **BREAK** key.

This program is our first example of a *loop*. Tracing the flow of control through the program soon shows why the word ‘loop’ is appropriate. First line 10 stores zero in the variable TEST, then line 20 prints the contents of TEST. Line 30 is new in that it uses the command GOTO, but its meaning should be clear from just reading it. The statement GOTO 20 causes the Dragon to obey line 20 as the next instruction. So after line 30 control is transferred to line 20 and the contents of variable TEST are printed on the screen for a second time. After this line 30 is again carried out. This transfers control back to line 20 and so on ... The program calls for the repetition of lines 20 and 30 forever and tracing the flow of control shows it to take the form of a loop. It is not a very useful loop however, and because it has no predetermined stopping point it is usually called an *infinite loop*.

A GOTO statement can be used to force the Dragon to carry out any instruction next. Its general form is

GOTO ‘line number’

for example

GOTO 20

You may be wondering what happens if you write something like GOTO 50 and line 50 doesn’t exist. In this case the Dragon will stop and give an error message ‘UL ERROR’ indicating that you are trying to transfer control to a non-existent line.

Although the example of the infinite loop serves to introduce the

idea of transferring control to a different point in the program, it doesn't really indicate the sort of thing that a loop is used for. An important idea in programming is the repeating of series of operations. This is often called *iteration*. For example, the instruction `COUNT=COUNT+1` simply adds one to the contents of the variable `COUNT` and stores the answer back in `COUNT`. In other words it increases the number stored in `COUNT` by one. If you repeat this operation by using a `GOTO` you have something more than adding one to a variable – you have a program that counts! Try the following:

```
1Ø COUNT=Ø
2Ø COUNT=COUNT+1
3Ø PRINT COUNT
4Ø GOTO 2Ø
```

You will see the screen fill with numbers without end. If you look carefully you should be able to see that the numbers are increasing by one each time – your program is counting. Notice that by adding a `GOTO` to an instruction that adds one to a variable we actually seem to produce a program that does a bit more than just count. In fact, we generate a *sequence* of numbers. This is much more the flavour of real programming than the one-after-the-other programs in Chapter Three. To see looping doing something a little more useful try:

```
1Ø COUNT=Ø
2Ø PRINT "X=";COUNT;" X SQUARED=";COUNT*COUNT
3Ø COUNT=COUNT+1
4Ø GOTO 2Ø
```

This program will print out two lists of numbers, the second being the square of the first.

Even if you are very familiar with the idea of a loop you can be confused about what the current value of a variable is at any point in the loop. For example, in the case of the counting loop program, the first value of `COUNT` that was printed was one, but in the squares program the first value was zero. This difference is simply due to *where* in the program the line that adds one to `COUNT` is placed, and also what value '`COUNT`' is set to before the loop starts. If you change line 1Ø in the squares program to read `1Ø COUNT=1` then the first value to be printed will be one.

Understanding what goes on in a loop gets easier with practice, but it's all a matter of following through the action of the program clearly and without rushing.

Choice and conditions – the IF statement

Although the GOTO is a very useful statement, the only thing that you can do with it is to form infinite loops. This is fine for simple things like printing tables of values, but is a bit too crude for many applications. What we are lacking is a statement that will stop the loop when a *condition* is satisfied. For example, suppose we want to write a program that will add ten numbers together and then print out the answer. At the moment the best that we can do is to read in each number in turn and add it to a running total which we print out each time, as in the following example:

```

1Ø SUM=Ø
2Ø COUNT=Ø
3Ø INPUT "NUMBER = "; NUMBER
4Ø SUM=SUM+NUMBER
5Ø COUNT=COUNT+1
6Ø PRINT COUNT;" TOTAL ="; SUM
7Ø GOTO 3Ø

```

If you run this program you will find that it produces rather a lot of output that you don't need. What we really want to do is read in the ten numbers, keep a running sum and only print out the answer at the end. This can be achieved using the IF statement:

```

1Ø SUM=Ø
2Ø COUNT=Ø
3Ø INPUT "NUMBER ="; NUMBER
4Ø SUM=SUM+NUMBER
5Ø COUNT=COUNT+1
6Ø IF COUNT=1Ø THEN GOTO 8Ø
7Ø GOTO 3Ø
8Ø PRINT "TOTAL = ";SUM

```

The only difference between this and the first program to add ten numbers together is the use of the IF statement in line 6Ø. Each time through the loop formed by the lines from 3Ø to 7Ø the IF statement is obeyed. This takes the form of comparing the contents of the variable COUNT to 1Ø. If it isn't equal to 1Ø, control passes to the next statement, i.e. line 7Ø, and the GOTO following the THEN has no effect. However if COUNT is equal to 1Ø the GOTO following the THEN is carried out and control passes to line 8Ø, printing the answer and ending the loop.

There are many ways of using the IF..THEN GOTO statement to

alter the flow of control, depending on whether or not a condition is true. Before we can go on to investigate the sort of thing that can be done with IF we have to find out what types of conditions we can use.

All of the conditions that you can use in an IF statement take a very simple form:

‘arithmetic expression1’ ‘relation’ ‘arithmetic expression2’

We already know what an arithmetic expression is, so the only new element is the *relation*. In BASIC there are six relations:

<i>Relation</i>	<i>Meaning</i>
=	equals
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

The meaning in BASIC of each of these relations is the same as their normal meaning.

Conditions are very often called *conditional expressions* and a condition is like an expression, in that it evaluates to a value but in its case there are only two possible results, true or false. It is important to read conditions in the right way to avoid confusion. For example, the condition COUNT=3 is *not* an instruction to make COUNT equal to 3, it is a *question* about what is stored in COUNT. If the value stored in COUNT is 3 then COUNT=3 is true, but if the value is anything other than 3 then COUNT=3 is false. Thus the Dragon uses the sign ‘=’ in two different ways – as an instruction to store a value in a variable and as a relation in a condition. The only way to tell which is the correct meaning in any case is to look at the rest of the instruction. Some examples of conditions are:

COUNT<>4	false if COUNT is 4 otherwise true
COUNT*2>10	true if COUNT*2 is greater than 10, otherwise false
3>1	ALWAYS true
1>6	ALWAYS false

To reinforce the idea that a condition is an expression that evaluates to one of two values (true or false) it is worth saying that

the Dragon represents both values as numbers. True is represented by -1 and false is represented as $\emptyset$. In other words, if a condition is true it evaluates to -1 and if it is false it evaluates to $\emptyset$. To prove that this is the case try the following program:

```
1 $\emptyset$  INPUT "FIRST NUMBER";A
2 $\emptyset$  INPUT "SECOND NUMBER";B
3 $\emptyset$  PRINT "THE RESULT OF";A;">";B;" IS ";A>B
4 $\emptyset$  GOTO 1 $\emptyset$ 
```

You will find that either a -1 or a $\emptyset$ will be printed, depending on whether $A>B$ is true or false in the case of the numbers you typed. You might be surprised that you can write a condition in a PRINT statement where you would normally write an arithmetic expression. This is simply another reflection of the fact that a condition is an expression just like an arithmetic expression and it can be used anywhere that an arithmetic expression can. Indeed, you can mix conditional and arithmetic expressions with no problems as long as you pay attention to the order in which they are evaluated. Any arithmetic is evaluated before any conditional expressions. For example

```
PRINT (2<1)+(3>1)+(3=2+1)
```

will print -2 on the screen because $(2<1)$ is false and evaluates to $\emptyset$, $(3>1)$ is true and evaluates to -1 and $(3=2+1)$ is also true and evaluates to -1 , which gives $\emptyset-1-1$, i.e. -2 . (Notice that the conditions all have to be enclosed in brackets to stop the Dragon trying to do the arithmetic first!) This sort of thing is fairly advanced BASIC so don't worry if you don't understand it fully – what is important is that you understand that something like $2=3$ is an expression and evaluates to true or false rather than an instruction to do some operation.

We have taken a slight detour around the subject of the IF statement to examine the idea of a conditional expression, but armed with this new information the remainder of this chapter should seem easier. The general form of the IF.. THEN GOTO statement is

```
IF 'conditional expression' THEN GOTO 'line number'
```

If the conditional expression evaluates to -1 , or true, then the GOTO following the THEN is obeyed. If the conditional expression evaluates to $\emptyset$, or false, then the statement following the IF is obeyed.

There are so many ways of using the IF.. THEN GOTO apart from breaking out of loops that it is difficult to give examples of everything. But if you understand the ideas of flow of control and the way that IF and GOTO can be used to change it then you should have no trouble in understanding the examples in the rest of this book.

Using IF

The only way to find out how useful IF can be is to write your own programs that use it. In this way you'll slowly pick up all the standard ways that you can change the flow of control depending on the result of a conditional expression. However, it might help to speed up this process and avoid clumsy ways of using the IF statement to give examples of some of the most common ways that it is used.

An IF statement can be used to skip a section of program.

```

1Ø INPUT A
2Ø IF A>Ø THEN GOTO 4Ø
3Ø A=-A
4Ø PRINT A
    
```

This short program refuses to let you enter a negative number! The IF statement in line 2Ø checks to see if the number in A is greater than zero and if so control passes to line 4Ø, effectively skipping line 3Ø. If A>Ø is false, line 3Ø changes the sign of the contents of A. In another program a different list of statements might be skipped according to some other condition. The *shape* of the skip is depicted in Fig. 4.3, which illustrates how the result of the condition decides whether or not the list of instructions is carried out or skipped.

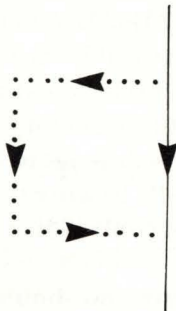


Fig. 4.3.

An extension of the idea of skipping some lines of BASIC is to choose between two different lists of commands. In this case it is easier to understand the idea after looking at the shape of the flow of control (see Fig. 4.4). Which list is carried out depends on the condition used in the IF statement. The reason why it is useful to

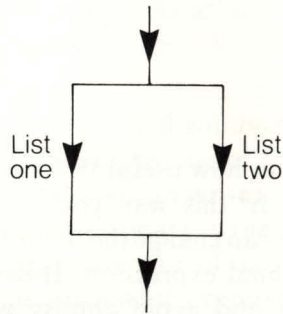


Fig. 4.4.

look at the flow of control diagram before looking at an example of an IF statement to select between two sets of instructions is that it is a little difficult to see the simple division into two in the BASIC. Consider the following program:

```

10 INPUT A
20 IF A<0 THEN GOTO 60
30 PRINT "A IS POSITIVE"
40 B=A
50 GOTO 80
60 PRINT "A IS NEGATIVE"
70 B=-A
80 PRINT A,B
  
```

According to the value of A either lines 30, 40 and 50 are obeyed, or lines 60 and 70 are. The *division* point in the diagram corresponds to the IF statement itself (line 20). The *join up* point is line 80, because this is the first statement that will be carried out no matter which of the two lists of instructions is selected. If you try to superimpose the flow of control diagram on the program you will see that, although it represents what happens, it is difficult to fit it. The reason for this is that it is impossible in BASIC to write the two alternative lists next to each other, so the flow of control diagram is more like the one in Fig. 4.5.

If you look at this diagram you should be able to see that it is a 'mangled' form of the previous one. Using the IF to select between

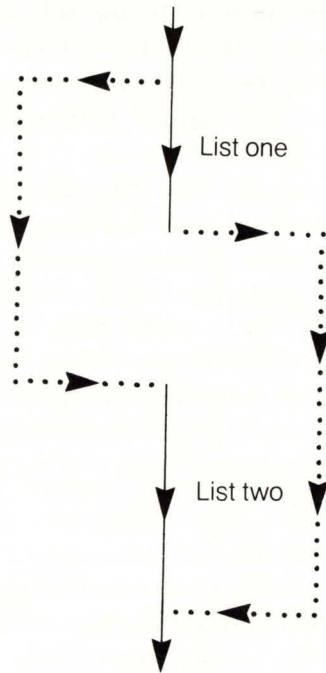


Fig. 4.5.

two alternatives is easier to understand from the first diagram, but the second corresponds more closely to reality. This is the most complicated flow of control diagram that you will find in BASIC,

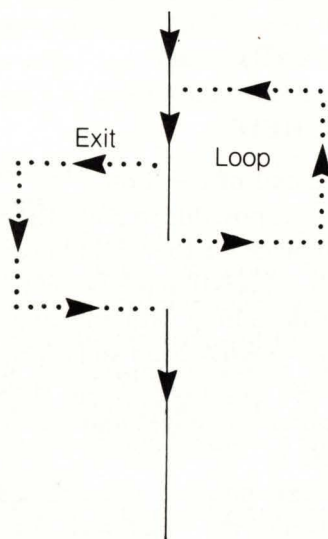


Fig. 4.6.

and to make it slightly simpler Dragon BASIC includes a special form of the IF statment – IF..THEN..ELSE – which will choose between two alternatives. This is a simple statement but we need a few more ideas before it becomes useful to us, so a discussion is left to the end of this chapter.

We have already met the only other important use of the IF statement, that of breaking out of a loop. The flow of control diagram for that circumstance can be visualised as shown in Fig. 4.6. You should be able to see the familiar shape of an infinite loop. The path that leads out of the loop and back to the normal flow of control corresponds to a GOTO taken when the condition in the IF statement is true. Notice that the point at which the IF statement breaks out of the loop can be placed anywhere. In other words, you can break out of a loop anywhere from the first statement to the last. For example

```
10 A=0
20 A=A+1
30 IF A=20 THEN GOTO 60
40 PRINT A
50 GOTO 20
60 PRINT "FINISHED"
```

has its exit point in the middle and

```
10 A=0
20 A=A+1
30 PRINT A
40 IF A=20 THEN GOTO 60
50 GOTO 20
60 PRINT "FINISHED"
```

has its exit point at the end of the loop.

In general, although it is possible to place the exit point of a loop anywhere, it is better to place it either right at the beginning or right at the end. The reason for this is that it is better to avoid carrying out *part* of a loop. (Technically a loop with only one exit point placed at the beginning is known as a *while* loop and a loop with only one exit point at the end is known as an *until* loop. However, these names come from other computer languages and are not important when programming in BASIC.)

A loop that has its exit point at the end can be simplified by turning the condition round the other way, for example:

```

1Ø A=Ø
2Ø A=A+1
3Ø PRINT A
4Ø IF A=2Ø THEN GOTO 6Ø
5Ø GOTO 2Ø
6Ø PRINT "FINISHED"

```

can be turned into

```

1Ø A=Ø
2Ø A=A+1
3Ø PRINT A
4Ø IF A<>2Ø THEN GOTO 2Ø
5Ø PRINT "FINISHED"

```

The loop in the first example comes to an end when A is equal to 2Ø and line 5Ø contains the GOTO that otherwise makes the loop continue. In the second example the GOTO has been eliminated by turning the condition around to make the loop continue when A is NOT equal to 2Ø. After seeing this sort of thing a few times you will adopt the shorter, neater form without thinking, but it is worth occasionally remembering where it derives from.

As stated at the beginning of this section, there are many ways of using an IF statement to alter the flow of control of a program, and you shouldn't feel restricted to just those that have been introduced in this section. However, it is wise not to experiment too much, because the more ways that you use the IF statement the more difficult your program will be to understand. In essence, the rule is to make the flow of control diagram as simple as possible so that your programs will be easy to read, understand and debug.

The FOR statement

Apart from the position of the exit point, loops differ in two other ways. All loops continue until a condition is satisfied, but in many cases this is equivalent to carrying out the loop a fixed number of times. For example, if you wanted to print the word 'HELLO' on the screen five times you could do it using

```

1Ø A=1
2Ø PRINT "HELLO"
3Ø A=A+1
4Ø IF A=<5 THEN GOTO 2Ø

```


However, this is such a common situation that BASIC provides two extra statements – FOR and NEXT – to make repeating lists of statements easier.

Using FOR and NEXT, the program that prints ‘HELLO’ on the screen five times can be written

```
10 FOR A=1 TO 5
20 PRINT "HELLO"
30 NEXT A
```

The meaning of the FOR and NEXT should be clear from the program. The variable A is used to count the number of times that the loop has been obeyed in the same way as in the earlier example. The difference is that everything is done automatically. The FOR statement first sets A to one. Each time the NEXT statement is carried out one is added to A and, as long as its value hasn’t exceeded 5, control is transferred back to line 20 (i.e. there is an implied GOTO 20). The result is that the PRINT statement at line 20 is executed five times before control passes on to the statement following the NEXT.

The general form of the FOR NEXT loop is

```
FOR ‘index variable’=‘start value’ TO ‘end value’
. . . . .
NEXT ‘index variable’
```

The ‘index variable’ is initially set to the ‘start value’. Each time NEXT is reached the ‘index variable’ is increased by one and as long as the value hasn’t exceeded the ‘end value’, control is transferred to the statement just after the FOR. To make sure that you understand exactly what a FOR loop can do, try the following examples.

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

10 FOR Z=100 TO 110
20 PRINT Z
30 NEXT Z
```

In general both the ‘start value’ and the ‘end value’ can be full arithmetic expressions, but it is important to realise that these are only evaluated *once* at the start of the loop. This becomes clear if you think of the FOR statement as only being carried out once at the start of the loop. The value of the ‘index variable’ can be used in

arithmetic expressions during the loop, but *its value must not be changed*. In other words, you can use the ‘index variable’ in a LET statement on the right-hand side of an = sign but not on the left. For example, the following short program will print a multiplication table.

```
10 INPUT "WHICH TABLE – ENTER 2 FOR TWO TIMES
    TABLE ETC ";T
20 INPUT "STARTING AT ";S
30 INPUT "ENDING AT ";E
40 FOR I=S TO E
50 PRINT I;" X ";T;" = ";I*T
60 NEXT I
```

Notice that both the start and end values of the FOR loop are arithmetic expressions, simple variables in fact! Also note the use of the ‘index variable’ I in line 50.

The simple FOR loop serves for most purposes. However, there is a slightly more advanced form that is occasionally useful and is certainly worth knowing about. In the simple FOR loop each time through the loop one was added to the value of the index variable. This is sensible if you are using the index variable to count the number of times that the loop has been carried out. However, it is sometimes the case that a calculation carried out inside the loop needs a value that changes by something other than one each time through the loop. This is catered for by the addition of the BASIC statement STEP, the general form of which is

```
FOR ‘index variable’=‘start value’ TO ‘end value’ STEP
‘increment’
```

The ‘increment’ specified following STEP is the amount that is added to the index variable each time through the loop. So the simple FOR loop is equivalent to STEP 1. The only thing that you have to be careful of when using FOR STEP is that you know when the loop will come to an end. The rule is that the loop terminates when the value of the index *exceeds* the ‘finish’ value. So the value of an index variable in a FOR loop can never become larger than the ‘finish value’. To see this in action try the following examples:

```
10 FOR A=.4 TO 10 STEP .01
20 PRINT A
30 NEXT A
```

or

```
10 FOR A=1 TO 100 STEP 25
20 PRINT A
30 NEXT A
```

The value of the 'increment' following STEP can be negative, in which case it is better called a *decrement*. For example:

```
10 FOR A=100 TO 0 STEP -15
20 PRINT A
30 NEXT A
```

You may have some slight difficulty in working out when this and similar loops end. However, the rule is almost the same as for a positive increment. Each time through the loop the value of the index variable decreases by the increment and the loop ends when its value first drops below the 'end' value.

As well as having a negative increment, it is possible for either the 'start' or the 'end' value to be negative, and this is where things can be confusing. Consider this short program:

```
10 FOR A=-100 TO 50 STEP 10
20 PRINT A
30 NEXT A
```

At first sight it may not seem to make sense. However, if you keep a cool head then you should be able to see that the same rules apply. The value of the increment is added to the index variable each time through the loop, until its value exceeds the 'end' value if the increment is positive, or is less than the 'end' value if the increment is negative.

Using the FOR loop

There are one or two rules that govern the use of FOR loops. As you can use any valid BASIC statement within a FOR loop it is possible to use a GOTO or an IF to leave a FOR loop before it is finished. However, this is to be avoided because your Dragon will eventually get round to giving you an error message if you do it too often! Do not leave FOR loops until they are finished!

It is often the case that FOR loops are used in combination. This is easy to understand as long as you think logically about the way each loop works. For example


```

10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A,B
40 NEXT B
50 NEXT A

```

is correct because the FOR loop formed by lines 20, 30 and 40 is completely contained within the FOR loop starting at line 10 and ending at line 50. This means that the ‘inner’ loop is carried out each time through the ‘outer’ loop. It is all too easy to make a slight mistake and end up with

```

10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A,B
40 NEXT A
50 NEXT B

```

which will give you an error message.

IF .. THEN and the colon

There is an even more general version of the IF statement than the one we have examined so far. As well as being able to follow the THEN by the BASIC command GOTO, you can in fact use any valid BASIC command. For example:

```

10 INPUT A
20 IF A<0 THEN PRINT "A IS NEGATIVE"
30 IF A=0 THEN PRINT "A IS ZERO"
40 IF A>0 THEN PRINT "A IS POSITIVE"
50 GOTO 10

```

The best way to think of this is as a sort of easier version of the IF to skip an instruction. Only in this case the instruction is only carried out if the condition is true. There isn’t very much to add to this description except to emphasise the fact that you can follow THEN by any valid BASIC statement – including another IF!!!

This extended version of the IF statement seems very useful at first, but you quickly come to the conclusion that it’s not often that you want to execute a *single* statement as a result of some condition. Clearly what is needed is some way of writing a list of BASIC statements following the THEN. Dragon BASIC does provide a

way of doing this, although it is important to realise that not all versions of BASIC do. You can group together a number of BASIC statements on a single line, separating each one with a colon, and they will be obeyed in turn from left to right. For example:

```
10 PRINT "FIRST":A=1:PRINT "SECOND";A
```

will be carried out as a single line of BASIC working from left to right and is equivalent to:

```
10 PRINT "FIRST"
20 A=1
30 PRINT "SECOND";A
```

The use of colons to group a number of BASIC statements together effectively extends the rule for the default flow of control. Now instead of just obeying instructions in order of increasing line number (i.e. effectively moving down the screen) instructions that are grouped together on a line are obeyed from left to right. This is exactly in the same way as a human would read a list of instructions from a sheet of paper (i.e. from left to right and then downwards).

As mentioned earlier, the place where it is really useful to have the ability to put more than one BASIC instruction onto a line is following a THEN. For example, at the end of most games programs it is usual to ask if the player would like another game. In the following example of a routine that could be added to the end of a program, the answer is expected in the form of 1, to mean yes, or 0, to mean no. Any other value as a response will be taken to be a mistake and a message to that effect will then be printed and the question asked again.

```
110 INPUT "DO YOU WANT ANOTHER GAME (YES=1,
      NO=0)";A
120 IF A=1 THEN GOTO xxx
130 IF A<>0 THEN PRINT "YOU MUST ANSWER 1 OR
      0":GOTO 110
140 PRINT "BYE!!"
```

Notice the use of the colon to group two BASIC statements together in line 130. The xxx in line 120 should of course be replaced by the line number of the start of the program. Being able to execute a group of BASIC statements following a THEN is very useful, and it can make programs easier to write and easier to understand, but you will soon find that you cannot group very many statements together on one line before it becomes unreadable. There is another solution

to this problem, but for this we will have to wait until Chapter Six.

IF ... THEN ... ELSE

As mentioned earlier there is another special version of the IF statement that can be used to select which of two statements is carried out – the IF .. THEN .. ELSE .. statement. The general form of this is

IF 'condition' THEN 'list 1' ELSE 'list 2'

and its meaning is clear from the English meanings of the words THEN and ELSE. If the condition is true then 'list 1', a list of any valid BASIC statements separated by colons, will be carried out. If the condition is false then 'list 2' will be carried out instead. For example:

```
IF A>=0 THEN PRINT "A IS POSITIVE" ELSE PRINT "A  
IS NEGATIVE"
```

will print the first message if $A \geq 0$ and the second message otherwise. You can use this form of the IF statement to replace the IF .. THEN GOTO method of choosing which of two lists of instructions is to be obeyed. For instance the example involving reversing the sign of a number becomes

```
10 INPUT A  
20 IF A>0 THEN PRINT "A IS POSITIVE":B=A ELSE  
   PRINT "A IS NEGATIVE":B=-A  
30 PRINT A,B
```

But if the lists of instructions are more than a few statements long the IF .. THEN .. ELSE statement becomes quite unreadable! If this is the case then the earlier method of using IF .. THEN GOTO is to be preferred even though its flow of control is a little tricky to follow at times. There is a way around this problem, but once again we will have to wait for Chapter Six to find out what it is!

STOP

There is a very simple BASIC statement that we have not discussed so far that is very useful when used in conjunction with the IF statement. Suppose that as a result of some condition we would like

the program to stop, then currently the only way that we know of achieving this is to GOTO a line number that is at the end of the program. The BASIC statement STOP can be used anywhere in a program to return control to the user. For example:

```

10 PRINT "DO YOU WANT TO CONTINUE Y=1, N=0"
20 INPUT A
30 IF A=0 THEN STOP
40 IF A<>1 THEN GOTO 10
50 PRINT "I CONTINUE MASTER"
60 GOTO 10

```

Notice the use of STOP in line 30. Also notice how line 40 is used so that if a value other than 1 or 0 is input, the original question will be repeated. If the user replies 1 then the program continues – in this case all it continues to do is to ask the same question until the user types 0, when it stops.

A final example

As a further example of both the IF and FOR statements consider the problem of turning the stopping distance program given in the previous chapter into a sort of quiz. For a range of speeds the player is asked for the thinking, braking and total stopping distance in feet.

```

10 PRINT "STOPPING DISTANCE"
20 MARK=0
30 FOR S=10 TO 80 STEP 10
40 PRINT "WHAT IS THE THINKING DISTANCE AT
   ";S;" MPH";
50 INPUT T
60 IF T=S THEN MARK=MARK+1
70 PRINT "WHAT IS THE BRAKING DISTANCE AT ";S;"
   MPH";
80 INPUT B
90 IF B=S*S/20 THEN MARK=MARK+1
100 INPUT "WHAT IS THE TOTAL STOPPING DIS-
   TANCE ";D
110 IF D=S*S/20+S THEN MARK=MARK+1
120 NEXT S
130 PRINT
140 PRINT "YOU SCORED ";MARK
150 PRINT "OUT OF A POSSIBLE 24"

```

The flow of control summarised

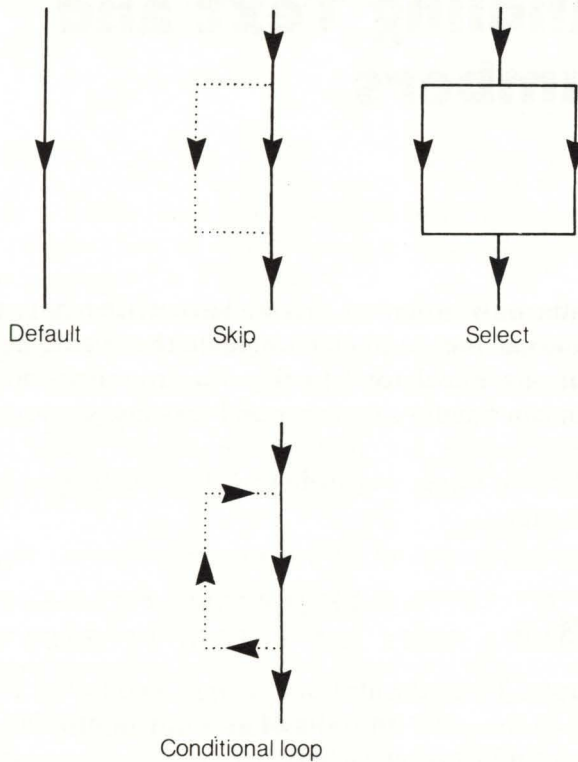


Fig. 4.7. Flow of control diagrams

It is difficult to say exactly what goes on in the mind of an experienced programmer. Whatever it is, it is a process that marks the difference between a beginner and an expert. One thing that does seem certain is that, in addition to the catalogue of programs that have been seen before and a collection of handy tricks, the standard forms of the flow of control diagrams that we have been studying in this chapter are ever-present. It is worth mentioning at this point that it has been proved that you can write any program using only the default, select and conditional loop. This is so important that it is worth gathering together the flow of control diagrams so that you too can store them away in your personal memory.

Chapter Five

Handling Text and Numbers

So far the only programs that we have written have used numbers. If this was all that computers could do they would be little different from pocket calculators! In this chapter we will look at how the Dragon can handle characters and text just as easily as digits and numbers. In the last part of the chapter some other ways of extending the things we can do with data are introduced – arrays and tape storage.

Strings

In Chapter Three the idea of a *string* – a collection of letters within double quotes – was introduced as a way of printing messages and prompts. In fact what we have been calling a string is really a *string constant*. The use of the word constant might alert you to the fact that there are such things as *string variables*. A string variable is similar to a simple variable in that it is a named area of memory that can be used to store information. In this case, the information is a collection of characters instead of a number. The rules for naming string variables are the same as for simple variables, except that a string variable **MUST** end in a \$ sign. The dollar sign is used to distinguish between a simple variable and a string variable (e.g. B is a simple variable but B\$ is a string variable and therefore different). The LET statement can be used to store a string constant in a string variable and the PRINT statement can be used to print its contents.

```
10 A$="THIS IS A STRING"  
20 PRINT A$
```

In fact, a string variable can be used anywhere that a simple variable can, as long as it makes sense. For example, you can use INPUT A\$ to store a string typed in from the keyboard while a program is

running, but `LET SUM=3+A$` is obviously nonsense (you cannot add a string to a number!). Notice in particular the difference between

```
1Ø A$="1"
```

and

```
2Ø A$=1
```

Line 1Ø is fine because the 1 is enclosed in double quotes and is therefore a string, but line 2Ø will give an error message because `A$` is a string variable and 1 is a number.

The introduction of string variables is exciting because it opens up the possibility of the Dragon handling text and even dialogues. So far, however, the only sort of program that we can write is

```
1Ø INPUT "WHAT IS YOUR NAME "; N$
2Ø PRINT "HELLO ";N$,"; I AM YOUR DRAGON
    COMPUTER"
```

which is all right for a start but can result in unnatural dialogues like

```
WHAT IS YOUR NAME? FRED BLOGGS
HELLO FRED BLOGGS, I AM YOUR DRAGON
COMPUTER
```

The trouble is, although we can `INPUT`, `PRINT` and store strings we have no way of changing them. This is rather like being able to `INPUT`, `PRINT` and store numbers, without being able to do arithmetic – it's obvious this would limit the programs we could write! We must therefore invent an 'arithmetic' for strings so that as well as having arithmetic expressions we can use 'string expressions'.

However, before moving on to look at how strings can be changed, there is one other complication that needs to be discussed. When it is first switched on the Dragon only allocates enough of its memory to store roughly 200 characters in the form of strings. If you are going to need more string storage than this you have to inform the Dragon of your intentions right at the start of the program, using the `CLEAR` command. `CLEAR n` will reserve enough memory to store approximately a total of `n` characters in the form of strings. For example, `CLEAR 10000` reserves sufficient memory for 10000 characters of string storage. It is important to know that as a side effect, `CLEAR` removes all variables that have been used in the program, so make sure that you only use `CLEAR` as the first statement in a program.

String expressions

Before introducing the Dragon's facilities for handling strings we must consider what sort of things you might like to do, and see if the Dragon can actually fit the bill. If a program had someone's first name in **F\$** and their last name in **L\$** then it would be useful to be able to 'join' them together to form one longer string. Such joining together of strings is known as *concatenation*. Another thing that would be useful is the ability to extract part of a string. For example, you could extract the last name from a string consisting of an initial and surname, i.e. extract "BLOGGS" from "F. BLOGGS". A string that is part of another string is often called a *substring*. For example, "BLOGGS" is a substring of "F. BLOGGS."

Another useful facility would be to replace one substring by another. For example, if we were trying to keep "F. BLOGGS" a secret we might want to replace the surname by asterisks, giving "F. *****". Finally, it would be a great advantage to be able to test for the presence of a particular substring in a string, to test, for example, to see if the substring "BLOGGS" occurred in the name stored in **N\$**. To recap, the string operations that we would like to find are – concatenation, substring extraction, substring replacement and substring searching.

Our first requirement, string concatenation, is immediately satisfied by the Dragon's concatenation sign **+**. If **A\$** contains the string "ABCD" and **B\$** contains "EFGH" then after

C\$=A\$+B\$

C\$ contains "ABCDEFGH". Notice that we now have two uses for the symbol **+**, as the sign for addition and as the sign for concatenation. You can use the **+** sign more than once in a string expression.

C\$="MR "+F\$+" "+L\$

will join up the four strings involved and if **F\$** contains a first name "FRED" and **L\$** a last name "BLOGGS" then **C\$** will contain "MR FRED BLOGGS". Notice the use of a single space between the two strings to avoid the result being "MRFREDBLOGGS"!

Extracting or changing a substring can be done by using a single new operation – **MID\$**. IF **s** is a string of characters

MID\$(s, m, n)

is the substring consisting of **n** letters starting with letter **m**. For example:


```
PRINT MID$ ("12345678", 3, 4)
```

will print "3456", i.e. the substring consisting of four characters starting at the third character. The MID\$ operation is easy to understand and use, as long as you remember that the first number indicates where the substring should start and the second number indicates how many characters should be in the substring.

You can use arithmetic expressions to specify m and n, for example:

```
MID$("ABCD", FIRST, 2)
```

or

```
MID$("ABCD", 1, L+3)
```

However, by now you may have realised that one of the most powerful principles in BASIC is that anywhere you can use a constant or a variable you can also use an expression. You shouldn't be frightened to write complicated string expressions, any more than you would be over complicated arithmetic expressions.

You can't really make a mistake using MID\$. If you specify that the substring should be longer than the number of letters to the left of the starting position then you will just get as many letters as there are! For example, MID\$("12345", 4, 10) is just "45". If you leave n out then the substring will consist of all the letters to the end of the string s. For example, MID\$("12345", 4) is also "45". The only error that you can make is to specify a starting position less than 1. For example, MID\$("12345", 0, 10) will give you an error message. However, if you use a correct starting position and specify a length of zero you will not get an error, but a substring with no characters in it! This is a very special string – the *null string*. The null string has no characters in it and plays a very similar role in string expressions to that of zero in arithmetic expressions. The string constant that corresponds to the null string is written "", i.e. a pair of double quotes with nothing in between. Notice the difference between "" and " ". The first is the null string and has no characters in it, the second is a string consisting of one character – a blank or space. From the point of view of a computer, a space is just as much a character as a letter of the alphabet, it takes up one position to print and it needs just as much computer memory to store! If you print a null string it has no effect whatsoever. The statements PRINT A;B and PRINT A;"",B produce the same result.

As an example of using MID\$, consider the problem of printing

the name of a month of the year, given its number, i.e. print DEC for 12 and MAY for 5, etc. Try the following short program.

```
10 Y$="JANFEBMARAPRMAYJUNJULAUGSEPOCT
   NOVDEC"
20 INPUT "MONTH NUMBER"; MON
30 PRINT "MONTH";MON;"IS";MID$(Y$,1+3*(MON-1),3)
40 GO TO 20
```

If you enter a number in the range 1 to 12 the program will print the correct abbreviation for the month in question. The way that it works is by extracting the three letter name from the long string Y\$. The best way to understand the use of MID\$ in line 30 is by working it out by hand for a few values of MON. If MON is 4 then $(1+3*(MON-1))$ is 10 and the substring is given by MID\$(Y\$, 10, 3) which gives the three letters "APR".

The MID\$ operation will extract any substring from a string, and in this sense it is all you need. However, there are two other ways of extracting substrings that have become common, and indeed popular, in BASIC – LEFT\$ and RIGHT\$. LEFT\$(s,n) will extract the string with n characters in it, starting from the first character in s; in other words the leftmost n characters. You should be able to work out that LEFT\$(s, n) is the same as MID\$(s, 1, n). RIGHT\$(s, n) will extract the last or rightmost n characters from the string s. You may think that this has no equivalent MID\$ operation, but once you know that the Dragon will 'tell' you the length of a string on request the equivalent is obvious. If you write LEN(s) the Dragon will count the number of characters in a string. For example, PRINT LEN("ABCD") will print 4 on the screen. Now you should be able to work out that RIGHT\$(s, n) is the same as MID\$(s, LEN(s)-n+1, n).

There is no way of directly changing characters that are part of a string. In other words, if S\$ contains "AB3D" there is no way of directly altering the "3" to "C". However, using MID\$ you can take the string apart and insert the new letters. For example, to replace the "3" in S\$ with "C" you could use

```
S$=MID$(S$, 1, 2) + "C" + MID$(S$, 4, 1)
```

The first MID\$ extracts "AB" and the second the final "D". Putting them all together gives "ABCD" as required. This is the only way that you can alter part of a string.

The only thing left from our initial list of string handling requirements is testing to see if a particular substring is present in another string. Dragon BASIC provides a direct method of

achieving this – the INSTR operation. It extends the use of conditional expressions (see Chapter Four) to strings and this can often be faster and simpler. INSTR (n, s, t) will search for the string s in the string t, starting the search at character n. So for example, INSTR (3, “FRED”, T\$) will search T\$ starting at the third character for the string “FRED”. If the string is not found INSTR produces the result zero, otherwise it produces the position of s in t.

The INSTR operation is very useful but it is often more than you really need. For most of the time all you need to know is if two strings are equal or different. In fact you can use all of the relations that were introduced in Chapter Four with strings. The meanings of = and <> are easy enough to understand. Namely, two strings are equal if they are of the same length and contain the same characters in the same order, otherwise they are not equal. However, what do the relations <, >, <= and >= mean when applied to strings? The answer to this question varies from BASIC to BASIC. In the case of the Dragon, however, they are defined so that A\$<B\$ is true if the string in A\$ would come before the string in B\$ in an alphabetically ordered list of strings. The trouble is that we are all so familiar with alphabetically ordered lists that we tend to forget how they work! If the two strings being compared are single letters, then A\$<B\$ if the letter in A\$ comes earlier in the alphabet than the letter in B\$. For example, “A”<“B” is true but “D”<“B” is false.

What about comparing strings that contain single characters that are not necessarily letters, i.e. what do we make of *<\$? In the case of the letters, the alphabet provided us with a ready-made order, so what we need is to extend this order to include all the other symbols that the Dragon can use. In other words we need a ‘super alphabet’! This is already available for the Dragon. If you look at Appendix A in the *Dragon Manual* you will see a listing of the complete set of Dragon characters in a predefined order, and it is this that is used as the super alphabet to decide if *<\$. If you don’t have the *Dragon Manual* to hand, or if you are just interested, you can print all the characters in their proper order by using the following program:

```
10 FOR I=33 TO 127
20 PRINT "CHARACTER ";I;" = ";CHR$(I)
30 NEXT I
```

Line 10 specifies 33 as the starting point, as the characters before that are unprintable. Don’t worry about the use of CHR\$ in line 20 – this will be explained later. Coming back to the question of whether or not *<\$ is true or false, * is character 42 and \$ is character 36, so \$

comes before * in the order, and *<\$ is false. You can decide the truth or otherwise of any relation in the same way.

If the two strings contain more than one character they are compared one character at a time until the first pair of different characters is found. The relationship between the two strings is then decided on the basis of those two characters. For example, `ABDC<AZCD` is true because the first pair of letters that are different is B and Z and `B<Z` is true. If one of the strings is the same as the other apart from the addition of a few extra characters then the comparison is based on length, i.e. `ABCD<ABCDEF`, because there is no pair of letters that is different and `ABCD` is shorter than `ABCDEF`.

Now we can do everything that we wanted to with strings! However, there are other string operations and these will be introduced in the next chapter.

Arrays

Strings and numbers are the only two types of data that the Dragon can handle, and this is quite sufficient for most purposes. However, the Dragon does provide a way of using the basic types of data in a more sophisticated way, the *array*.

Consider the problem of reading in five numbers and printing them out on the screen in reverse order. So far the only method that we could use is

```
10 INPUT A1, A2, A3, A4, A5
20 PRINT A5, A4, A3, A2, A1
```

which is not too bad for five numbers, but think what the program would look like if the problem was to reverse 100 numbers!

What we need to be able to do is to refer to a variable like `A(I)` where `I` can take values from 1 to 5 in a `FOR` loop. Then we could write

```
20 FOR I=1 TO 5
30 INPUT A(I)
40 NEXT I
50 FOR I=5 TO 1 STEP -1
60 PRINT A(I)
70 NEXT I
```

This is in fact exactly what BASIC allows you to do. The collection

of variables $A(1)$ to $A(5)$ is called *the array A* and a particular variable $A(I)$ is called an *element* of the array (see Fig. 5.1). The only complication is that before you can use an array you must tell your

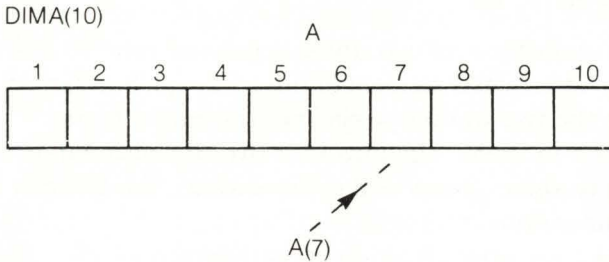


Fig. 5.1. A one-dimensional array

Dragon how many elements the array is going to have. This is done using the DIMension statement:

```
10 DIM A(5)
```

which should be added to the previous program to make it work! If you define a variable as having only five elements and then try to use $A(6)$ you will get an error message for trying to use something that doesn't exist! It is tempting to think that it is better to define arrays larger than you need to try to avoid such error messages, but be warned, arrays can quickly use up all the memory that your machine has to offer!

There are two slight complications that haven't been mentioned so far. The first is that if you use `DIM A (10)` then the Dragon will actually create an array with 11 elements. Where, you might ask, is the extra element? The answer is that every array has a zeroth element. For example, `DIM A(10)` creates $A(0)$ to $A(10)$, i.e. eleven elements! The reason why this extra element has been ignored until now is that many versions of BASIC do not provide zeroth elements in arrays. If you fall into the habit of using elements like $A(0)$ then your programs won't work on BASICs that do not provide $A(0)$, but if you avoid using any zeroth element the only consequence is that you waste a tiny amount of memory needed to create $A(0)$. For this reason the zeroth element of every array will be ignored! The second problem is that you can only dimension an array once in a program! If you first use `DIM A(10)` and then later use `DIM A(20)` or even `DIM A(10)` again you will get an error message – so keep DIM statements out of loops!

In addition to being able to define arrays that can be thought of as

rows of variables (see Fig. 5.1), you can define arrays that correspond to organising variables into tables made up of rows and columns. For example

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

defines a collection of variables organised into 10 rows and 10 columns. A particular element of this array can be referred to as A(I, J), where the two indices select the row and column.

Although in theory the idea of a two-dimensional array can be extended to three-, four- and n-dimensions, the Dragon is limited to two dimensions.

You can form arrays from string variables as well as numbers, and this can be used to store and manipulate lists of words. For example

```
DIM S$(10)
```

is a string array composed of 10 different strings. Each element of a string array behaves exactly like a standard string and can be used to store a variable number of characters. For example, the number reversing program can be used to reverse a list of words:

```
10 DIM A$(5)
20 FOR I=1 TO 5
30 INPUT A$(I)
40 NEXT I
50 FOR I=5 TO 1 STEP -1
60 PRINT A$(I)
70 NEXT I
```

You can use MID\$, LEFT\$, etc., and any other string operations on any element of a string array.

A word game

As an example of using string arrays consider the problem of writing a program to play the game of hangman. Because the computer cannot 'think up' a list of words for you to guess, it is necessary to ask someone else to type in a list for you to guess. Once the list of words has been entered the player must try to guess each word in turn letter by letter. Each letter that is entered must be checked against each letter in the word. If it is present then the letter in the word must be replaced by a blank to make sure that the player cannot guess it a second time. When all the letters have been guessed

the program moves on to the next word, or if there is none, comes to an end. After this description you should be able to make a good attempt at your own hangman program, before looking at the one below:

```

10 DIM W$(5)
20 FOR I=1 TO 5
30 INPUT "WORD = ";W$(I)
40 NEXT I
50 FOR I=1 TO 5
60 G=0
70 T$=W$(I)
80 G=G+1
90 INPUT "GUESS=";A$
100 F=0
110 FOR J=1 TO LEN (W$(I))
120 IF LEFT$ (A$, 1)=MID$ (W$(I), J, 1) THEN PRINT
    "YES ! -";LEFT$ (A$,1):W$(I)=MID$(W$(I),1,J-1)+
    " "+MID$(W$(I),J+1):A$=" "
130 IF MID$(W$(I),J,1)<>" " THEN F=1
140 NEXT J
150 IF F=1 THEN GOTO 80
160 PRINT "YOU GOT IT IN ";G;" !!"
170 PRINT "THE WORD WAS ";T$
180 NEXT I
190 PRINT "GAME OVER"

```

Line 10 defines the array W\$ so that it can hold five words. Lines 20 to 40 are used to input the words. The FOR loop starting at line 50 and ending at line 180 repeats the guessing part of the program five times, once for each word. The variable T\$ is used in line 70 to store the word until the end of the game so that the array element can be modified by storing blanks in the place of letters that have been guessed. The guess is input in line 90. Each letter in the word is checked against the current guess by the FOR loop starting at line 110 to line 140. The variable F is used to check that there are still letters left to be guessed.

Initialising variables – DATA and RESTORE

It often happens that a standard set of values needs to be stored in a set of variables or an array for a program to work properly. For

example, suppose we want to print the number of days in a given month. We could set up an array of 12 elements and 'store' the answer in each element – i.e. the number of days in Jan. would be stored in the first element of the array, the number in Feb. in the second, and so on. This is a very simple and useful idea, but how do you initialise each element of the array to the correct value? You could use a FOR loop and an INPUT statement to read in the answers or you could write 12 LET statements.

To make life slightly easier, BASIC provides the facility to store data within a program so that it can be transferred to any variables of your choice. The data is stored in a DATA statement that is composed simply of the word DATA followed by a list of values separated by commas. For example, the days in each month could be written as

```
10 DATA 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
```

To transfer these data values into variables the READ statement is used. A READ statement is simply the word READ followed by a list of variables separated by commas. Each time a READ statement is encountered, data values are transferred into variables in the variable list – one data value per variable. The best way to think of this is to imagine a pointer initially set to the first data value in the DATA statement. Each time a READ statement transfers a data value into a variable, the pointer is moved on to the next data value. Whenever a READ statement is obeyed data is transferred, starting from whatever data value it has reached after previous READ statements. So the array holding the 12 data values in the example DATA statement given above can be produced using

```
20 DIM M(12)
30 FOR I=1 TO 12
40 READ M(I)
50 NEXT I
```

You can have as many DATA statements in a program as you like, anywhere that you like, and they are treated as if all the data that they store was contained in one big DATA statement. So when the pointer moves past the end of a DATA statement it moves to the beginning of the next DATA statement. If there isn't one, however, you'll get an error message.

There is no restriction on the type of data you can store in a DATA statement – i.e. you can use strings or numbers – but you must always be careful to READ data into variables of the same type

– i.e. strings into string variables and numbers into numeric variables. Strings in DATA statements do not have to have double quotes around them unless they include commas, otherwise it wouldn't be possible to tell where one data item ended and another began. As an example of using strings in DATA statements consider the following

```
1Ø DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,
    OCT,NOV,DEC
2Ø DIM M$(12)
3Ø FOR I=1 TO 12
4Ø READ M$(I)
5Ø NEXT I
```

and compare it to the example given earlier.

Sometimes, especially in games, it would make things easier if we could re-read the data. This can be done using the RESTORE command. Following RESTORE that pointer is back at the beginning of the first DATA statement in the program.

For example, try

```
1Ø DATA 1,2,3,4
2Ø READ A,B,C,D
3Ø PRINT A, B, C, D
4Ø RESTORE
5Ø GOTO 2Ø
```

Saving data on tape

We have already seen that it is possible to SAVE and LOAD programs on tape. It would seem to be logical to extend the facility to include SAVEing and LOADing data stored in variables to and from tape. This is indeed possible on the Dragon. Before you can save any data to tape you have to open a tape file. This may sound complicated, but it is really only a way of giving a name to the data that you are going to store in the same way that you give a name to programs that you store. The format of an OPEN statement is:

```
OPEN "O", #-1, filename
```

The "O" stands for Output and the #-1 tells the Dragon that you are going to use the tape recorder. So far the Dragon knows that at some point in the program we are going to save data on to tape. The data will be saved under the name 'filename'. To send data to the tape you

have to use a slightly modified version of the PRINT statement:

```
PRINT #-1, 'output list'
```

Once again the #-1 lets the Dragon know that you are sending data to the tape recorder. The 'output list' is exactly the same as the print list in a normal print statement, and indeed the best way to think of the PRINT #-1 statement is that it is a normal PRINT statement but the output it produces is sent to the tape rather than the screen. Finally, when you have finished recording on data you have to inform the Dragon that you no longer want to use the tape recorder by using

```
CLOSE #-1
```

The only difficult point in all of this is that you have to remember to set the tape recorder to record before the Dragon wants to use it.

Reading data back works in roughly the same way. First you have to open the file using

```
OPEN "I", #-1, filename
```

where the only difference is that the "I" stands for Input. To read data back from the recorder you can use a slightly modified INPUT statement:

```
INPUT #-1, 'input list'
```

and when you have finished with the file you have to close it using

```
CLOSE #-1
```

The only real difference between reading and writing a file of data is that when you are reading you must be careful not to read beyond the end of the file. To help you avoid this the Dragon provides the EOF or End Of File indicator. If you have reached the end of the file EOF (-1) will be true. So before you read from a file use

```
IF EOF(-1) THEN 'do not read'
```

Using the tape recorder to handle data files is not difficult but such programs are usually very long. As a small example to demonstrate the basic operation of data files, try the following program which writes the numbers 1 to 1000 onto tape:

```
10 PRINT "PLACE A FRESH TAPE IN THE RECORDER"
20 PRINT "AND PRESS PLAY AND RECORD"
30 INPUT "ARE YOU READY ";AS
```



```
40 IF LEFT$(A$,1)<>"Y" THEN GOTO 30
50 OPEN "O",-1,"TEST"
60 FOR I=1 TO 100
70 PRINT I
80 PRINT -1,I
90 NEXT I
100 CLOSE -1
```

Then use the following program to read them back.

```
10 PRINT "REWIND TAPE AND PRESS PLAY"
20 INPUT "ARE YOU READY ";A$
30 IF LEFT$(A$,1)<>"Y" THEN GOTO 20
40 OPEN "I",-1,"TEST"
50 IF EOF (-1) THEN CLOSE -1:STOP
60 INPUT -1,I
70 PRINT I
80 GOTO 50
```

Apart from remembering to press the correct controls on the tape recorder it's as easy as that!

Chapter Six

Functions and Subroutines

At this point in learning BASIC you should be in a position to see that the expression is the main way that programs *change* data. Without expressions – arithmetic, conditional and string – BASIC would be reduced to moving values from one place to another. It is only by the use of expressions that values can be combined and compared to produce new results. To make expressions even more useful, BASIC provides a large range of operations that can be used to make expressions, in the form of *functions*. You can also extend the range of functions by creating your own, *user-defined* functions. This creation of new operations can be taken one stage further in BASIC by using the GOSUB and RETURN statements to group statements together into *functional units* or *subroutines*.

In the first part of this chapter we look at the general idea of a function and then move on to examine some of the more common functions available to the Dragon programmer. The sections that deal with particular functions can be read very quickly, or even skipped, until you need to use them or until they are used in an example. However, don't skip the section on *special functions* because these are particularly important.

The idea of a function

Before dealing with the way the Dragon handles functions, it is worth looking at functions in general. You may already be familiar with the idea of a function from mathematics. For example, $\sin(x)$ is a function. However, the idea of a function isn't really anything to do with advanced mathematics. At its most simple, a function is an operation on data that produces a *single* value as its result. For example, finding the larger of two numbers is an operation on data that returns a single value – the maximum of 3 and 42 is 42, the

maximum of 2 and 2 is 2 – and *maximum* is therefore a function.

The Dragon doesn't have a maximum function but it is nevertheless a useful one to consider as an example because it is easy to understand.

The standard way of writing a function involves writing its name to the left of the values that it is to operate on enclosed in brackets. In the case of finding the maximum of two numbers a sensible name for this function is 'max' and so the previous two examples can be written

max (3,42)

and

max(2,2)

Following the usual BASIC convention that anywhere that you can use a constant or a variable you can use an expression, the following is also allowed.

max (count+3, total*20)

The data values that follow the name of the function are called *parameters*. It is possible for functions to have any number of parameters, but the functions supplied on the Dragon use one, two or at most three parameters.

Functions can be used in expressions just as if they were variables or constants. For example

RESULT=max(3.3,4.2)

would evaluate our function max and store the answer (4.2) in RESULT. (Remember that the Dragon doesn't have a max function so don't try this example.) You can now see why the condition that a function should give only one result is so important. If a function gave more than one result, which one would be stored in the variable RESULT or which one would be used to evaluate the rest of the expression? As we want to use functions in expressions they *can* only return one answer. Sometimes it is possible to change something that isn't a function into a function by simply choosing one of the possible answers. For example, the Dragon has a function SQR which gives the square root of a number. Now if you ask for the square root of four the answer is obviously two, i.e. two times two is four, but it is all too easy to forget that minus two is also the square root of four. A minus times a minus is a positive and so -2×-2 is 4 (not -4). The objection that the square root operation isn't a

function can be avoided by simply deciding that SQR will be a function that returns the positive square root of a number.

The Dragon's functions

The Dragon has a very wide range of functions, some of them so specialised that it is better to deal with them in detail in other chapters. However, there is a 'central core' of functions that you would expect to find in any BASIC, and these will be explained in this chapter. A full list, with brief explanations, of all the Dragon's functions can be found in its manual.

The core functions can be divided into three groups: the arithmetic functions such as SQR and ABS; the trig functions such as SIN and COS; and the string functions such as LEN and CHR\$. In addition there are a number of unclassifiable but very important 'one-off' functions such as RND. The most important thing is to have some idea of what functions are available, so a brief reading of the description of each function listed below is recommended. However, it is difficult to appreciate, let alone remember, the subtler details of the use of a function until you actually *need* to use it! If you want to see the effect of any of the functions use the following program:

```
1Ø INPUT X
2Ø PRINT SIN(X)
3Ø GOTO 1Ø
```

but change the PRINT SIN(X) to the function that you are interested in.

Arithmetic functions

ABS – absolute value of a number

The absolute value of a number is obtained by ignoring its sign and treating it as positive, i.e. ABS(−2) is 2 and ABS(2) is 2.

EXP – exponential function

EXP(x) raises e, the exponential number, which has the value 2.71821, to the power of x. That is EXP(x) is the same as e^x .

It is difficult to explain why this function is so important, but it crops up in just about every area of mathematics, (see also the function LOG). When using EXP it is well worth being aware of the

fact that $\text{EXP}(x)$ gets very big for even small values of x . The largest integer that $\text{EXP}(x)$ can accommodate on the Dragon is 88. Larger numbers will cause an OV or overflow error.

FIX – the integer value of a number

The action of the FIX function is very easy to understand – it simply changes any number to a whole number by ‘chopping off’ any fractional part. For example $\text{FIX}(1.1)$ is 1 and $\text{FIX}(-1.1)$ is 1.

INT – round a number down

The INT function is closely related to the FIX function in that it will remove the fractional part of any number. The difference is that this is achieved by rounding the number *down* to the nearest whole number. For positive numbers this corresponds to ‘chopping off’ the fractional part, e.g. $\text{INT}(3.21)$ is 3, and so in this case INT gives the same result as FIX, but for negative numbers things are a little more complicated. Rounding a negative number down looks a little strange, e.g. $\text{INT}(-4.7)$ is -5 , but this is simply because $-4 > -5$, i.e. -5 is *smaller* than -4 !

LOG – natural logarithm of a number

The natural logarithm of a number is the power that you have to raise e to give the number. Most people are more familiar with a slightly different form of the logarithm. The logarithm that is given in most log tables is in fact the logarithm to the base 10. In other words it is the number that 10 has to be raised to give the original number. Some versions of BASIC include both versions of the log function. The Dragon lacks such a function, but this is no great disadvantage because you can obtain the log to the base ten by using $\text{LOG}(x)/\text{LOG}(10)$. As $\text{LOG}(x)$ is the number that you have to raise e to, to get x you should be able to see that $\text{EXP}(\text{LOG}(x))$ is x .

SGN – the sign of a number

The sign of a number is $+1$ if the number is positive and -1 if it is negative. For example, $\text{SGN}(-232)$ is -1 and $\text{SGN}(3239)$ is 1.

SQR – the square root of a number

The square root of a number is a result that when multiplied by itself gives the original number, i.e.

$$\text{SQR}(x) * \text{SQR}(x) \text{ equals } x$$

Notice that negative numbers do not have square roots because if you multiply any number, even a negative one, by itself you will get a

positive number. If you try to take the square root of a negative number you will therefore get an FC or function call error.

The trigonometrical functions

The best known examples of functions are probably the *trigonometrical* or *trig* functions. It is beyond the scope of this book to go into detail about the theory of trigonometry, and anyway the need to use such functions always arises from a very specific problem. There is, however, one use for the trig functions that is important to nearly every computer user interested in graphics, namely drawing circles. Dragon users are particularly lucky, however, in that Dragon BASIC includes a command that will plot a circle. This is covered in Chapter Nine and its presence avoids the need to go into any detail about using SIN, COS and TAN to draw circles. If you need more specific information about the trig functions then see Appendix D of the Dragon's manual.

If you do need to use any of the trig functions, it is important to realise that the Dragon doesn't measure angles in degrees but in radians. If you want to convert an angle in degrees to radians then use

$$\text{radians} = \text{degrees} * \text{PI} / 180$$

Where PI is approximately 3.14159, and to convert radians to degrees use

$$\text{degrees} = \text{radians} * 180 / \text{PI}$$

The three trig functions available on the Dragon are:

- SIN – sine of an angle measured in radians
- COS – cosine of an angle measured in radians
- TAN – tangent of an angle measured in radians

The Dragon also has available the inverse function related to the TAN function.

ATN – arctangent

The arctangent of a number is the angle in radians whose TAN is equal to the number, i.e. $X = \text{TAN}(\text{ATN}(X))$. If you need the inverse of SIN or COS then you can use ATN as described in the *Dragon Manual* (Appendix D), but it is better to use a *user-defined function*, a topic that is covered later in this chapter.

String functions

We have already met most of the string functions in Chapter Five. For completeness and to provide a handy reference, all of the string functions are described here. However, only the new functions are described in any detail. To see how the functions that require string input work, modify the following program:

```
1Ø INPUT X$
2Ø PRINT ASC(X$)
3Ø GOTO 1Ø
```

ASC – the ASCII code of a character

The ASC function does the opposite to the CHR\$ function, in that it returns the position in the list of characters of any particular character. For example, ASC("A") is 65 and CHR\$(65) is "A". If ASC is applied to a string of more than one letter the code of the first character in the string is returned as the result. If the string is null, i.e. contains no characters, the code returned is zero.

CHR\$ – character function

CHR\$(N) will give the character that is at the Nth position in the list of all the Dragon's characters. CHR\$ will return all the characters that the Dragon can use even if they cannot be printed on the screen. So, if you type CHR\$(8), or any number up to and including 33, all you will see is a blank screen.

HEX\$ – hexadecimal representation

HEX\$ is a function that you certainly won't use unless you are going to become involved in the technicalities of how the Dragon works! Its action is very simple to describe: it converts any number into a string of characters that represents the number in base sixteen. In other words it returns the hexadecimal representation of the number. For example, HEX\$(15) is "F" and HEX\$(255) is "FF". Before you can use HEX\$, however, you need to understand something about hexadecimal numbers.

INSTR – find a substring within a string

INSTR (n, 'string1', 'string2') will search for the occurrence of 'string1' within 'string2'. The search will start from letter N in 'string2'. If 'string1' is present in 'string2', INSTR returns its starting position, otherwise it returns zero.

LEFT\$ – the left part of a string

LEFT\$(‘string’,N) returns the first N characters of a string.

LEN – the length of a string

The LEN function returns the length of any string. For example, LEN (“COMPUTER”) is eight and LEN (“”) (the null string) is 0.

MID\$ – the mid part of a string

MID\$(‘string’,M,N) returns a string of length N starting at character M. If N is omitted then the entire string to the right of character M is returned.

RIGHT\$ – the right part of a string

RIGHT\$(‘string’,N) returns the last N characters of a string.

STR\$ – converts numbers to strings

The STR\$ is a function that is useful for advanced applications. It converts any number (or the result of an expression) to the string of characters that would be displayed if the number (or the result of the expression) were printed. The STR\$ function therefore provides a link between numbers and strings – for example, “JULY”+STR\$(31) works out to the string “JULY 31”.

STRING\$ – makes a string by repeating a character

The function STRING\$ can be used to make a string consisting of a single character repeated a specified number of times. STRING\$(‘character’,N) produces a string of N ‘characters’. For example, STRING\$(“A”,3) produces “AAA”. If you want you can specify the character to be repeated by using its ASCII code rather than quoting it. For example, STRING\$(65,3) also produces “AAA” because ASC(“A”) is 65. The main use of STRING\$ is in PRINT statements to print multiple spaces or to print lines of special characters to separate areas of the screen. For example PRINT STRING (“*”,32) will print a line of 32 asterisks.

VAL – converts digit strings to numbers

The VAL function is the opposite of the STR\$ function in that it converts a string into a number. The string can be any sensible collection of digits, i.e. anything that you could type in response to an INPUT statement. For example, VAL (“34”) is 34 and if A\$=“123.456” then VAL (A\$) is 123.456

Special functions

There are three functions, RND, INKEY\$ and TIMER that are so generally useful that it seems worthwhile to treat them on their own and at some length. RND is a function that returns a number, so it could otherwise have been treated in the section on arithmetic functions. INKEY\$ returns a character, so it could otherwise have been treated as a string function. TIMER returns a number, and so could also have been treated in the section on the arithmetic functions, but you can also assign a value to it as if it was a variable and this makes it a very strange function.

RND

RND ($\emptyset$) is a function that returns a number in the range $\emptyset$ to less than 1 which can be treated as if it were random. To say that a computer can give a number at random always sounds like a contradiction, and indeed to some extent it is. The point of confusion comes from the use of the word 'random'. If you are using the computer to play a game then all that you need is a sequence of numbers that are not predictable by anyone playing the game. In other words, for most purposes a list of numbers can be said to be random if there is no detectable pattern. If you run the following program

```
1 $\emptyset$  PRINT RND ( $\emptyset$ )
2 $\emptyset$  GO TO 1 $\emptyset$ 
```

you should see a list of numbers that shows no obvious pattern. (In fact there is a pattern but it is so complicated it takes a Dragon to follow it!) This sort of randomness is more correctly called *pseudo randomness* and the RND function is a *pseudo random number generator*. The numbers that it produces are evenly spread throughout the range $\emptyset$ to less than 1, i.e. any number is equally likely to come up as any other, and there should be no discernible pattern that would help someone predict the next number that RND will produce.

The main trouble with RND is that it's not often that we need a random number in the range $\emptyset$ to less than 1. We normally need the program to do one of a number of different things at random. The best way of doing this is to change the RND into a random whole number between 1 and n, where n is the number of any things we want to select from. The Dragon provides another form of the RND function for just this purpose:

```
RND(N)
```


produces random whole numbers between, and including 1 and N. For example, if you want to program a six-sided dice then you would choose six as the value of N, and

```
1Ø PRINT RND(6)
2Ø GOTO 1Ø
```

will print a number 1 to 6 with approximately the same frequency and in such a way that there should be no obvious pattern. The subject of how to use random numbers in programs is too vast to cover in this book, but many examples will crop up in other chapters. If you want to explore randomness further, Chapter Five of *The Complete Programmer* by Mike James, published by Granada, is devoted to this topic.

INKEY\$

The function INKEY\$ is closely related to INPUT, in that it can be used to *read in* a single character from the keyboard. The difference is that INPUT A\$ waits for something to be typed on the keyboard and the ENTER key to be pressed, but INKEY\$ doesn't wait. If you try the following program

```
1Ø INPUT A$
2Ø IF A$<>"" THEN PRINT A$
3Ø GOTO 1Ø
```

you will have to press ENTER before you see anything on the screen. However, if you change line 1Ø to

```
1Ø A$=INKEY$
```

the character corresponding to any key that you press appears on the screen at once. Notice that another difference between INPUT and INKEY\$ is that INKEY\$ doesn't automatically print anything that you type on the screen.

Whenever the Dragon meets the INKEY\$ function it immediately examines the keyboard. If there is a key already pressed the appropriate character is returned by the function. If no key is pressed the function returns the null string. No matter what has happened the INKEY\$ function does *not* wait for a key to be pressed.

The main use of INKEY\$ is in games, where the 'arrow' keys are used to control the movement of something on the screen. The following program indicates how the Dragon detects that one of the arrow keys has been pressed:

```

1Ø A$=INKEY$
2Ø IF A$="" THEN GOTO 1Ø
3Ø IF A$=CHR$(8) THEN PRINT "LEFT"
4Ø IF A$=CHR$(1Ø) THEN PRINT "DOWN"
5Ø IF A$=CHR$(94) THEN PRINT "UP"
6Ø IF A$=CHR$(9) THEN PRINT "RIGHT"
7Ø GOTO 1Ø

```

Line 1Ø gets the character corresponding to any key pressed on the keyboard, if any. Line 2Ø tests to see if A\$ is the null string, i.e. no key has been pressed, and if it is, sends control back to line 1Ø. Thus the loop formed by lines 1Ø and 2Ø only stops when a key is pressed. Then lines 2Ø to 5Ø test to find out which of the arrow keys are pressed and print an appropriate message. As three of the arrow keys produce unprintable characters there is no choice but to use CHR\$ functions with the correct character codes within the IF statements. Line 7Ø repeats the whole program. Notice that if any key other than an arrow key is pressed the loop formed by lines 1Ø and 2Ø stops but nothing is printed on the screen. In Chapter Nine an example is given where the same sort of program is used to drive a dot around the screen.

TIMER

The TIMER function can be used to give your programs a sense of time. The value returned by TIMER increases by one every fiftieth of a second. You can think of it as a clock that ticks every fiftieth of a second. For example:

```

1Ø PRINT TIMER/5Ø
2Ø GOTO 1Ø

```

will print a number on the screen that increases by one every second.

There is one odd, but very useful way that TIMER can be used that makes it different from all other functions. You can assign a value to TIMER just as if it was a variable, and this value will be incremented every fiftieth of a second. For example try

```

1Ø TIMER=Ø
2Ø PRINT TIMER/5Ø
3Ø GOTO 2Ø

```

which zeros the timer and then prints a count of the number of seconds from the start of the program. If you leave this example

program running you will eventually see the number printed by line 20 go back to zero and the count start over again. There is an upper limit of 65535 on the value of TIMER and after this value it is reset to zero. This means that the largest time interval that TIMER can cope with is 1310.7 seconds or roughly 21 minutes.

The most obvious use of TIMER is to time how long something takes to happen, but it is equally useful in producing delays in programs of known lengths. For example:

```
10 TIMER=0
20 IF TIMER<50 THEN GOTO 20
30 PRINT "ONE SECOND"
```

will print its message on the screen one second after you type RUN.

User-defined functions – DEF FN and FN

In our general introduction to functions, the idea of a function to find the maximum value of two numbers was discussed, but it was pointed out that, although the Dragon has a wide range of functions, such a max function isn't among them. Dragon BASIC does allow definition of new functions, but not with enough freedom to allow us to create such a function to determine the maximum of two numbers. However even this limited ability is worth having.

You can define a new function in terms of an expression. For example, the Dragon lacks a *square* function, i.e. one that will work out the square of any number. The names of all user-defined functions are FN followed by one letter, so you can define the new function FNS to fill this gap by

```
DEF FNS(X)=X*X
```

The word DEF is used to indicate that this is a function definition. The meaning of this function should be clear – whenever the Dragon sees the function called FNS it squares the parameter within the brackets next to it. For example:

```
A=FNS(2)
```

would result in 4 being stored in the variable A. This is all fairly straightforward, but what about the following:

```
10 DEF FNS(X)=X*X
20 X=3
```



```

30 Z=4
40 A=FNS(Z)
50 PRINT "X=";X;" Z=";Z;" A =";A

```

Before you run this program try to work out what the values of X, Z and A will be. The correct answer is that X is 3, Z is 4 and A is 16. Any possible confusion comes from the fact that X is used in the definition of the FNS function and in the program. A point to note is that the names that you give to parameters in the definition of functions are nothing to do with any variables that you might use in the rest of the program. In this sense DEF FNS(A)=A*A, DEF FNS(Z)=Z*Z, etc., all define the same function! In a function definition the parameter merely shows what is to happen to the *real* parameter when the function is used – because of this such a parameter is often called a *dummy parameter*.

If you've managed to cope with these ideas, you might like to try to work out what this program's result is.

```

10 DEF FNT(I)=SUM+I
20 SUM=0
30 PRINT FNT (30)
40 SUM=10
50 PRINT FNT (30)

```

In this case the variable SUM, used in the expression to define the function, isn't a dummy parameter and it is therefore taken to be a variable in the main program, i.e. SUM in the function definition is the same as SUM in the rest of the program. The idea of a dummy variable gets easier after you have had time to think about it.

To summarise, the rules for forming a user-defined function are:

- (1) Every function that you use must be defined using a DEF FN statement somewhere in the program before it is first used.
- (2) Only numeric user-defined functions are allowed.
- (3) A function definition must include exactly one parameter.
- (4) Any valid Basic expression can be used in a function definition.

These four rules may seem like a lot to remember, but they all accord with common sense. Some examples of functions may help to clarify matters:

```

DEF FNL(X)=LOG(X)/LOG(10)
DEF FNC(X)=(ATN(X)/SQR(1-X*X))+1.5708

```

The first function will calculate log to the base ten (see the notes on the LOG function given earlier). The second function calculates the arcsine function that the Dragon lacks (see the earlier note on the trig functions).

Subroutines GOSUB and RETURN

The idea of creating new operations by defining functions is a very powerful one, but it is already possible to see its shortcomings. Sometimes it would be an advantage to give a name, not just to one line of BASIC in the form of a user-defined function, but to a whole collection of lines. This is the idea behind a subroutine. A subroutine is nothing more than a group of BASIC statements that can be used as often as required just by writing their 'name'. (In the same way that a single line of BASIC can be used as often as required by using a user-defined function's name.) The trouble with BASIC subroutines is that they are very limited. You cannot even give a one-letter name to a subroutine. It has to be referred to by the line number of its first line and there is no provision for parameters of any sort. Even with these restrictions the BASIC subroutine is still well worth knowing about and using.

If the lines of BASIC that go to make up the subroutine start at line *n* then you can use the subroutine by

GOSUB *n*

Where GOSUB stands for GO to SUBroutine. Indeed, the action of a GOSUB is very like GOTO in that it transfers control to line *n*. The difference between a GOTO and a GOSUB is that the GOSUB command causes the Dragon to store the line number of the GOSUB in a special area of memory set aside for the purpose. This stored line number is used by the RETURN statement to transfer control to the line following the GOSUB when the subroutine has finished. For example, the effect of a GOSUB and a RETURN on the flow of control is

```

      10 GOSUB 10000 - - - - -
- - >20 PRINT A
      30 ...
      40 rest of program
      10000 A=56 <- - - - -
- - -1010 RETURN

```

Line 10 transfers control to the subroutine that starts at 1000, which simply stores 56 in the variable A. The RETURN statement at line 1010 ends the subroutine and automatically transfers control back to line 20.

You can use *any* BASIC statement that you can use elsewhere in a program within a subroutine. In particular, there is nothing to stop a subroutine from using GOSUB and transferring control to another subroutine. If you do this then the next RETURN will transfer control back to the statement after the most recent GOSUB. In other words, if one subroutine calls another then RETURN behaves as you would expect it to, by transferring control back to the place where each subroutine was initiated, i.e. a RETURN never forgets where it came from!

This pair of instructions GOSUB and RETURN are all that there is to the BASIC subroutine. All the variables in a subroutine are the same as the variables in the rest of the program, there are no parameters of any kind. As suggested earlier this might lead you to believe that subroutines are not very useful – this is far from the truth.

Using subroutines

If you read the *Dragon Manual* (Chapter 5) you are lead to believe that the most useful way to use a subroutine is to replace any piece of program that is needed more than once. For example, if in a large program you need to print the same message over and over again, then it is better to turn that line into a subroutine and GOSUB to it every time it is needed. Although this is an important use of subroutines it is often the case that it is a good idea to form parts of a program into subroutines even if each part is only used once! The reason for this is that programs that use subroutines are easier to understand, easier to find mistakes in and easier to modify. This is not the sort of statement that anyone can prove because what is easier in this context is clearly a matter of opinion. The use of subroutines in writing BASIC programs will be illustrated by the examples in the rest of the book. If you discover some other method of programming that you like better, then no one will be able to argue with you! All I can say is that a great many programmers agree that subroutines are a good thing!

No matter how you use subroutines, there are two applications that you cannot afford to ignore. In Chapter Four the idea of grouping together BASIC statements using the colon was

introduced. This was then used in the two extended versions of the IF statement – IF .. THEN ‘list’ and IF .. THEN ‘list 1’ ELSE ‘list2’ – to avoid complications in the flow of control. The only problem was that a list of BASIC statements on a single line separated by colons soon becomes difficult to read and this limits the usefulness of these forms of the IF statement.

Now we have another way of grouping statements together that doesn’t become confusing no matter how many statements are involved. So instead of using

IF ‘condition’ THEN ‘list of BASIC statements’

use

IF ‘condition’ THEN GOSUB xxx

where xxx is the line number where the ‘list of BASIC statements’ used in the first version starts once it has been converted into a subroutine, i.e. with one statement to a line and ending with a RETURN statement. And instead of using

IF ‘condition’ THEN ‘list 1’ ELSE ‘list 2’

use

IF ‘condition’ THEN GOSUB xxx ELSE GOSUB yyy

where xxx is the line number where the subroutine containing the BASIC statements of ‘list 1’ starts and, similarly, yyy is the line number where the subroutine containing the BASIC statements of ‘list 2’ starts. Remember if a list of instructions is more than a few statements long – turn it into a subroutine!

Chapter Seven

Low Resolution Graphics and Colour

No matter what you intend to use your Dragon for, sound and graphics are likely to play an important part. Almost any program can be improved by the addition of graphics and carefully chosen sound. At its simplest, graphics can just involve positioning output where you want it on the screen. At its most ambitious it is about drawing lines, circles and other shapes in colour. This chapter introduces the *low resolution* graphics facilities of the Dragon and explains how they can be used in two and four colours. Chapter Eight then presents a discussion of the sound capabilities of the Dragon and how sound can be combined with the graphics commands already discussed. Chapter Nine then examines *high resolution* graphics and finally Chapter Ten deals with the more advanced graphics techniques available.

Most people are confused by the apparently vast number of ways that the Dragon has of producing graphics – its so-called *graphics modes*. However, each of these modes has its own particular advantages and disadvantages that govern when each should be used. Each mode differs from the rest in the size of the smallest dot that you can produce on the screen and this obviously governs the level of detail that you can include. Each mode also differs in the number of colours that you can use and as a general rule the more colours that you can use the less detail you can include. Choosing which graphics mode would be best for your application is therefore a matter of deciding on a balance of colours and detail. However, it is important to realise that the Dragon has two very different ways of producing a TV display. The first method is used to produce the text display that you see as soon as you switch on. In fact this text screen can also produce limited graphics and so this mode is called *text/low resolution graphics mode*. The second method of producing a screen display is so different from the first that it cannot be used directly to display text. All you can do in it is to plot points, lines, circles, etc.,

and as the degree of detail that you can use is high this method is referred to as *high resolution graphics*. This simple distinction is only confused by the fact that there are five different combinations of resolution and colour range available within high resolution graphics. These are referred to as the five high resolution graphics modes.

So to summarise – the Dragon has six graphics modes which consist of one low resolution text/graphics mode and five high resolution modes. The low resolution text/graphics mode is the easiest of all the modes to use and is controlled using the familiar PRINT statement and two new commands SET and RESET. For this reason, and because the low resolution commands lead naturally on to the high resolution commands, we start off by considering the text/graphics mode in the rest of this chapter.

Controlling PRINT

So far we have used the PRINT statement to print numbers and strings on the screen, either one to a line or next to each other on the same line. Although this is sufficient for most programs there is often a need to control exactly where something will be printed on the screen. In Chapter Three the PRINT statement was defined as

PRINT 'print list'

where 'print list' was defined to be a list of items, each one separated by a semicolon. The need for the semicolons is simply to show where one item ends and another begins. For example PRINT AB will try to print a single variable called AB, while PRINT A;B will try to print two variables A and B next to each other on a single line. The Dragon actually allows the use of two symbols to separate print items and each has a different effect on the layout. The semicolon, ';', that we have been using since Chapter Three simply means print the next item without leaving any space. Using a comma ',' as a separator means move to the next print 'zone' before printing the next item. The Dragon's screen is divided into two print zones – the first sixteen printing positions on a line and the last sixteen positions on a line. So PRINT "A","B" will print "A" in column one and "B" in column seventeen. You can place more than one separator between any two print items without any ill effects, e.g. PRINT "A",,"B" will cause the Dragon to move on two print zones before printing "B". If you think about it, this means that "A" will be printed at the

beginning of a line and “B” will appear at the beginning of the next line. There is one special case that is worth commenting on. If either of the print separators is placed at the end of a list of print items the automatic starting of a new line is suppressed. For example:

```
PRINT “A”,  
PRINT “B”
```

is the same as PRINT “A”, “B”, and

```
PRINT “A”;  
PRINT “B”
```

is the same as PRINT “A”; “B”.

Finally, before moving on to other things it is worth recalling that a blank line can be left by simply using

```
PRINT
```

without any items in the print list.

PRINT @

The use of the two print separators has certainly increased our control over how things are printed on the screen, but we still cannot easily make a print item start at a particular column on a particular line.

To achieve this we need something more than different separators. There are two ways to specify the exact position of a character on the screen: you can either give its line and column numbers or you can state how many character locations you had to move from the start of the screen to reach it. So to reach the top left-hand character you would have to move zero character locations from the start of the screen – the top left-hand character location *is* the start of the screen! To reach the character location immediately to the right of the start of the screen you have to move one location. To reach the character location at the start of the next line you have to move 32 character locations, and so on. If you work out how far each character location is from the start of the screen you will get a pattern of numbers as shown in Fig. 7.1. The first character location of every line is a multiple of 32, and the other character locations on the line take intermediate values. The Dragon uses this numbering of the character locations in the PRINT@ statement to enable you to

start printing anywhere on the screen. The format of the command is

`PRINT @‘character location’, ‘print list’`

where ‘character location’ is the number of the character location as just described where the first print item in ‘print list’ will be printed. Notice that there has to be a comma between the two parts of the statement. For example:

`PRINT @64, “HELLO”`

will print HELLO at the start of the third line.

You can use `PRINT @` to move anywhere on the screen no matter where the current printing position is and without altering anything else on the screen. For example:

`1Ø PRINT @128, “THERE”`
`2Ø PRINT @64, “HELLO”`

will first print THERE on the fourth line and then HELLO on the second line. You can even alter parts of existing lines – for example:

`1Ø PRINT@128,“HELLO”`
`2Ø PRINT@136,“THERE”`

will first print HELLO at the start of the fourth line and then will print THERE on the same line but eight character locations further on. However, it is important to realise that `PRINT @` will clear the remainder of the line after the last character printed to green. So

`1Ø PRINT @136,“THERE”`
`2Ø PRINT @128,“HELLO”`
`3Ø GOTO 3Ø`

won’t work because the second `PRINT` clears the THERE from the remainder of the line. If you end a `PRINT @` with a semicolon then it will not clear the remainder of the line, so allowing you to print anywhere on the screen without disturbing anything else. So to make the previous example work, add a semicolon to the end of line `2Ø`.

Using `PRINT @` you have complete control over where something will be printed on the screen. The only trouble is that the numbering system used for the character locations is a little difficult to use. It is easier to specify a particular line and column number than the single character location number used in `PRINT @`. However, there is an easy way to use `PRINT @` to print at a given line and column number. If you number the lines starting with line zero at the top of the screen

and number the columns starting with zero at the far left, then to print at line Y and column X use:

```
PRINT @X+32*Y,'print list'
```

So to print "A" on the top line and ten characters in you would use

```
PRINT @9+32*0,"A"
```

and

```
PRINT @15+32*8,"A"
```

would print the letter A on the ninth line and 16 characters in. The only point to watch with this simple method is that the numbering of the lines and columns begins at zero and not one.

The reason why this method is so useful is that it allows you to think of positions on the text screen in the same way that positions are specified in all of the other graphics modes. You can specify any location on the screen by first giving its horizontal position, usually called the *x co-ordinate*, and its vertical position, usually called the *y co-ordinate*. In the case of the text screen the column number is the *x co-ordinate* and the line number is the *y co-ordinate*. The value of thinking about screen positions in terms of a pair of co-ordinates is that it is easier to describe shapes using co-ordinates. For example, you can draw a rectangle of asterisks:

```
10 FOR I=0 TO 10
20 PRINT @(I+10)+32*2,"*";
30 PRINT @(I+10)+32*12,"*";
40 PRINT @10+32*(I+2),"*";
50 PRINT @20+32*(I+2),"*";
60 NEXT I
```

This program may look a little complicated, but if you look at the PRINT statements you should be able to see that each one has the form PRINT @X+32*Y, ... For example, line 20 prints asterisks at character locations with *x co-ordinate* I+10 and *y co-ordinate* 2. As I varies from 0 to 10 this results in a row of asterisks being printed on line number 2 from column number 10 to column number 20. In the same way line 30 prints a row of asterisks on line 12. Lines 40 and 50 print vertical columns of asterisks by keeping the *x co-ordinate* fixed and changing the *y co-ordinate*.

So useful are *x* and *y co-ordinates* that you should always use PRINT @ to print at a particular line and column number on the screen in the form PRINT @x+32*y.

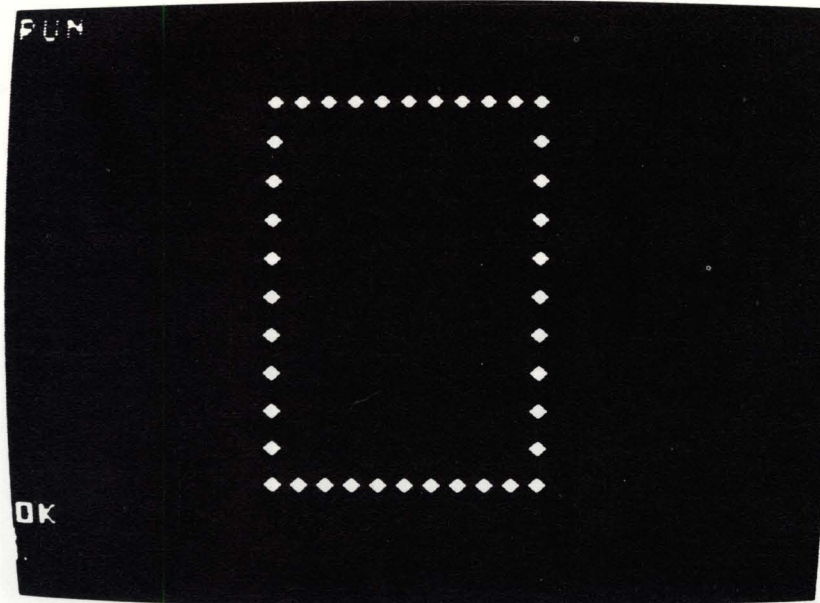


Fig. 7.2. Rectangle of asterisks.

Graphics characters

In addition to the usual letters of the alphabet, numbers and punctuation the Dragon has sixteen special characters called *graphics characters*. Each of these characters has a very simple shape obtained by dividing a character sized block into four smaller squares. What makes the shapes different is the combination of smaller squares that are black. To see these sixteen graphics characters try:

```
1Ø FOR I=128 TO 143
2Ø PRINT I, CHR$(I)
3Ø NEXT I
```

Apart from not being available from the keyboard and having larger character codes than normal, the graphics characters can be used in the same way as any other character. That is, they can be used in PRINT and PRINT @ statements and they can be used in strings. For example, you can change the rectangle printing program to use appropriate graphics characters instead of asterisks:

```

10 FOR I=1 TO 9
20 PRINT @(I+10)+32*2,CHR$(140);
30 PRINT @(I+10)+32*12,CHR$(140);
40 PRINT @10+32*(I+2),CHR$(138);
50 PRINT @20+32*(I+2),CHR$(138);
60 NEXT I
70 GOTO 70

```

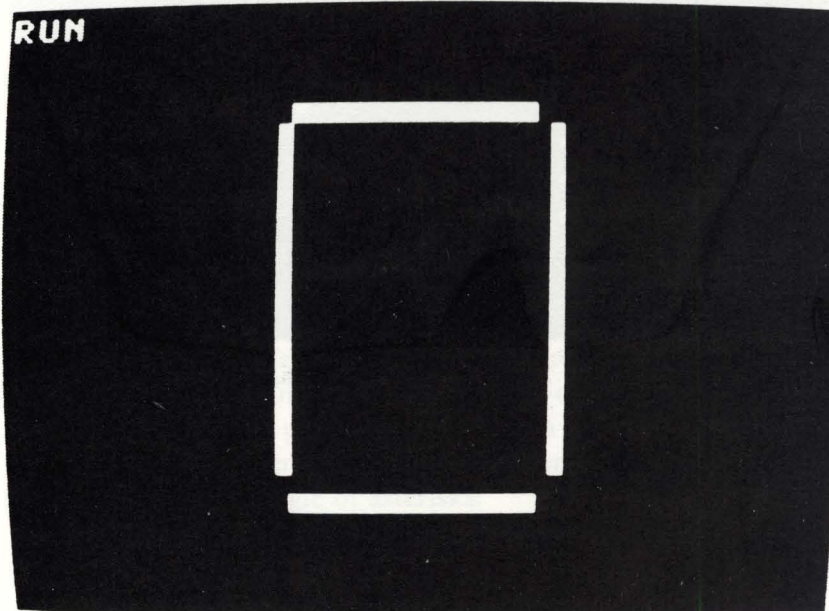


Fig. 7.3. Rectangle of graphics characters.

The only trouble with this rectangle is that the corners are missing! Putting them in is a matter of choosing the correct L-shaped graphics symbols and this is left for you to do.

As you can imagine, drawing more complicated shapes using PRINT @ and an appropriate choice of graphics characters is very difficult. In fact there is no need to worry, because the Dragon provides two additional commands, SET and RESET, that make drawing with the graphics characters easy. However, small numbers of graphics characters are often used in PRINT statements to produce simple solid shapes. For example if you want to print the outline of a ship during a game you could use

```

10 PRINT @X+32*Y,CHR$(136);CHR$(140);CHR$(140)

```


which will print a ship at line Y and column X. If you want to use a shape in more than one PRINT statement then it is better to define a string variable containing the graphics characters:

```
10 S$=CHR$(136)+CHR$(140)+CHR$(140)
20 PRINT @X+32*Y,S$
```

If you would like to see the ship move across the screen try

```
10 S$=" "+CHR$(136)+CHR$(140)+CHR$(140)
20 FOR X=0 TO 27
30 PRINT @X+32*5,S$
40 NEXT X
```

Notice the space at the start of the string S\$. If you want to know what the space is for, try leaving it out!

It is important to notice that you can mix text and graphics characters without any restrictions. For example, try

```
10 PRINT "HI THERE"
20 FOR I=1 TO 8
30 PRINT CHR$(131);
40 NEXT I
```

Graphics characters in colour — CLS

So far the graphics characters that we have been using have been black shapes on the usual green background. In fact, you can print the graphics characters as black on any one of eight different colours as the background. The colours are:

<i>Code</i>	<i>Colour</i>
1	Green
2	Yellow
3	Blue
4	Red
5	Buff (white)
6	Cyan (light blue)
7	Magenta
8	Orange

You can select the background colour used in a graphics character by using slightly different character codes. If CHR\$(X) is a graphics

character with a green background then `CHR$(X+16*(N-1))` is the same character but with a background colour given by N. For example, `CHR$(133)` is a black vertical bar on a green background, and `CHR$(133+16*(3-1))` or, in other words, `CHR$(165)`, is the same shape but on a red background.

As a demonstration of colour graphics characters the following program prints random shapes on random background colours.

```

10 X=RND(32)-1
20 Y=RND(16)-1
30 C=RND(8)
40 G=127+RND(16)
50 PRINT @X+32*Y,CHR$(G+16*(C-1));
60 GOTO 10

```

Lines 10 and 20 generate a random position as a pair of co-ordinates X and Y. Line 30 generates a number between 1 and 8 to choose the background colour. Line 40 generates a number between 128 and 143 to choose one of the standard (black on green) graphics characters. Finally line 50 prints the character at the selected place and in the selected colour.

It is important to realise that you can only select the background colour of the graphics characters. If you print normal text with graphics it will always be black on green or green on black, and there is no way that you can change this by using different character codes. (As will be explained later, there is a way to replace the green background by an orange one.) The fact that the graphics characters are always black on another colour considerably reduces their usefulness, but you can draw black shapes against backgrounds of different colours in different parts of the screen.

Graphics character 143 is very special from the point of view of colour as it contains no black at all. This means that you can print solid blocks of colour without interruption. For example, the following program prints colour bars:

```

10 FOR Y=0 TO 15
20 PRINT @0+32*Y,;
30 FOR C=1 TO 8
40 FOR I=1 TO 4
50 PRINT CHR$(143+16*(C-1));
60 NEXT I
70 NEXT C
80 NEXT Y

```

This is not an easy program to understand at first sight because it uses three nested FOR loops. The best way to understand its workings is to start with the inner loop and work out (this is very often the case with nested FOR loops). The innermost loop – lines 40 to 60 – prints the graphics character 143 four times in a row in the colour selected by C. This loop is contained within a loop – lines 20 to 70 – that repeats the inner loop eight times, once for each colour. The outermost loop repeats the whole process sixteen times, so printing the same pattern on each line. You may be a little puzzled by line 30. What it is doing is to put the next printing position at the start of each line.

Although you can't alter the background colour used to print text you can fill the whole screen with any colour using the CLS command.

CLS C

will clear all existing characters off the screen and fill it with the colour specified in C. The only additional fact to note is that you can also fill the screen with black by using the code zero. That is, CLS 0 fills the screen with black. This can be particularly useful in that if you print graphics characters on a black background all you can see are the parts in the *background* colour. In this sense the background colour now becomes the foreground colour. For example, go back to the example of the black ship graphic used earlier and ask yourself, how could you have a few different coloured ships on the screen at the same time? The answer is that you can't unless you use a black background. Try the following program:

```

10 CLS 0
20 S$=CHR$(135+16*2)+CHR$(131+16*2)+
   CHR$(131+16*2)
30 R$=CHR$(135+16*1)+CHR$(131+16*1)+
   CHR$(131+16*1)
40 PRINT @10+32*3,S$;
50 PRINT @20+32*3,R$;
60 GOTO 60

```

Line 10 clears the screen to black. Line 20 defines a blue ship and line 30 defines a yellow ship. Lines 40 and 50 print them on the screen. If you print anything on the screen then you will find that you still get black characters on a green background, and this of course spoils the uniform black background created by CLS 0. The solution to this problem is to print any messages on the screen using *inverse characters* – in other words in green on black.

SET and RESET

The Dragon's text screen is made up of 32 columns and 16 lines of character printing positions. In other words, the resolution of the text screen is 32 by 16. However, if you use the graphics characters then the resolution is higher. Since each graphics character is obtained by dividing a standard text square into four you should be able to see that the smallest point on the screen that you can alter is a quarter of a text square. This means that you could imagine the text screen divided up into 64 dots horizontally and 32 dots vertically. Any dot on the screen can be specified in the usual way using x and y co-ordinates, only now the x co-ordinate goes from 0 to 63 and the y co-ordinate from 0 to 31. This 64 by 32 screen is usually referred to as the *low resolution graphics screen*, but it should be obvious that in fact it is nothing new – it is just the familiar text screen used with the graphics characters. If you were given a pair of co-ordinates for the low resolution graphics screen then by careful choice of a graphics character you could make that point specified appear in any of the eight colours or black. This is a difficult job, however, and the Dragon provides two extra commands to relieve you of the hard work.

The SET command will set any low resolution point to any colour. The format of the command is

SET(x,y,c)

where x and y are the co-ordinates of the point and c is the code of the colour that you want it to be set to. The colour code can be in the range 0 to 8. The SET command sounds as though it is the answer to all our problems. If you want a point in a particular colour then just SET it. However, you cannot get away from the fact that SET works by printing graphics characters and this brings with it certain limitations. For example, if starting from a cleared green screen you SET a point near the middle to a colour, say blue then the SET command will select the graphics character with a single dot coloured blue and print it in the correct position on the screen. The trouble is that a graphics character with a single coloured dot also has three black dots in an L shape, and these also appear on the screen whether you like it or not. Now you should be able to understand why:

10 CLS

20 SET (32,16,3)

doesn't just produce a single blue dot in the middle of the screen, but a blue dot at 32,16 and three black dots.

Another problem with SET, brought about by the nature of the graphics characters that it uses, is that you can only SET one colour (and black) within each character location. In other words, if you SET a point to a particular colour then all the non-black points already present in the same character location will change colour! To see this try:

```
1Ø CLS
2Ø SET(32,16,5)
3Ø SET(33,16,5)
4Ø FOR I=1 TO 1ØØØ:NEXT I
5Ø SET(32,17,4)
```

Lines 2Ø and 3Ø set two dots within the same character location to white and then, after a short pause, produced by the FOR loop in line 4Ø, line 5Ø sets another point within the same character square to red. The result of this last SET is to leave the character location not with two white dots, one red dot and one black dot, but with three red dots and one black dot. This is because all the coloured dots within a character square take on the colour of the most recently SET dot within the square! This is bad enough on a green background, but if you start with a screen cleared to another colour then things are even worse. CLS clears the screen by filling each character location with a graphics character of the correct colour. If you SET a point anywhere on this screen all four dots within the character location are coloured and so they all change to the colour specified in the SET command! To see this try:

```
1Ø CLS 4
2Ø SET(32,16,5)
```

Plotting a white dot in the middle of a blue screen causes all of the dots within the same character location to change to white!

What all this means is that if you want to use SET to change the colour of individual dots on the screen you have to start with a black background. Try

```
1Ø CLS Ø
2Ø SET (32,16,5)
```

and you will only see a single white dot appear roughly in the middle of the screen. At last the SET command seems to be performing as promised! Of course the black dots are still present on the screen but

you can't see them against the black background! There is nothing that you can do about the restriction of only one colour per character location, but at least your drawings are not spoiled by unwanted black dots.

If you start from a black screen you can make a coloured dot appear by SETting it. You might think that you could make it disappear by SETting it to black, but if you try SET (x,y,Ø) you will find that it doesn't produce a black dot. To overcome this problem the Dragon provides an extra command:

RESET(x,y)

which sets the dot at x,y to black. To see a flashing dot try:

```
1Ø CLS Ø
2Ø SET (32,16,5)
3Ø RESET (32,16)
4Ø GOTO 2Ø
```

Line 2Ø SETs a dot to white and line 3Ø RESETs it to black.

There is no doubt that the Dragon's low resolution colour graphics are in general complicated to use and you may not have followed all of the details of how and why some of the strange effects explored above happen. However, it is now easy to give a summary of the conclusions about how low resolution colour graphics should be used:

- (1) Always start off with a black screen.
- (2) Use SET(x,y,c) to change the dot at x,y to the colour c and RESET(x,y) to change it back to black.
- (3) If you SET a dot then any other coloured dots within the same character location will change to the same colour.
- (4) RESETting a dot to black doesn't affect any other dots in any way.
- (5) If you want to mix text and graphics then use inverse characters (obtained by pressing SHIFT and Ø) in PRINT statements, and end all PRINT statements with a semicolon or you will clear the remainder of any line that you print on to green.

All these restrictions are unfortunate because they make low res graphics difficult to use. Ideally the SET command would allow you to set any point to any colour without worrying about affecting any other points on the screen. But for this sort of freedom we will have to move up to high resolution graphics. Low resolution graphics

however isn't all bad. Its single great advantage is that you can mix text and graphics in this mode. Also, it is the only mode the Dragon has that gives you the possibility of using eight colours even if there are restrictions.

As a simple example of low resolution graphics, consider the problem of drawing a rectangle again:

```
1Ø CLS Ø
2Ø FOR X=2Ø TO 3Ø
3Ø SET (X,1Ø,5)
4Ø SET (X,2Ø,5)
5Ø NEXT X
6Ø FOR Y=1Ø TO 2Ø
7Ø SET (2Ø,Y,5)
8Ø SET (3Ø,Y,5)
9Ø NEXT Y
```

If you want to see what happens if you ignore the rules given above, remove the Ø from the CLS in line 1Ø. The result may look interesting but it is not what was intended!

The POINT function

You can find out what is already on the screen by using the POINT function. Depending on what is on the screen at X,Y the function POINT(X,Y) returns the following results:

 -1 if the point is within a text character
 and 0-8 depending on the colour of a graphics character

The point function is not complicated and its major use is to avoid SETting or RESETting a point in an area of the screen that is already in use, or simply finding out what area of the screen you are at. For example, suppose you had written a game that involved a man walking across the screen and perhaps falling into quicksands. If you make the quicksands yellow you can easily test to see if the man has walked into the quicksands by using POINT(X,Y) and checking for yellow.

Using joysticks

The Dragon's graphics become much more interesting once you

couple them to the use of a pair of joysticks. The position of either of the joysticks can be found using the JOYSTK function:

JOYSTK(0)

gives the horizontal position of the left joystick, and

JOYSTK(1)

gives the vertical position of the left joystick.

JOYSTK(2)

gives the horizontal position of the right joystick, and

JOYSTK(3)

gives the vertical position of the right joystick.

Each one of the positions is returned as a number in the range 0 to 63. While this is ideal for specifying the X co-ordinate of a point in low resolution graphics, it is too big for the Y co-ordinate. However, dividing the result of JOYSTK(1) or JOYSTK(3) by two gives a number in the range 0 to 31, which can be used directly as the Y co-ordinate of a dot. For example, to control the position of a dot on the screen using the left joystick try:

```
10 CLS 0
20 X=JOYSTK(0)
30 Y=INT(JOYSTK(1)/2)
40 SET(X,Y,5)
50 RESET(X,Y)
60 GOTO 20
```

Line 10 gets the x co-ordinate and line 20 gets the y co-ordinate. Line 40 SETs the specified dot to white and line 50 RESETs it to black. By repeating these simple steps the point appears to move around the screen according to the position of the left joystick. You can use the right joystick to control the position of a second point on the screen in the same way.

The joysticks also have a built in 'fire' button, but unfortunately Dragon BASIC doesn't provide any function to test whether or not they are pressed. However, it is possible to define a function that will test either button. Try the following program:

```
10 DEF FNB(S)=(PEEK(65280) AND S)/S
20 PRINT FNB(1),FNB(2)
30 GOTO 20
```

If you press the left button then FNB(1) changes from 1 to 0 and if you press the right button FNB(2) changes from 1 to 0. How FNB works will be clearer after Chapter Eleven, but you don't need to know how it works to use it. For example by adding it to the previous joystick program you can plot a dot whenever the button is pressed:

```
10 DEF FNB(S)=(PEEK(65280) AND S)/S
20 CLS 0
30 X=JOYSTK(0)
40 Y=INT(JOYSTK(1)/2)
50 SET (X,Y,5)
60 IF FNB (1)=1 THEN RESET(X,Y)
70 GOTO 30
```

The program works by testing the state of the button (line 60) and only RESETting the point if the button isn't pressed.

Chapter Eight

Sounds and Games

The Dragon has a number of commands that control the production of sound over your TV's loudspeaker. These commands are not difficult to understand. However, it can take quite a lot of ingenuity to produce any sounds worth listening to! In the first part of this chapter we will examine some of the ideas involved in using sound to good effect. In the second part an example of a game involving both sound and low resolution graphics will be presented and explained. Although most of this game could have been written at the end of Chapter Seven it is remarkable how much excitement can be added to a game by the careful use of sound.

Simple sounds – SOUND

The Dragon's simplest sound command is

SOUND P,T

where P specifies the *pitch* of the note and T the *time* that it will be sounded for. For example

SOUND 80,16

will sound middle C for about one second. If you try this command and cannot hear anything then remember that the Dragon doesn't have a built-in loudspeaker, and any sound is produced by your TV set. To hear the result of a SOUND command you have to tune the set in carefully (use the test card in Chapter Two) and have the volume adjusted to a comfortable level. An important point about all the Dragon's sound producing commands is that all computing stops for the duration of the sound – the Dragon can only do one thing at a time.

To hear the range of notes that you have at your disposal try:

```
1Ø FOR I=1 TO 255 STEP 1
2Ø SOUND I,1
3Ø NEXT I
```

The pitch value must lie between 1 and 255, but the time that a note can last can be any number greater than zero. To determine how long a note will sound for, you need to know that T is measured in 4/50ths of a second. So a T value of 50 will make a sound for exactly 4 seconds.

If you know a little bit about music theory then you can use SOUND, or the more advanced PLAY command, to write your own tunes. However, if you would first of all like to hear what the Dragon can do on its own try:

```
1Ø SOUND RND(255),1
2Ø GOTO 1Ø
```

The noise that this program produces is interesting at first, but soon becomes boring. The trouble is that music that is too random just doesn't sound interesting. Although it is very difficult to introduce enough, order, into the computer-generated music to make it sound anything like traditional music, you can see the overall effect of increasing the order if you try the following program:

```
1Ø N=2ØØ
2Ø N=N+2*(SGN(RND(Ø)-.5))
3Ø SOUND N,1
4Ø GOTO 2Ø
```

This plays a long sequence of notes that go either up or down by one at most, and sounds just a little more like music than the first program. In fact it's not a bad imitation of the *Flight of the Bumble Bee*! Before leaving the subject of random music it is interesting to hear the effect of changing the note length in each of the above programs. Try substituting values of 1,5,1Ø and 15 in line 3Ø.

Programming tunes- PLAY

The SOUND command is fine for producing the occasional beep, but if you want to program tunes or play music the Dragon's PLAY command is much easier to use. The command

```
PLAY 'string'
```

will produce a number of distinct sounds according to what is stored in 'string'. Specifying the pitch of a note in `PLAY` is achieved by simply using the letter that names the note in music.

Traditionally the letters A to G have been used to stand for the eight distinct notes recognised by western music. It is important to understand that as far as music is concerned two notes with widely different pitches can have the same name. For example, if you can find an A key on a piano keyboard and play each of the white notes in turn you will play the notes ABCDEFG and then the next note that you play will be called A once again, even though it is much higher in pitch. Musically there are only eight notes, which can be played at any of a number of pitches. Obviously if you ask the Dragon to play the note A you have to have some way of specifying which of the many possible A's it is to play! To do this you have to use the O or Octave command which will select the pitch that you want the note played at. The way that it does this is confused by the fact it recognises that the most important note is C and not A. If you try the following two `PLAY` commands,

`PLAY "ABCDEFGG"`

and

`PLAY "CDEFGAB"`

you will discover that the first one doesn't produce a steadily increasing pitch as you might expect, but the second one does. The reason for this is that the Dragon's naming of notes starts with C as the lowest in pitch and then increases in the order DEFGAB with B as the highest note. If you want to play a note higher than B then you have to move to the next *octave* using the O command. The Dragon starts off by using the octave that includes the note called middle C on the piano. The octaves that the Dragon can produce are numbered 1 to 5, with the 'default' middle C octave numbered 3. So

`PLAY "O3CDEFGAB"`

will play the scale of C without the final high C to finish it, but

`PLAY "O3CDEFGABO4C"`

plays the full scale of C by changing to the next octave, O4 before playing the last note.

The idea of changing octaves to produce a note is not difficult once you get used to it. There is another way that the Dragon uses to name notes that emphasises the fact that the scale starts from C.

Instead of giving the note's name you can give it a number, with note 1 corresponding to C. The rest of the correspondence is not quite as simple as you might expect:

Note name	Note number
C	1
D	3
E	5
F	6
G	8
A	10
B	12

You will notice that the note numbers do not increase evenly. The reason for this is that there really isn't an equal 'distance' between the notes of the musical scale. If you can find middle C on the music stave then a note drawn on this line corresponds to a pitch of 1. Moving up or down by one place on the stave increases or decreases the pitch by 2 or 1 (see Fig. 8.1). The reason for this is that notes

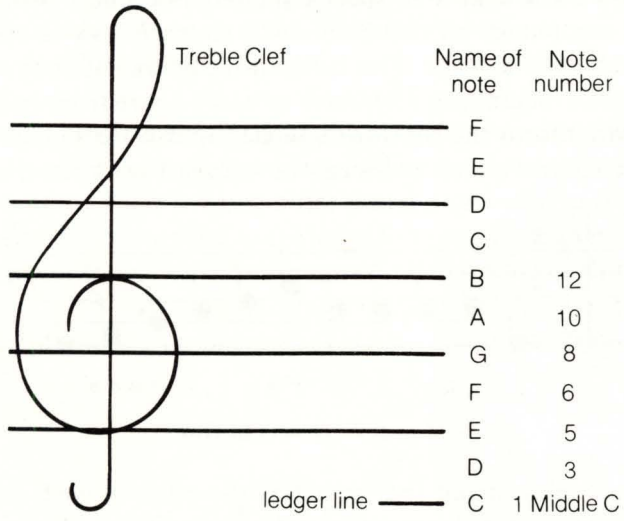


Fig. 8.1.

sometimes differ by a whole tone and sometimes by only a *semitone*. The pattern of tone/semitone differences is easy to remember because it is exactly the same as the arrangement of black and white notes on the piano. For example, starting from C gives the following

pattern of tone/semitone differences:

C — D — E — F — G — A — B — C

T T S T T T S

An additional complication is that most music involves the use of sharps and flats, i.e. the notes produced by the black keys on a piano. These are easy to deal with once you realise that a sharp raises the value of the note by one and a flat lowers it by one. For example, C is 1 and so C sharp is 2, E is 5 and so E flat is 4. The only thing that you have to remember is that if a note is shown as sharp or flat at the start of the music in the *key signature* then it and all its octaves must be sharpened or flattened. If you use the note names instead of numbers you can indicate a sharp by + or the traditional symbol # and a flat by -. So PLAY "2;4" and PLAY "C#E-" will both play the notes C sharp and E flat. The use of the semicolon in the first PLAY command is necessary to separate the note numbers clearly. You can use a semicolon to improve readability anywhere in a play string.

Now we know how to specify the pitch of a note produced by a PLAY command, all that remains is to learn how to specify how long each note lasts for. This is best done by way of an example. The well known beginning of *Hearts of Oak*, apart from being a good tune, could form the basis for a jingle suitable for a game involving ships. The first eleven notes can be seen in Fig. 8.2 and converting



Fig. 8.2. 'Hearts of Oak'.

them to a play string that specifies the pitch of each note is easy enough. The three sharp signs at the beginning apply to all G's, F's and C's in the tune, and this rule is best applied by writing the name of each note underneath and then writing a sharp sign by each G, F and C. The only complication in translating these notes' names into a play string is the need to change octaves to reach C# above the fourth A. The resulting play string is

"O3;EAAAA;O4;C#;O3;BAG#F#E"

Notice that as well as changing from O3 to O4 to play the C# in the middle of the tune you have to remember to change back to O3 for the rest of the notes – an O command remains in effect until it is countermanded!

Only one thing now keeps us from hearing *Hearts of Oak* and this is the problem of how long each note should be sounded for. Fortunately, musical notation is rigorously logical (after all it is one of the first programming languages!). Time is divided into intervals and a plain ordinary note, like the first in *Hearts of Oak*, should last one interval. The time that a note lasts is shortened by the number of ‘streamers’ drawn on its tail. Each streamer halves the length of the note. For example, the fourth note has two ‘streamers’. The first shortens it to half an interval and the second reduces it to a quarter. The only complication is that a dot following a note is an instruction to lengthen it by half the time that it would normally last (it makes you wonder how musicians cope!). So the third note would normally be one half a time interval, but because it is followed by a dot it has to be sounded for one half plus one quarter, i.e. three quarters. Translating this musical notation into the play string is easy. The L command can be used to set the length of all the notes that follow it. L1 indicates a whole time interval, L2 half the time interval, and so on. Notice that the number following the L indicates the fraction of the time interval that a note should sound for. This is very close to the traditional musical notation, and to make it even closer you can follow an L command by a dot which will increase the time interval by half its length. There are two notes that do not occur in *Hearts of Oak* that have to be sounded for twice as long and four times as long. These are included in Fig. 8.3, along with all the other note lengths.

Note	Time
	4
	2
	1
	$\frac{1}{2}$
	$\frac{1}{4}$
	$\frac{1}{8}$

Fig. 8.3. Lengths of notes

If we take the longest note that we could ever want to sound as one time interval, then a black note without a tail corresponds to L4 and a note with a single tail lasts for L8. Using all this information we can write a PLAY command for *Hearts of Oak*:

```
10 PLAY "O3;L4;EA;L8.;A;L16;A;L4;A;L8;O4;C#;O3L16;B;
    L4;A;L8.;G#;L16;F#;L4.;E"
```

This command may look complicated but if you study it you should be able to see that it is just the play string for the pitch values of *Hearts of Oak* given earlier, combined with a specification of the length of time each note should last for.

If you find this rendition of *Hearts of Oak* a little too slow for a games jingle then you will be pleased to hear that it can be played faster by using the T or tempo command. The tempo command sets the actual time that a note lasts for. The default value is T2 and increasing the number that follows the T will increase the tempo that the music is played at. In other words it will decrease the time that each note lasts for. Try adding T4 to the front of the play string and then experiment with different tempos down to T1 and up to T155.

One effect that can cause trouble unless you know about it is that the conditions of tempo, note length, volume and octave set by a PLAY command remain in force until the end of the program. So if you use more than one PLAY command in a program you might be surprised that later PLAY commands start off with odd values of octave, volume, etc.

Advanced PLAYing

The play commands O,L and T introduced in the last section give you enough freedom to write convincing play strings for most tunes. However, there are a few other play commands that extend the usefulness of the PLAY command.

You can insert a pause (a period of silence) into a play string by using the P command. This has the same format as the L command, except that you cannot use a dot to increase the period of a pause. So P1 pauses for one time interval, etc.

You can specify the volume of each note by using the V command. For example, V31 will cause all the notes following it to have the maximum volume and V1 will cause them to have the smallest volume. If you don't use a V command then the default value of V15 is used. In other words, all the music that we have heard so far has been at half volume!

The final play command is the most difficult to understand, but it is very useful. You can get the PLAY command to execute a list of commands stored in a string variable as part of the play string by using the X command. For example, if you have a piece of music stored in A\$ then

```
10 PLAY "T2O2XA$;T4O1XA$;"
```

will play it twice. Notice that you have to follow the name of the string variable by a semicolon to show where it ends.

In addition to these three extra commands you can alter the current value of the octave, volume, tempo and note length by using the following notation:

- + adds one to current value
- subtracts one from current value
- > multiplies current value by two
- < divides current value by two

So for example if the current octave was 2, O+ would change it to 3.

The most advanced use of the PLAY command relies on the fact that the play string is indeed a string, that can be manipulated just like any other string. For example if you want to play a note sequence with increasing volume you could use

```
10 FOR I=1 TO 12
20 A$="V"+STR$(2*I)+";" +STR$(I)
30 PLAY A$
40 NEXT I
```

The play string is made up in A\$ by line 20. The STR\$ function is used to convert numbers into strings.

Using this idea of building up a play string it is possible to turn the Dragon into a keyboard instrument! Try the following simple program:

```
10 DATA C,D,E,F,G,A,B,O4CO3
20 DIM N$(8)
30 FOR I=1 TO 8
40 READ N$(I)
50 NEXT I
60 A$=INKEY$
70 IF A$="" THEN GOTO 60
80 PLAY N$(VAL(A$))
90 GOTO 60
```

This will allow you to play notes with the top row of number keys from 1 to 8. It works by continually ‘scanning’ the keyboard using INKEY\$ (see Chapter Six). Each number is entered as a string variable and is then converted from a string to a number by the VAL function and used in the PLAY command (line 30) to control the pitch. The array N\$ holds the names of each note from middle C to C an octave above. Notice that you have to change octave up one for the last C and have to remember to change it back again! You should be able to write a program to make every key on the keyboard produce a different note. What is more difficult is to find an arrangement of keys and notes that makes the Dragon easy to play!

Some sound effects

The trouble with using the Dragon to produce sound effects is that while any noise is coming from the loudspeaker it stops doing any calculations. So even if you write a program that makes an excellent sound of a rocket taking off it would be difficult to add to any simultaneous graphics of a rocket taking off. The best you could do is to make a bit of noise, move the rocket a little, then make a bit more noise, then move the rocket a little more, and so on. A continuous ‘whoosh’ accompanying a moving rocket is not something that can be achieved from BASIC alone.

However, you can make some very interesting sounds using nothing more than the PLAY command. If you write a play string for a sequence of notes at different volumes and PLAY it at a very high tempo you don’t hear each individual note, just a single sound. For example, try

```
10 S$="T155BCBCBCBCBCBCBCBCBCBCBCBCBC"
20 PLAY S$
```

which produces a warbling noise. More complicated play strings can be made up using loops. For example:

```
10 FOR I=1 TO 31 STEP 2
20 S$=S$+"V"+STR$(I)+"CB"
30 NEXT I
40 S$="T155"+S$
50 PLAY S$
```

which produces a very interesting warbling ‘zap’ type noise. To produce other sound effects is a matter of experimentation along

these lines. When dealing with play strings it is important to remember the CLEAR command introduced in Chapter Five for releasing memory for use by strings.

A speaking Dragon

SOUND and PLAY are the only two proper sound commands that the Dragon has. However, there are two commands which can be used to control a tape recorder connected to the Dragon that open up some very interesting possibilities.

If you have the remote control connected to your tape recorder then the command MOTOR ON will switch its motor on and MOTOR OFF will similarly turn it off. Of course these commands will only have any visible effect if the tape recorder is set to play or record. One simple use of the MOTOR command is to switch the motor on so that a tape can be rewound.

A more interesting use is to be found in conjunction with the AUDIO command. Following AUDIO ON any signal played back on the tape recorder will be relayed through the TV's loudspeaker. AUDIO OFF will stop any sound from the tape recorder coming through the TV's loudspeaker. To see the way these commands work, place a tape with talking or music into your cassette recorder, and try the following program:

```

10 PRINT "PRESS PLAY AND THEN ANY KEY WHEN
   READY"
20 IF INKEY$="" THEN GOTO 20
30 MOTOR ON
40 AUDIO ON
50 TIMER=0
60 IF TIMER/50<60 THEN GOTO 60
70 AUDIO OFF
80 TIMER=0
90 IF TIMER/50<60 THEN GOTO 90
100 GOTO 40

```

You will be amazed to hear the sound from the tape come over the TV's loudspeaker for one minute to be followed by one minute's silence.

Practical applications of AUDIO and MOTOR are likely to be much more involved than this. For example, if you wanted to add a commentary to something that was happening on the screen you

would have to record a special tape with the voice precisely timed. One way of synchronising audio material on the tape is to intersperse it with small data files. You could move on to a new part of the audio tape by simply opening and reading a data file of the correct name. To avoid the noise of the computer data coming through the TV set you would have to remember to turn the AUDIO OFF, but apart from that the tape would wind on to the beginning of the data file. Once at the beginning of the data file you would then read it to its end and then after a short delay turn the AUDIO ON to produce the desired sound over the TV set. All this requires a lot of planning!

Attack the saucer - a low res game

The game listed below uses most of the BASIC commands that have been introduced in this chapter and Chapter Seven and demonstrates the way in which sound effects can be used to enhance a graphics game.

```

10 T=15
20 F=0
30 FX=0
40 FY=6
50 GOSUB 1000
60 FOR X=0 TO 29
70 GOSUB 2000
80 GOSUB 3000
90 PRINT@X+32*15,CHR$(128);CHR$(143);
100 NEXT X
110 FOR X=29 TO 0 STEP -1
120 GOSUB 2000
130 GOSUB 3000
140 PRINT@X+32*15,CHR$(143);CHR$(128);
150 NEXT X
160 GOTO 60

1000 SS=CHR$(128)+CHR$(135)+CHR$(139)+CHR$(128)
1010 CLS 0
1020 RETURN

2000 F$=INKEY$
2010 IF F$="" THEN RETURN

```

```

2020 SOUND 10,1
2030 PRINT@FX+32*FY, CHR$(128);
2040 FX=X
2050 FY=15
2060 F=1
2070 RETURN

3000 T=T+RND(0)*2-1
3010 IF T<5 THEN T=T+1
3020 IF T>25 THEN T=T-1
3030 PRINT@T+32*5,SS;
3040 IF F=0 THEN RETURN
3050 PRINT@FX+32*FY,CHR$(128);
3060 FY=FY-1
3070 PRINT@FX+32*FY,CHR$(126);
3080 SOUND 50-FY,1
3090 IF FY<4 THEN F=0:PRINT@FX+32*FY,CHR$(128);
      :RETURN
3100 IF POINT(2*FX,2*FY-2)=0 THEN RETURN
3110 GOSUB 4000
3120 RETURN

4000 FOR I=1 TO 15
4010 PRINT@T+1+32*5,CHR$(143+(RND(8)-1)*16);
      CHR$(143+(RND(8)-1)*16);
4020 SOUND RND(200),1
4030 NEXT I
4040 RETURN

```

The game itself is relatively straightforward to play. An alien 'flying saucer', in the form of two graphics blocks, moves rather jerkily across the screen. A ship, whose purpose is to attack the alien, moves rapidly backwards and forwards at the bottom of the screen. A missile can be launched at any time from the attacking ship by pressing any key. Once a missile has been fired it moves up the screen accompanied by a whistling noise, increasing in pitch, until it either misses the saucer or hits it with a resulting explosion. If at any time during the flight of a missile another key is pressed, then the first missile is erased from the screen and a new missile fired at the saucer.

The program has been written as a small collection of subroutines and is not particularly difficult to understand. It is easier to follow the main part of the program after a description of each subroutine.

Subroutine 1000–1020 sets up a string S\$ containing the graphics blocks that define the saucer and clears the screen to black. Subroutine 2000–2070 checks to see if any key has been pressed and fires the missile. If no key has been pressed then control is returned to the main part of the program (line 2010). If any key has been pressed then any existing missile is removed from the screen by printing a black graphics block (line 2030) at the current missile position stored in FX and FY. Then the current missile position is set to the current position of the attacking ship (lines 2040–2050) and variable F is set to 1 to indicate that a missile has been fired and is in flight. Subroutine 3000–3120 moves the saucer a random amount to the right or left, prints the missile if one is in flight and checks to see if it has hit the saucer. Lines 3000–3030 are responsible for moving the saucer. Notice the checks to stop it from moving off the edge of the screen in lines 3010 and 3020. The printing of the saucer in line 3030 also serves to remove the old saucer from the screen because of the black graphics blocks included at either end of the string graphics blocks. Lines 3040–3080 look after moving the missile. Line 3040 checks to see if there is a missile in flight (i.e. F=1). If there isn't, control is passed back to the main part of the program. Lines 3050–3080 move the missile up the screen by one line. Line 3050 blanks out the old missile and line 3070 prints it at its new position. Line 3080 makes a sound that increases in pitch as the missile moves higher up the screen. Lines 3090–3110 test to see if the missile has hit or missed the saucer. Line 3090 tests to see if the missile's position is such that it has passed the saucer and is about to go off the screen. If this is the case a black graphics block is printed to remove the missile and the variable F is set to zero to indicate that there are no missiles in flight. Line 3100 uses the POINT function to discover what is at the next screen location that the missile will move into. If POINT returns 0 then the missile is about to move onto a black location, otherwise it is about to hit something. If it is, then the missile is about to hit the saucer and the explosion subroutine is called.

Subroutine 4000–4040 is the explosion subroutine. This works by simply printing blocks of random colour over the current position of the saucer and making random noises to go with it.

Now that all the subroutines have been described, the working of the main part of the program is easy to understand. Lines 10–40 set up values of some of the important variables – T is the horizontal position of the saucer, F is used to indicate when a missile has been fired and FX and FY are the co-ordinates of the missile. Then subroutine 1000 is called to set up the saucer string and clear the

screen to black. The main work of the program is done by the two FOR loops 60–100 and 110–150. The first FOR loop moves the attack ship to the right one place at a time. Each time the attack ship moves, subroutine 20000 is called to check for a ‘fire missile’ command and subroutine 30000 is called to move the saucer and the missile. The second FOR loop moves the attack ship to the left, but otherwise it is identical to the first FOR loop.

This concludes the description of this short games program. If you study it to the point that you are sure that you understand it, the way to find out if you’re right is to try to modify it! The game would be made much more exciting by the addition of only a few very simple features. You could, for example, add a routine to keep a score of the number of saucers hit, or give the attacking ship only a limited number of moves before the saucer fires a missile back at it! Try experimenting with these suggestions and your own ideas. After all, the only way to learn to program is to program!

Chapter Nine

High Resolution Graphics

The presence of high resolution, or hi res, graphics is something that might have lured you into buying your Dragon. After reading Chapter Seven on low res graphics you could be forgiven for expecting that hi res graphics are more difficult. However, hi res graphics are free of the limitations that make low res graphics tricky. In hi res you can plot any point in any of the available colours without affecting any of the other points. The price to be paid for this freedom is that you cannot *print* onto a hi res screen, although as we shall see in Chapter Ten there is a way of producing alphanumeric characters.

As explained in Chapter Seven, the Dragon has five different hi res graphics modes, but the only way that these differ is the number of points that you can plot, i.e. the *resolution*, and the number of colours that you can use. In particular the new commands that you need to control the hi res graphics screens are the same for all the modes. This means that we can describe the new commands without worrying about which of the four graphics modes they are going to be used in. This is not to say that there aren't important differences between the way that the graphics modes are used, but these can be discussed later.

The high resolution modes

The Dragon uses a completely different method for displaying its high resolution screens. This means that we will have to learn a new set of commands for changing the colours of points and drawing shapes. However, the basic method of specifying which point you want to alter – using x and y co-ordinates – remains the same. No matter what hi res mode you are in, the Dragon will accept co-ordinates as if the screen was composed of 256 points horizontally

and 192 points vertically, with point $\emptyset, \emptyset$ in the top left-hand corner. The term 'as if' is used because in all but one of the modes there are in fact fewer points than this. You may think that this is unnecessarily complicated, but imagine the alternative of using a different range of co-ordinates for each of the modes! Using a single set of co-ordinates means that you can write a graphics program in one mode and if it turns out to be unsatisfactory try it out in another mode without changing it. This will become clear after we have examined the resolutions available in each of the graphics modes.

The five modes also differ in the number of colours that can be displayed on the screen at the same time. The resolution available within each mode and the number of colours that you can use are summarised below:

Mode number	Resolution horizontal $\times$ vertical	Number of colours
0	128 $\times$ 96	2
1	128 $\times$ 96	4
2	128 $\times$ 192	2
3	128 $\times$ 192	4
4	256 $\times$ 192	2

Notice that the horizontal resolution is the same in all the modes apart from mode 4. Treating the screen as if it was always composed of 256 dots horizontally by 192 dots vertically is fine in mode 4, because in this mode there really are this number of dots. So for every pair of co-ordinates that you can give there is really a dot on the screen that you can specify the colour of. However, in all the other modes there are only 128 dots horizontally, even though you

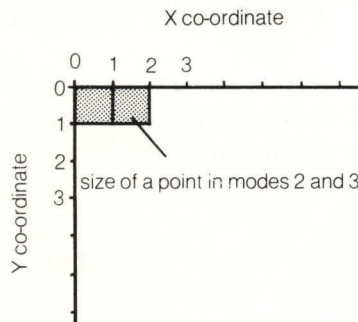


Fig. 9.1.

are still going to specify the x co-ordinate in the range 0 to 255. What this means is that some co-ordinates will refer to the same dot on the screen. So, if in Mode 3 say, you try to set point 0,0 to green and point 1,0 to red you will fail because 0,0 and 1,0 refer to the same point! As there are only half the number of horizontal points that the maximum x co-ordinate would suggest, you have to increase the x co-ordinate by 2 before you move on to a new point (see Fig. 9.1). In modes 1 and 0 the number of vertical dots is only half the number that the maximum y co-ordinate would suggest, and in this case both the x and y co-ordinates have to increase by two to move on to a new point (see Fig. 9.2). In practice for much of the time

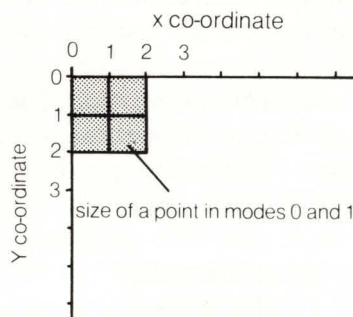


Fig. 9.2.

you can just forget about which co-ordinates correspond to separate points on the screen and just treat all resolutions as if they had 256 by 192 points. However, some of the unexpected results that you might see when using hi res graphics can be explained by not having as many points on the screen as you might think you are working with!

Getting into hi res – PMODE and SCREEN

There are two commands that affect what display mode you are in – PMODE and SCREEN. To select one of the hi res modes all you have to do is give the command

PMODE mode

where mode is 0 to 4 depending on which mode you want to use. So if you want to get into mode 4 all you have to write is PMODE 4. If you do try this command, however, you may be puzzled to see that it has no effect at all! The reason for this is that although the PMODE

command selects which mode you are drawing in it doesn't select what screen is being displayed. The command

SCREEN type,colours

must be used to select which screen is displayed, where 'type' is 0 for the standard text screen/low res screen described in Chapter Seven, and 1 for the hi res screen selected by the most recent PMODE command. The second parameter 'colours' is either 0 or 1 and gives you a little extra flexibility in that it determines which of two possible sets of colours the screen will be displayed in. What the colours are and how they are used will be described in the next section. For the moment we will concentrate on how to select and display a graphics screen and will use a standard value of 0 for 'colours'.

So to summarise, the PMODE command will select the hi res mode that you are drawing in, but SCREEN actually determines which of text or graphics is displayed. For example, to see a mode 2 screen you have to use:

```
10 PMODE 2    select the mode
20 SCREEN 1,0 display hi res screen
```

If you run this program you might just see a flash as the hi res screen is displayed for a moment! The Dragon switches back to the text screen as soon as the program stops or if you use a PRINT statement within your program. To see this, try the following:

```
10 PMODE 2
20 SCREEN 1,0
30 FOR I=1 TO 1000:NEXT I
40 PRINT "TEXT SCREEN"
50 FOR I=1 TO 1000:NEXT I
60 SCREEN 1,0
70 GOTO 70
```

The first two lines select mode 2 and display the hi res screen as in the previous example. Line 30 causes a delay long enough for you to look at the mode 2 screen, which will probably have nothing but random rubbish on it. Following this delay, line 40 prints to the text screen (remember you cannot print to the hi res screen). This causes the text screen to be displayed. Notice that whatever was on the text screen before you selected the hi res screen is still there, as well as the new message printed by line 40. After another delay (line 50) the hi

res screen is again displayed because of the SCREEN command at line 60. You might be able to notice that the hi res screen is still displaying the same random rubbish it was the first time you saw it. The important point to note is that the text screen and whatever high res screen you are using still exist when they are not being displayed. Finally line 70 causes the program to loop until you press the escape key, when you will once again see the text screen.

The hi res commands – COLOR,PCLS,PSET and PRESET

Now that we know how to select a hi res mode and display it, all that remains is to find out how to alter the colours of the points on the screen. First, though, we need to find out what colours are available in each mode. As described in the last section, each of the modes can be displayed in one of two sets of colours as selected by the SCREEN command. These colours are for each mode:

<i>Mode</i>	<i>Colour set 0</i> (SCREEN 1,0)		<i>Colour set 1</i> (SCREEN 1,1)	
	colour	code	colour	code
0	black	0	black	0
	green	1	buff	5
1	green	1	buff	5
	yellow	2	cyan	6
	blue	3	magenta	7
	red	4	orange	8
2	black	0	black	0
	green	1	buff	5
3	green	1	buff	5
	yellow	2	cyan	6
	blue	3	magenta	7
	red	4	orange	8
4	black	0	black	0
	green	1	buff	5

In each of the modes one of the colour sets is likely to be of more use than the other, but of course this depends on your application. Notice that in the two-colour modes (0, 2 and 4) the black and buff (white) colour set is likely to be more useful than black and green. In

the four-colour modes the choice is not so clear cut in that the first colour set – green,yellow,blue and red – contains bold clear colours but lacks buff (white), but the second colour set – buff,cyan,magenta and orange – contains two colours that are shades of blue, cyan a light blue and magenta a purple. The choice can sometimes be difficult but it usually comes down to whether you need buff or not! You should recognise the colour codes as the same as those used for the colours in low res graphics. When you are drawing in high res you should use the colour codes in the colour set that you have selected, but you can change the colour set used to display the hi res screen using the SCREEN command at any time. The effect of changing the colour set is simply to swap each colour displayed for the one on the same line in the table above. For example, if you were displaying a picture in black and green in mode 0 then the command SCREEN 1,1 would cause all the green areas to change to white almost immediately. This fast way of changing colours can be useful so it is worth keeping in mind.

Once you have selected the colour set that you are going to use there is one other decision to make. One of the colours must be designated as a background colour and another must be designated as a foreground colour. The reason for this is that many of the hi res commands automatically produce points in the currently selected foreground colour. You can think of all high res commands as drawing in the foreground colour against the background colour. The command

COLOR foreground,background

sets the colours to be used for the foreground and background colours in all hi res commands from then on. (Notice that the American spelling of colour is used.) It is important to realise that COLOR changes nothing that is already on the screen – it only sets the colours to be used in future operations.

As we discovered in the last section, just displaying a hi res screen produces whatever rubbish is already stored there. Before we can begin to draw pictures on the screen we obviously have to clear all the rubbish off. This can be done by using

PCLS c

which will clear the screen to the colour specified by the colour code c. If you just use PCLS without a colour code then the screen will be cleared to the currently selected background colour. For example:

```

10 PMODE 1
20 SCREEN 1,0
30 FOR C=1 TO 4
40 PCLS C
50 FOR I=1 TO 1000:NEXT I
60 NEXT C

```

which clears the screen to each of the colours in the first colour set.

After PCLS, perhaps the simplest hi res command for altering what is displayed on the screen is

PSET(x,y,c)

This changes the point at x,y to the colour given by the colour code c. Notice the similarity between SET (see Chapter Seven) and PSET. However, there are no restrictions or strange effects with PSET, you can set any point to any colour without affecting any other point. Try the following program, both to discover how PSET works and to investigate the way x,y co-ordinates work.

```

10 PMODE 3
20 PCLS 1
30 INPUT X,Y,C
40 SCREEN 1,0
50 PSET(X,Y,C)
60 FOR I=1 TO 2000:NEXT I
70 GOTO 30

```

You can enter values for X in the range 0 to 255, Y in the range 0 to 191 and C in the range 1 to 4. The program is trickier than you might imagine. Line 10 selects mode 3 and line 20 clears it to colour 1 (green), but as no SCREEN command has been given all the text screen is still being displayed. Line 30 gets the three input values and then a SCREEN command at line 40 switches the display to the hi res screen, where line 50 plots a plot at X,Y in the colour C. The delay loop at 60 is necessary because the INPUT statement at line 30 automatically switches the display back to the text screen.

For an automatic demonstration of PSET try:

```

10 PMODE 1
20 PCLS 3
30 SCREEN 1,0
40 X=RND(256)-1
50 Y=RND(192)-1
60 C=RND(4)

```



```

70 PSET(X,Y,C)
80 GOTO 40

```

Experiment with this program by trying different resolutions and colour sets.

The final hi res graphics command that we will look at in this section is redundant in the sense that you could quite easily do without it!

PRESET(x,y)

is the hi res equivalent of RESET and changes the colour of the point at x,y to the currently selected background colour. However, PRESET(x,y) produces exactly the same result as PSET(x,y,c) if c is the colour code of the background colour set by the last COLOR command. Although PRESET can always be replaced by an appropriate PSET it is slightly easier to use when setting a lot of points to the background colour, or where you have forgotten what the background colour is!

Drawing lines and boxes - LINE

Changing the colour of single points using PSET or PRESET isn't as useful in hi res graphics as you might think. The reason for this is that the points are so small! Perhaps the most useful hi res command is

LINE (x1,y1)-(x2,y2),a

which produces a straight line between the point at x1,y1 and the point x2,y2. The colour of the points on the line is determined by a. If a is PSET then the current foreground colour is used and if a is PRESET then the current background colour is used. (Recall that the foreground and background colours are set using the COLOR command.) For example:

LINE (0,0)-(100,100), PSET

will draw a line from the point 0,0 to the point 100,100 in the current foreground colour and

LINE (0,0)-(100,100),PRESET

will set all the points on the line to the background colour. To see lines in different colours try the following:

```

10 PMODE 3
20 SCREEN 1,0
30 COLOR 4,1
40 PCLS
50 X1=RND(256)-1
60 Y1=RND(192)-1
70 X2=RND(256)-1
80 Y2=RND(192)-1
90 C=RND(4)
100 COLOR C,1
110 LINE(X1,Y1)-(X2,Y2),PSET
120 GOTO 50

```

which draws lines at random in a random colour! Notice the use of the **COLOR** command at line 90 to set the foreground colour to be used by the **LINE** command at line 110.

As well as drawing shapes and outlines you can have fun by using **LINE** to make patterns as in the following program:

```

10 PMODE 4
20 SCREEN 1,1
30 COLOR 0,5
40 PCLS
50 FOR I=1 TO 191 STEP 4
60 LINE (0,I)-(255-I,0),PSET
70 LINE (I,0)-(255,I),PSET
80 LINE (0,191-I)-(255-I,191),PSET
90 LINE (I,191)-(255,191-I),PSET
100 NEXT I
110 GOTO 110

```

The output from this program is reproduced in Fig. 9.3. You will find that you get different effects by altering the values of the **STEP** in line 50. Notice the way that mode 4 is used with a white background and a black foreground.

Although it may seem out of keeping with its name, the **LINE** command can also be used to draw boxes! If you add the letter **B** to the end of a **LINE** command, then instead of drawing a line between the two points it will turn them into the opposite corners of a box. Thus the command

```
LINE (X1,Y1)-(X2,Y2),PSET,B
```

draws a box in the current foreground colour with the points **X1,Y1** and **X2,Y2** as opposite corners. If you use **PRESET** then the box

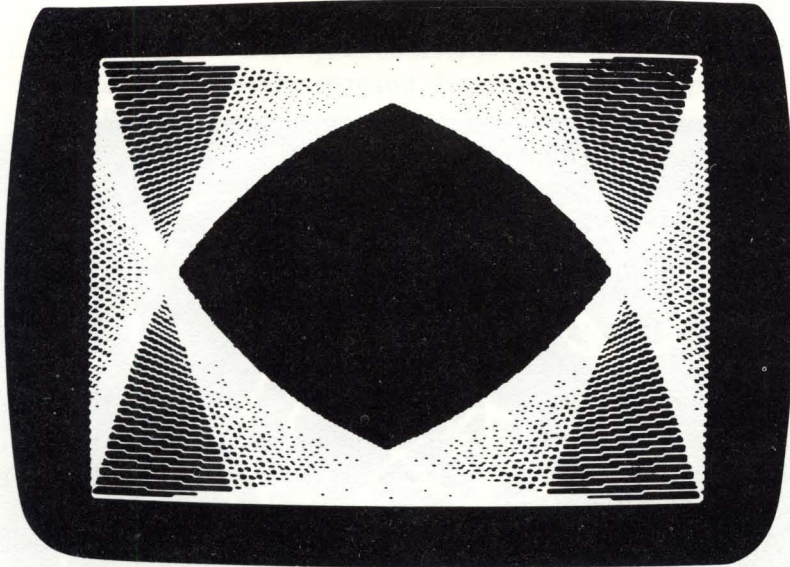


Fig. 9.3. String pattern program.

will be drawn in the current background colour. If you want to see the Dragon draw boxes at random all you have to do is to go back to the random line drawing program given above and add a B to the end of line 110.

As if this wasn't enough, you can also draw filled in boxes using LINE. If you add a BF to the end of the command you not only get the outline of a box – it is also filled in with the current foreground or background colour depending on whether you have used PSET or PRESET. To see a demonstration of this just add BF to the end of line 110 in the random line drawing program!

Drawing circles and ellipses – CIRCLE

The command

`CIRCLE (x,y),r`

will result in a circle centered on the point x,y, and with a radius r being drawn on the screen. For example

`CIRCLE (100,50),40`

draws a circle in the current foreground colour centered at 100,50

and radius 40. As an example of the circle command, the following program draws random circles, as illustrated in Fig. 9.4:

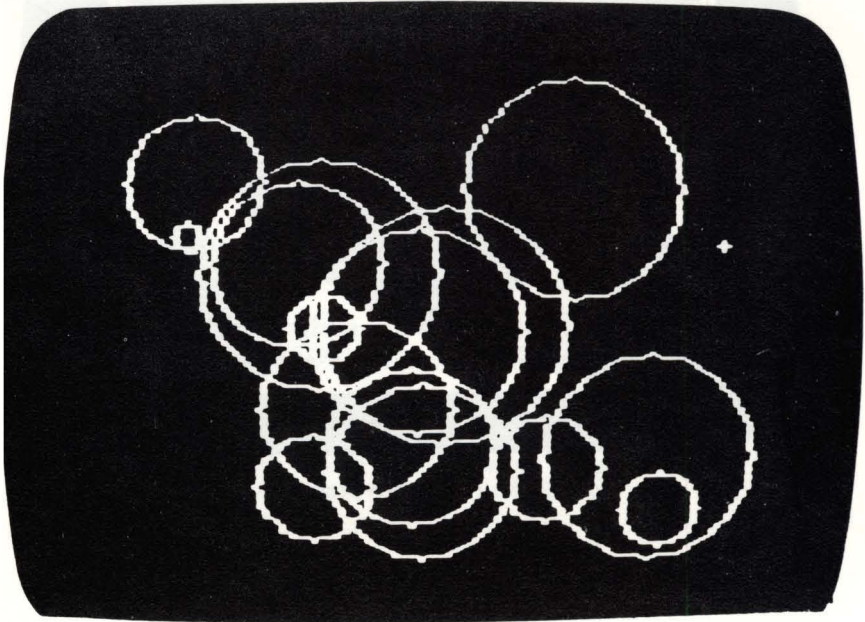


Fig. 9.4. Random circle program.

```

10 PMODE 3
20 SCREEN 1,0
30 COLOR 4,1
40 PCLS
50 R=RND(50)
60 X=R+RND(255-2*R)
70 Y=R+RND(191-2*R)
80 CIRCLE (X,Y),R
90 GOTO 50

```

Notice that when drawing random circles you have to be careful not to go off the edge of the screen. This is allowed for in this program by the inclusion of $-2*r$ in lines 20 and 30. Nothing terrible happens if your circle does go off the edge of the screen. It's just that any points that go past the edge are plotted on the edge and this produces 'flattened' circles!

The circle command as given above is simple, easy to use and remember. However, it is not the complete form of the circle

command. There are other parameters that can be included to gain more control over what is produced. If you just want to produce a simple circle then use the command given above. The complete form of the CIRCLE command is

CIRCLE (x,y),r,c,hw,start,end

As we have already seen, the first three parameters, x,y and r, control the position of the centre of the circle and its radius. All the other parameters are optional. The fourth parameter c is the colour code of the colour that the circle will be drawn in. This can be useful if you just want a single circle in some other colour than the background colour. The fifth parameter can be used to produce 'squashed' circles or ellipses, and the sixth and seventh parameters, start and end, can be used to produce parts of circles.

The way that the hw (height/width ratio) parameter works is easy to understand once you have noticed that you can specify two radii for a shape like a circle. If you imagine a flattened circle, then how flattened it is could be described by giving its horizontal radius and its vertical radius – see Fig.9.5. In the CIRCLE command the r

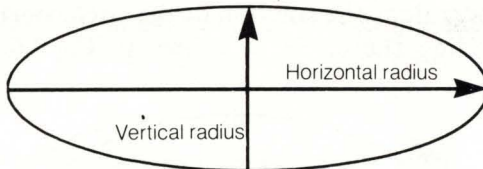


Fig. 9.5.

parameter specifies the horizontal radius or half width of the circle to be drawn, and the hw parameter gives the ratio of the height to the width – hence its name hw ratio. If you set a value of r equal to 50 and hw equal to 2 then the horizontal radius will be 50 and the vertical radius will be $r \cdot hw$ or 100. This would produce a long thin 'vertical' ellipse. However, if you use r equal to 50 and hw equal to .5 the horizontal radius will again be 50 but the vertical radius will be $50 \cdot .5$ or 25. This will produce a thin 'horizontal' ellipse.

To summarise, the horizontal radius is specified by r and the vertical radius is given by $hw \cdot r$. Values of hw greater than 1 produce vertical ellipses and values of hw less than 1 produce horizontal

ellipses. Of course a value of *hw equal* to 1 produces a circle, and this is the value that is assumed if you leave *hw* out of the CIRCLE command. To see some ellipses try the following:

```

10 PMODE 4
20 SCREEN 1,1
30 COLOR 0,5
40 PCLS
50 FOR H=.1 TO 2 STEP .1
60 CIRCLE (100,90),50,,H
70 NEXT H
80 GOTO 80

```

Notice the use of the double comma in line 60 to show that the *c* parameter has been left out.

The final two parameters in the CIRCLE command can be used to draw parts of circles or ellipses. The parameter *start* defines where the circle will start and finish where it will finish! This sounds easy, but how do you define a position within a circle? The Dragon uses the idea of a fraction of a circle – for example .5 of a circle is a semicircle and .25 is a quarter of a circle. The only remaining problem is where does the fraction of the circle start from? This is solved by dividing the circle as shown in Fig.9.6. To specify a

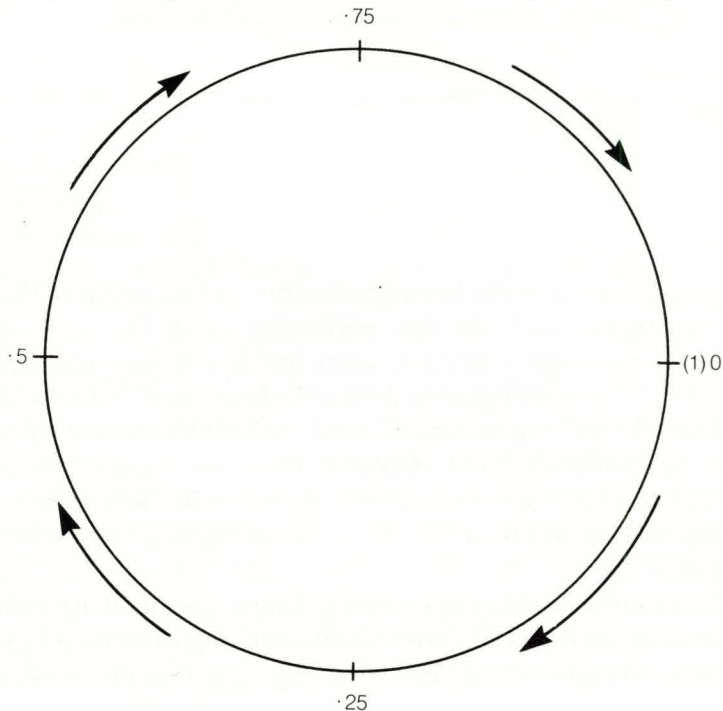


Fig. 9.6.

horizontal semicircle below the diameter you would set start to 0 and end to .5. On the other hand, to specify a horizontal semicircle above the diameter you would have to set start to .5 and end to 0. You can draw any part of a circle or an ellipse by giving start and end the appropriate values as indicated by Fig. 9.6.

As an example of partial circle drawing try the following program.

```

10 PMODE 4
20 SCREEN 1,1
30 COLOR 0,5
40 PCLS
50 R=RND(50)
60 X=R+RND(255-2*R)
70 Y=R+RND(191-2*r)
80 S=RND(0)
90 E=RND(0)
100 CIRCLE (X,Y),R,,1,S,E
110 GOTO 50

```

The complete circle command is quite sophisticated but it has to be admitted that the simple version of the command introduced at the start of this section is the most commonly used and most generally useful!

Finding out what's on the screen - PPOINT

In the same way that POINT can be used to find out what is on the text/low res screen, the function PPOINT can be used to find out what colour is on the screen at any given hi res position. The general form of the PPOINT function is

PPOINT(x,y)

which will return the colour code of the point at x,y. The sort of thing that PPOINT is used for is similar to the use of POINT in the saucer program given in Chapter Eight.

Using hi res graphics

There are so many hi res commands – there are yet more to come in the next chapter – that it can be difficult to see how they can be used to produce effective displays. As always, the best way to learn is to

have a look at some examples and then try to write your own programs.

The first example uses the INKEY\$ function to control the position of a dot on the screen. By pressing the appropriate arrow keys you can draw shapes on the screen:

```

10 PMODE 4
20 SCREEN 1,1
30 COLOR 0,5
40 PCLS
50 X=100
60 Y=80
70 A$=INKEY$
80 IF A$=CHR$(8) THEN X=X-1
90 IF A$=CHR$(9) THEN X=X+1
100 IF A$=CHR$(94) THEN Y=Y-1
110 IF A$=CHR$(10) THEN Y=Y+1
120 PSET (X,Y,0)
130 GOTO 70

```

Notice that the INKEY\$ function has to be used to avoid the automatic switching back to the text screen that would occur with INPUT. You can see a sample of the output of this program in Fig.9.7. You could try to add some improvements of your own, such as diagonal movements and the ability to move from one place to another without drawing a line.

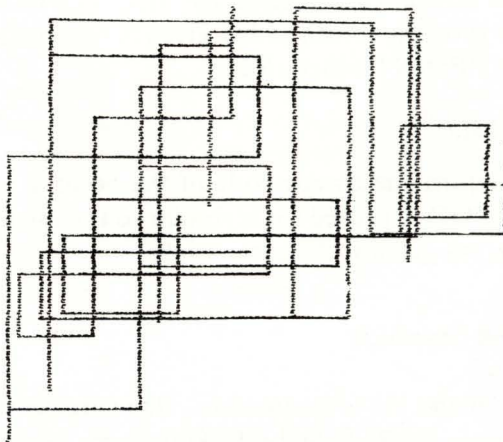


Fig. 9.7. Etch-a-sketch program.

The second example is based on a set of patterns discovered by the nineteenth century French physicist, Lissajous, and aptly called 'Lissajous figures':

```

10 PMODE 4
20 SCREEN 1,1
30 COLOR 0,5
40 PCLS
50 T=0
60 T=T+0.1
70 X=50*(1+SIN(1.1*T))
80 Y=50*(1+COS (T))
90 PSET (X+50,Y+50,0)
100 GOTO 60

```

The output of this program can be seen in Fig. 9.8. You can produce

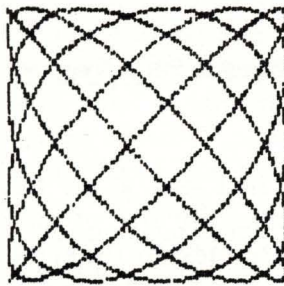


Fig. 9.8. Lissajous figure.

a range of different patterns by changing the value 1.1 in line 70.

The final example in this chapter is a program that plots the graph of $\text{SIN}(x)/x$. This produces a particularly interesting shape, as you can see in Fig. 9.9. One difficulty to be aware of is that $\text{SIN}(x)/x$ is impossible to work out when x is zero, so this point has to be carefully left out of the graph.

```

10 PMODE 4
20 SCREEN 1,1
30 COLOR 0,5
40 PCLS
50 FOR I=1 TO 255
60 IF I=127 THEN GOTO 100
70 X=(I-127)/5

```



```
80 Y=ABS(150*SIN(X)/X)
90 PSET(I,191-Y,0)
100 NEXT I
110 GOTO 110
```

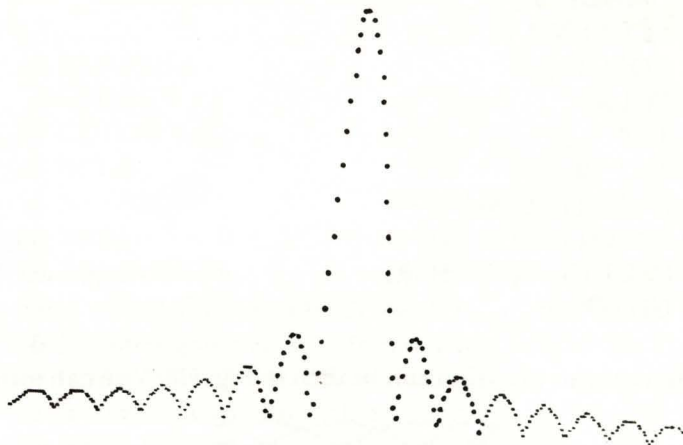


Fig. 9.9. SIN X program.
X

This concludes our first look at the Dragon's hi res graphics. Although hi res graphics are free of the restrictions of low res graphics, you cannot print text on the hi res screen and the number of colours that you can use is limited to a maximum of four. In the next chapter some more hi res graphics commands will be introduced that greatly extend the range of things that you can do.

Chapter Ten

More High Resolution Graphics

Although we have seen a number of high resolution graphics commands in Chapter Nine there are yet more! To begin with, the Dragon has a command, **DRAW**, that can be used to draw irregular shapes on the screen almost as easily as the straight lines and circles of the last chapter. If you want to produce solid shapes on the screen, the **PAINT** command can be used to fill any outline shape with a specified colour. Once you have drawn and painted an object it can be stored in an array using **GET** to be produced anywhere on the screen by a single **PUT** command. Finally, the Dragon has within it a number of internal graphics screens. You can select which of these screens is currently being displayed and which is to be used by the current graphics commands. This means that while displaying one screen you can be preparing another and so avoid showing the user how the screen was produced.

You can produce graphic displays using only the commands described in the last chapter. In this sense the commands described here are 'icing on the cake'! It is tempting to rush on and try to master the Dragon's powerful extended graphics commands, but you will find this more difficult if you haven't finished learning about the simpler commands. Before reading on make sure that you understand Chapter Nine.

The **DRAW** command

The **DRAW** command is very much like the drawing program given in the last chapter. For example, you could draw a square by moving 20 units up the screen 20 right, then 20 down and finally closing the square by moving 20 to the left. The **DRAW** command works in exactly this way. You can imagine that you are controlling the position of a pen that is drawing on the screen and moving

according to the instructions that you give it. The instructions are all of the form 'move x units up' or 'move y units down', but written in an abbreviated form. For example, you can draw a square by

DRAW "U2ØR2ØD2ØL2Ø"

where U2Ø means up 2Ø, R2Ø means right 2Ø, D2Ø means down 2Ø and L2Ø means left 2Ø. The square would be drawn from wherever the imaginary pen found itself at the start of the command. Obviously it is important to be able to start a drawing off from any given point and to this end you can use the

BMx,y;

command which will move the pen to the point x,y without leaving any 'marks'. The letters BM stand for Blank Move, and in fact you can place a letter B in front of any pen movement command to make the pen move without drawing anything. The general form of the DRAW command is

DRAW 'string'

and the 'string' that follows the word DRAW contains any of the permitted draw commands from the following list:

	<i>Pen motion</i>
Mx,y	move pen to x,y
Msx,sy	move pen relative to current position s must be either + or -.
Ud	move pen up d units
Dd	move pen down d units
Ld	move pen left d units
Rd	move pen right d units
Ed	move pen 45 degrees d units
Fd	move pen 135 degrees d units
Gd	move pen 225 degrees d units
Hd	move pen 315 degrees d units

	<i>Drawing conditions</i>
Cn	change colour of pen to n
Ak	change angle of drawing
Sk	change scale of drawing

	<i>Options</i>
N	no update of pen position by next move
B	no mark produced by next move

Special commands

X

execute a substring

The meaning of some of these commands needs further explaining, so don't worry if they don't make sense at the moment. The move commands are straightforward once you know which directions the four angles 45, 135, 225 and 315 degrees define. To see the directions that correspond to the movement commands see Fig. 10.1. You will also notice that there are two versions of the M

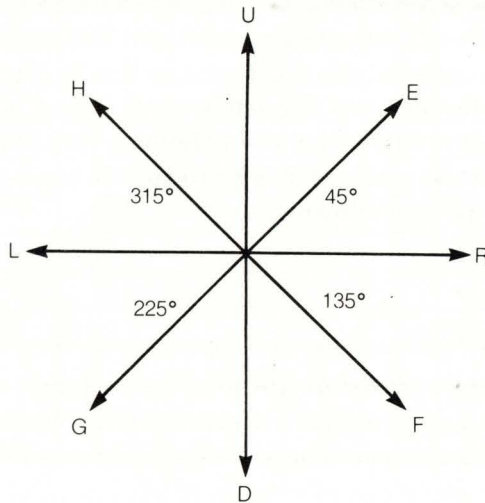


Fig. 10.1. Draw directions.

command. If you specify $M5\phi, 5\phi$ then the pen will move to the point $5\phi, 5\phi$. However, if you specify $M+5\phi, -5\phi$ then the pen will move to a point 5ϕ points to the right and 5ϕ points up from its current position. In other words if the pen is at x, y then $M+5\phi, -5\phi$ will move it to $x+5\phi, y-5\phi$.

Before moving on to look at the other types of command allowed in the draw string it is worth making sure that the move commands are well understood. The following program produces roughly the same results as the drawing program in the previous chapter, but it uses the DRAW command instead.

```
1 $\phi$  PMODE 4
2 $\phi$  SCREEN 1,1
3 $\phi$  COLOR 5, $\phi$ 
4 $\phi$  PCLS
```

```

50 A$=""
60 B$=INKEY$
70 IF B$=CHR$(13) THEN GOTO 100
80 A$=A$+B$
90 GOTO 60
100 DRAW A$
110 GOTO 50

```

The program begins by setting mode 4 (the highest resolution mode) and clearing the screen to black. Then the `INKEY$` function is used to input characters from the keyboard and add them to the string `A$`. If 'return' is pressed this is detected by line 70 ('return' is `CHR$(13)`) and then the string of characters in `A$` is used as a draw string in line 100. So any valid draw string that you type in will be carried out by line 100 and you will be able to see the result on the screen. For example, try the following:

```

L20 'return'
E20 'return'
D20 'return'

```

These three draw commands will produce a triangle on the screen. Notice that the 'pen' doesn't move between `DRAW` commands. You could 'lump' the three commands together in one `DRAW` string:

```

L20E20D20 'return'

```

and produce exactly the same effect. Notice that `E20` moves the pen 20 units diagonally. This is equivalent to moving 20 units horizontally and 20 units vertically – see Fig. 10.2. You will soon get the idea of using `U`, `D`, `L`, `R`, `E`, `F`, `G` and `H` as direction commands to move the pen around the screen. Check that you also understand the `M` command to move the pen to a particular point. For example, try:

```

M50,50;M100,50

```

Notice the use of the semicolon to separate the two commands. Semicolons can be used anywhere in a draw string to make it more readable. If you want to move to a point on the screen without leaving a mark then all you have to do is to write `B` in front of each command that you use to move the pen to the desired position. If you want to leave the pen at a particular position but draw a line to some other point you can use the `N` command. Any command that

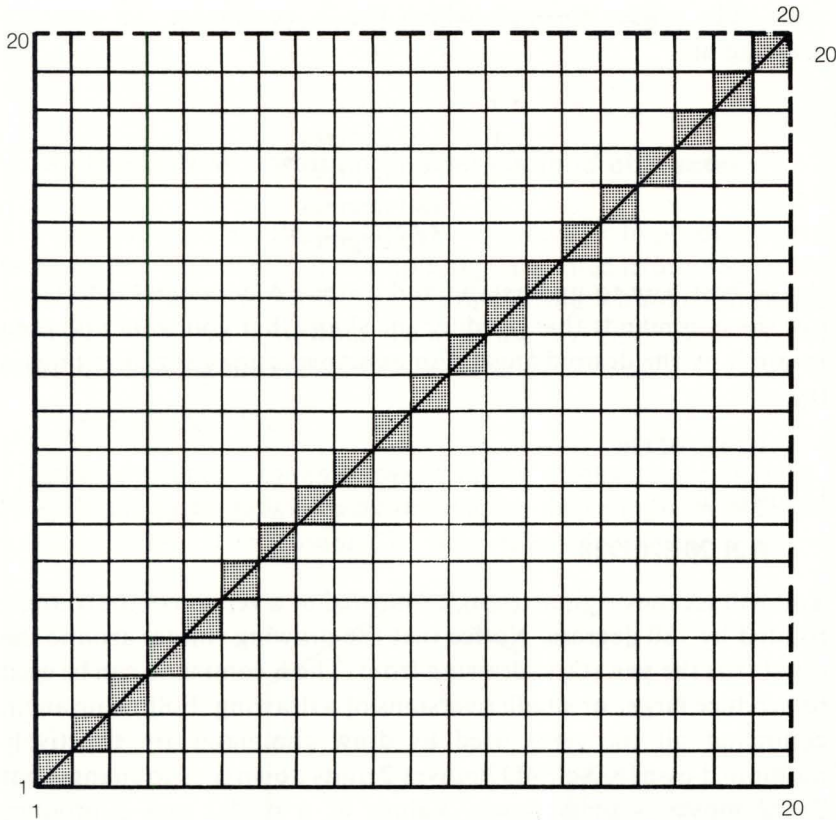


Fig. 10.2.

starts with N will carry out its draw operation and then return the pen to where it was. For example if you try

```
NU20NR20ND20NL20
```

instead of getting a square you will produce a cross. The reason for this is that the pen starts from a particular point, then moves up 20, but instead of then moving 20 right it returns to the original position and then moves 20 right and so on.

The drawing condition commands are fairly easy to understand. The Cn, or colour command, can be used to change the colour of the line that the pen produces. Following a Cn command the pen will draw in the colour with the code given by n. So following C0 the pen will draw black lines, assuming that black is a colour that can be used in the current mode. The A command can be used to alter the angles of the drawing and the S command can be used to alter its

scale. The angles through which the A command can rotate a drawing are:

k	angle
0	0 degrees
1	90 degrees
2	180 degrees
3	270 degrees

The best way to understand and use the A command is to work out the commands that produce the shape that you want and then rotate it by the desired angle. For example, using the triangle again try

```
L2ØE2ØD2Ø
```

followed by

```
A2L2ØE2ØD2Ø
```

You will see the original triangle followed by a version of the triangle rotated by 180 degrees. Notice that the drawing rotates around the point that the pen starts drawing from. The S command can be used to produce larger or smaller versions of a drawing. Following an Sn command all the steps used by draw command are effectively multiplied by n/4. So S4D2 moves 2 units down (i.e. no change) but S8D2 moves 4 units down. Values of n of 1,2 and 3 produce drawings smaller than the step size would suggest; a value of 4 gives a normal sized drawing and larger values of n make the drawing bigger. For example try

```
S8L2ØE2ØD2Ø
```

which produces a triangle twice as big as you would expect.

The final draw command is X, or execute a substring. This command allows you to include a string variable within a draw string. For example you could write

```
C$="S16CØ"  
DRAW "XC$;L2ØE2ØD2Ø"
```

which would draw the triangle four times larger and in black. The contents of the string variable following the X are treated as if they had been written directly into the draw string.

This is all there is to the DRAW command and the commands that can be used to form the draw string. However, using this

powerful Dragon command to produce the effect that you require is another matter.

Using the DRAW command - user-defined characters

The most obvious use for the DRAW command is to produce fairly large outline drawings. For example, if you wanted to draw a picture of a house then you could use a draw string without too much difficulty. Try the following draw string by entering it to the previous drawing program:

`U3ØE2ØF2ØD3ØL3Ø`

This seems a very reasonable draw string for a house shape – see Fig. 10.3 – but in fact it fails because the last line doesn't join up with the starting point. The trouble is that there has been a slight

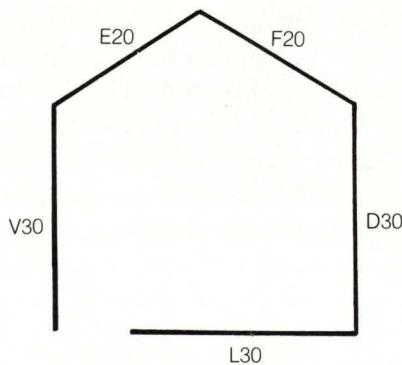


Fig. 10.3. House shape.

miscalculation in the width of the shape! If you recall that a diagonal movement like E2Ø moves 2Ø units horizontally and 2Ø units vertically, you should be able to work out that the correct draw string is:

`U3ØE2ØF2ØD3ØL4Ø`

That is, the last move is 4Ø units to the left. In this case it was easy to work out the move needed to close the outline, but in general after many moves it can be difficult to find the correct move to close the shape. The solution is to know the co-ordinates of the starting point and use the M command to make the final move back to it! For example try:

`BM1ØØ,1ØØ;U3ØE2ØF2ØD3ØM1ØØ,1ØØ`

Once you have a draw string that produces the shape that you want you can make the shape appear anywhere on the screen, in any colour and at any scale and angle that you desire using a very simple technique. For example, to draw two house shapes at different places use:

```

10 PMODE 4
20 SCREEN 1,1
30 COLOR 5,0
40 PCLS 0
50 A$="U30E20F20D30L40"
60 DRAW "BM100,100;XAS;"
70 DRAW "BM150,150;XAS;"
80 GOTO 80

```

The basic idea is straightforward. First define a string variable which holds all of the draw commands that you do not want to change. Then use a DRAW command to position the pen, change the colour, angle or scale and then execute the string variable to produce the shape. You can even include variable data into the DRAW command by using the STR\$ function to convert numbers to strings. For example, if you want to draw a ship outline at various positions across the screen then try:

```

10 PMODE 4
20 SCREEN 1,1
30 COLOR 5,0
40 PCLS 0
50 A$="R30U5L20U5L5D5L5D5"
60 Y=50
70 FOR X=0 TO 200
80 DRAW "BM"+STR$(X)+","+STR$(Y)+";XAS;"
90 NEXT X

```

You will see a ship shape drawn at a large number of points across the screen. To make it look as though the ship is moving it is necessary to 'blank out' the old version of the ship before drawing the new one. To do this change line 80 to

```
80 DRAW "BM"+STR$(X)+","+STR$(Y)+";C5ZA$;"
```

and add line 85

```
85 DRAW "BM"+STR$(X)+","+STR$(Y)+";C0XA$;"
```


Line 80 draws the ship in white and line 85 'undraws' it by drawing it in black, the background colour.

In general if you want to draw a shape defined by the string `AS` at `X,Y` in colour `C` then use

```
DRAW "BM"+STR$(X)+"," +STR$(Y)+";C"+STR$(C)+
"XAS;"
```

You can add variable angle and scale commands in exactly the same way.

Once you have a string that correctly defines a shape you can use it to produce the shape in any of the graphics modes. For example, the string that draws the house shape can be used in mode 0 to produce a house shape at lower resolution. However, there is one problem in using the `DRAW` command at resolutions lower than those provided by mode 4. All pen movements are in terms of the standard co-ordinate grid used in commands such as `PSET` and `LINE`, that is 0-255 horizontal by 0-191 vertical. This is fine in mode 4 because there are as many points displayed on the screen as there are co-ordinate points, but in the other modes this is not so. For example, in modes 2 and 3 there are only 128 points horizontally, although the co-ordinate grid still numbers them 0-255, as if there were twice as many. So in modes 2 and 3 a move of one unit may or may not show up, depending on where you are on the screen. To see this load the drawing program given earlier in this chapter and change lines 10-30 to

```
10 PMODE 3
20 SCREEN 1,0
30 COLOR 3,1
```

and try entering `L1` repeatedly. You will only see any horizontal movement of the line for every other `L1` command. This is what you would expect, in that although each `L1` command decreases the x co-ordinate by one you only move the pen onto a new screen point every time the x co-ordinate changes by *two*. If you try the same experiment with the vertical direction by entering `U1` repeatedly you will see no such problem; each `U1` command produces a new plotted point. If you repeat the same experiment in modes 0 or 1 you will find that the repeated use of `L1` or `U1` only produces a visible change on the screen every other time. This is because in these modes both the horizontal and vertical directions have only half the number of screen points as the co-ordinate grid would suggest!

What this means in practice is:

- position of the pen by at least two each time.

the shape is positioned.

disappointed at the result.

draw string for this might be:

D2L3R7L2D5U3L3D3U4R2U3

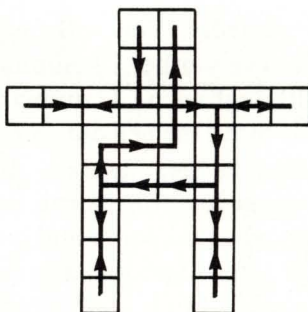


Fig. 10.4. Little man and its draw moves.

(Remember, there is more than one way of drawing a man!) If you input this string to the drawing program set up for mode 4 you will

see a very acceptable solid 'little man' figure. This figure will look the same no matter where you draw it on the screen, and all in all it is very serviceable. However, if you use the same string in the program set up for mode 0 you will be disappointed. The man-shaped figure works quite well if the starting point has even x and y co-ordinates, but if you start from odd x and y co-ordinates all you will see is a cross. Try:

```

10 PMODE 0
20 SCREEN 1,1
30 PCLS
40 AS="D2L3R7L2D5U3L3D3U4R2U3"
50 DRAW "BM51,51;XAS;"
60 DRAW "BM100,100;XAS;"
70 GOTO 70

```

The 'little man' produced by line 60 does indeed look like a little man, but the 'little man' produced by line 50 is very wrong! Imagine the effect of moving the man across the screen – it would look awful! Even large shapes that have fine detail can look very unconvincing as they move across the screen, because of the way the fine detail changes as the object moves from odd to even positions.

The solution to this problem is to multiply all of the movements by two. This can be achieved by doubling the scale, i.e. setting it to 8 by changing line 50 to:

```

50 DRAW "S8BM51,51;XAS;"

```

Now the man will look the same at both positions. In modes 2 and 3 the problem is more difficult to solve because movements in the vertical direction are fine but the horizontal direction shows the same variation in shape according to position as seen in modes 0 and 1. If you simply change the scale to 8 (i.e. double in both x and y) you will be wasting some of the vertical resolution. The correct thing to do is to multiply all the horizontal movements by two and leave the vertical moves alone. This sounds easy until you realise what this means for diagonal moves! Diagonal moves move the pen the same amount vertically as horizontally, so how can you multiply just the horizontal distance by two? You should be convinced by now that drawing small figures in modes 2 and 3 is not easy!

To repeat:

To draw small shapes in modes 4, 1 and 0 design the shape in mode



0.5 The letter 'a' on

lining them up m

THE CONCLUSION

where x,y must be a point inside the outline (it can be any point but it must be inside the shape!). What happens is that the Dragon will start to fill the screen with the colour c from the point x,y and only stop when it reaches a boundary of colour b. Try the following short program:

```
10 PMODE 3
20 SCREEN 1,0
30 PCLS
40 DRAW "C4BM50,50;R80D80L80U80"
50 PAINT (60,60),3,4
60 GOTO 60
```

The DRAW statement produces the outline of a square in colour 4. The PAINT command then fills the outline with colour 3. It is important to make sure that the starting point is inside the outline. To see the effect of getting this wrong change line 50 to:

```
50 PAINT (5,5),3,4
```

Slightly less obvious is the effect of a small 'hole' in the outline. For example try changing line 40 to

```
40 DRAW "C4BM50,50;R80D40BD2D38L80U80"
```

This leaves a small, almost invisible gap in the outline and as you would expect the paint 'leaks' out!

The PAINT command is very useful for filling large areas of the screen with colour, but it is very slow. This means that if you want to animate the shape don't use the PAINT command!

GET and PUT

The DRAW command can be used to produce outline shapes and small solid objects anywhere on the screen. However, the Dragon does have two extra commands that can be used to 'read' the contents of an area of the screen into an array and then 'redraw' the contents at some other place. In other words you can produce a small drawing using a combination of LINE, CIRCLE, DRAW or PAINT and then read it into an array and then produce it any time you need it using one comma.

The command that reads an area of the screen into an array is:

```
GET(x1,y1)-(x2,y2),array-name,G
```

The co-ordinates $x1, y1$ and $x2, y2$ define a rectangle in the same way that the pairs of co-ordinates define a rectangle in the LINE command (see Chapter Nine). The array must be large enough to hold all of the graphics points in the rectangle. There is a complicated way of working out the smallest array needed to store the graphics points in the rectangle, but it doesn't matter if you use an array that is too big, apart from the waste of storage. It is easy to define an array that is always large enough using:

```
DIM (x2-x1,y2-y1)
```

In other words, using a two-dimensional array with as many elements as there are points in the rectangle. The final parameter G is optional and governs the ways in which you can re-draw the shape that you have stored in the array.

To re-draw the shape after GETting it into an array you have to use the PUT command:

```
PUT (x1,y1)-(x2,y2), array-name, action
```

The co-ordinates once again define a rectangle, but this time the information in the array is transferred to the rectangle as graphics points. If you didn't use the G option in the GET command then you *must* leave the 'action' term out of the command. If you did use the G option then you can use one of the following five 'action' parameters:

PSET –	Set points to the same colour that they were in in the original drawing
PRESET –	Resets each point that is set in the original, i.e. background colours become foreground and <i>vice versa</i>
AND –	Sets a point if the point is already set and was set in the original
OR –	Sets a point if either the point is already set or was set in the original
NOT –	Sets a point if it was unset and <i>vice versa</i> , irrespective of the information stored in the array

It is important to notice the following:

- (1) If you do not use G then do not use 'action'.
- (2) GET and PUT without G or 'action' will simply save and copy the original version of the drawing without modification.

- (3) The rectangle used in a PUT must be the same size as the rectangle used in the GET.
- (4) You shouldn't change modes between GET and PUT!

If you want to see the way that GET and PUT work try the following:

```

10 PMODE 3
20 SCREEN 1,1
30 PCLS
40 DIM A(25,25)
50 DRAW "BM0,0;C4R20D20L20U20"
60 PAINT (2,2),2,4
70 GET (0,0)-(25,25),A
80 PUT (50,50)-(75,75),A
90 PUT (100,100)-(125,125),A
100 GOTO 100
    
```

To see the effect of the 'action' commands add a G to the end of line 70 and then add action commands to the end of 80 and 90.

GET and PUT sound as though they are going to be very useful, but in practice GET and PUT are generally slow, and for small shapes it is often quicker and simpler to re-draw the shape from scratch!

Graphics pages — PCLEAR and PCOPY

The Dragon has a total of eight internal graphics 'pages' that can be used to create graphics screens. The number of pages that are needed to create a single screen depends on the graphics mode in use:

<i>Mode</i>	<i>Pages</i>
0	1
1	2
2	2
3	4
4	4

As you can see, even the highest resolution graphics mode only uses four of the possible eight pages reserved for graphics. As each page takes a considerable amount of memory (about 1.5k per graphics

page) it makes sense to free all the pages that you are not going to use for graphics and allow BASIC to use the space for programs instead. You can do this using the PCLEAR command:

PCLEAR n

will reserve n pages for graphics use and release the rest (8-n) for BASIC to use. It is important to use PCLEAR only at the very beginning of a BASIC program to avoid BASIC trying to make use of extra memory after settling into the amount of memory that it thought it had. If you don't use a PCLEAR command at the start of your program then BASIC assumes that you need four pages for graphics. So if you are using mode 0 and you want to gain three extra pages for BASIC to use, start your program with PCLEAR 1. It is important to allow the number of pages that each mode requires or you may find that graphics commands are modifying your BASIC program.

Apart from being able to save memory space the PCLEAR command can be used to reserve more graphics pages than a graphics mode needs. For example, mode 0 only needs one page but you could start off a program with PCLEAR 8 and reserve eight pages for it. What can you do with the extra seven pages? Well the answer is that you can draw on any of the eight pages and also independently display any of the eight pages! To set which page you are drawing on you have to use a slightly extended form of the PMODE command:

PMODE mode, startpage

The value of 'mode' determines the graphics mode in use as before, the value of 'startpage' determines which graphics page in memory any subsequent graphics commands will affect. The pages are numbered 1 to 8 so PMODE 0,1 will cause any high resolution commands that follow to draw on graphics page 1 – and as 1 is the default value for 'startpage' everything is as normal. However, PMODE 0,2 will draw on the second page and this is something new. There is one extra piece of information that we need to see what is drawn on any of the graphics pages. As explained in Chapter Nine, carrying out a SCREEN command is the action that causes a screen to be displayed and this is still true when using different graphics pages. You can change graphics pages using PMODE, but you will carry on looking at the old page until you carry out a SCREEN command which displays the page used in the last PMODE. To see this in action try the following:

```

10 PCLEAR 8
20 FOR I=1 TO 8
30 PMODE 0,I
40 PCLS
50 CIRCLE (100,80),I*10
60 NEXT I
70 FOR I=1 TO 8
80 PMODE 0,I
90 SCREEN 1,1
100 FOR J=1 TO 100:NEXT J
110 NEXT I
120 GOTO 70

```

Line 10 reserves eight pages for graphics use. Lines 20 to 60 draws a circle on each graphics page, but without displaying anything other than the original text screen. Line 30 selects the page that the drawing commands affect and then line 40 clears it and line 50 draws the circle. The radius of the circle increases as the page number increases. The second half of the program merely displays each page in turn by using a PMODE command in line 80 followed by a SCREEN command in line 90. The FOR loop in line 100 is there simply to slow things down so that you can see what is happening. Notice that each circle is displayed complete. You do not see it being drawn because that happens only once at the start of the program when the text screen is being displayed.

The Dragon also has a command that allows you to copy the entire contents of one graphics page to another:

PCOPY n TO m

will copy everything in page n to page m. This is useful if you want to produce a number of displays all based on an original. You could set up the original in one page and then copy it to several others, where the changes could be made without spoiling the original.

So far we have only looked at using multiple pages with mode 0. This is the easiest mode in which to use multiple pages, because a screen in mode 0 uses exactly one page. However, using multiple pages in other modes is easy as long as you are clear about the page number and which pages are used to form a screen. For example, in modes 1 and 2 two pages are needed to store a screen of graphics. This means that the pages are used two at a time to form a maximum of four screens:

screen one = page 1 and 2
 screen two = page 3 and 4
 screen three = page 5 and 6
 screen four = page 7 and 8

Each screen is referred to by its lowest page number. So PMODE 1,1 will draw on the first screen and PMODE 1,5 will draw on the third screen in mode 1 graphics. The only complication is in the use of the PCOPY command. To copy an entire screen now requires two PCOPYs. For example, to copy all of screen two to screen four you have to use PCOPY 3 to 7 and PCOPY 4 TO 8.

In the case of modes 3 and 4 the problem is similar, only now there are only two screens, each consisting of four graphics pages:

screen one = page 1,2,3 and 4
 screen two = page 5,6,7 and 8

Once again the screen is selected by referring to the lowest page that it uses and a complete copy between screen one and two would be PCOPY 1 TO 5, PCOPY 2 TO 6, PCOPY 3 TO 7 and PCOPY 4 TO 8.

The way that graphics pages have been organised into screens given above is not unique, in that you can consider a screen to start at any page number and go on to use as many pages as it needs, but this is seldom needed in a program and the organisation given above is easy to use and remember.

There are two main uses for multiple pages in high resolution graphics. The first is simply to avoid having to show a graphics display while it is being built up by showing a different screen and then switching to the new one only when it is finished. The second use is to animate a fixed scene by drawing a number of versions, one per screen, and then displaying the screens in turn. There are probably many other applications for multiple page graphics and this is one of the most exciting and underused Dragon facilities

The saucer game again!

To finish off this chapter on high resolution graphics a new version of the saucer game is given. The purpose of this is to demonstrate how high resolution graphics compare with low resolution graphics.

```

10 T=100
20 F=0
30 FX=0
40 FY=180
50 GOSUB 1000
60 FOR X=2 TO 250 STEP 4
70 GOSUB 2000
80 GOSUB 3000
90 DRAW "BM"+STR$(X-4)+",180;C3S8XR$;BM"+
  STR$(X)+",180;C2S8XR$;"
100 NEXT X
110 FOR X=250 TO 2 STEP -4
120 GOSUB 2000
130 GOSUB 3000
140 DRAW "BM"+STR$(X+4)+",180;C3S8XR$;BM"+
  STR$(X)+",180;C2S8XR$;"
150 NEXT X
160 GOTO 60

1000 PMODE 1
1010 PCLS 3
1020 S$="R12L9R6LIU1L4UIR4LIU1L2RIU1"
1030 R$="R4LIU1L2UIR2UI1L2RIU1"
1040 SCREEN 1,0
1050 RETURN

2000 F$=INKEY$
2010 IF F$="" THEN RETURN
2020 SOUND 10,1
2030 PSET (FX, FY, 3)
2040 FX=X+2
2050 FY=170
2060 F=1
2070 RETURN

3000 DRAW "BM"+STR$(T)+",20;C3S8XS$;"
3010 T=T+2*SGN(RND(0)*2-1)
3020 IF T<0 THEN T=T+2
3030 IF T>240 THEN T=T-2
3040 DRAW "BM"+STR$(T)+",20;C2S8XS$;"
3050 IF F=0 THEN RETURN
3060 PSET (FX,FY,3):FY=FY-4

```

```

3070 PSET (FX, FY,2)
3080 SOUND 50-FY/8,1
3090 IF FY<10 THEN F=0:PSET (FX,FY,3):RETURN
3100 IF PPOINT(FX,FY-2)=3 THEN RETURN
3110 GOSUB 4000
3120 RETURN

4000 FOR I=1 TO 5
4010 PSET (RND(4)*2+T,22-RND(4)*2,RND(4))
4020 SOUND RND(200),1
4030 NEXT I
4040 F=0
4050 PSET (FX,FY,3)
4060 RETURN

```

The program structure is the same as that given in Chapter Eight and you should have no trouble in following the program. See if you can identify the way that the high resolution graphics commands are used. If you compare the two programs you will see that high resolution graphics is considerably slower than low resolution graphics – the choice is yours, speed or detail!

Chapter Eleven

Logic and Other Topics

In this final chapter a number of different topics are introduced. It would be misleading to think of this as a collection of advanced topics just because they have been left until last. Using the BASIC and other information dealt with in earlier chapters you should find it possible to write any program that you want to. However, there are facilities on the Dragon that, although not entirely necessary do make things easier. This chapter collects together these extras and explains how they work. The commands covered are AND, OR, NOT, PEEK, POKE, USR, CLEAR, MEM, PRINT USING, INPUT LINE and REM

Logic and the conditional expression

In everyday speech we often say things like, 'did you buy apples and oranges?', 'do you prefer tea or coffee?' The use of words like 'and' and 'or' are so common that we rarely stop to think about them. It would be a great advantage if the use of 'and' and 'or' could be extended to BASIC conditional expressions, and indeed in Dragon BASIC you can write expressions such as:

$A < 0$ AND $B = 3$

$A < 0$ OR $B = 3$

The meaning of each of these expression is in line with the usual English meaning of 'and' and 'or'. The first expression evaluates to true if both of the conditions ' $A < 0$ ' ' $B = 3$ ' are true, and the second expression is true if either of the two conditions is true. As well as AND and OR the Dragon also allows the use of NOT, which simply changes the value of a conditional expression from true to false and *vice versa*. For example, $3 = 2$ is false but NOT $3 = 2$ is true.

You can combine AND, OR and NOT with conditions to make

more complicated expressions that evaluate to one of the values 'true' or 'false' (if you recall 'true' is represented by -1 and false by $\emptyset$). Conditional expressions that include AND, OR or NOT are usually called *logical expressions* and we can now re-write the definition of the IF statement as:

IF 'logical expression' THEN 'list of BASIC statements'

or

IF 'logical expression' THEN 'list 1' ELSE 'list 2'

For example, if you want to check that you're not using an x co-ordinate that goes outside the screen area you could use

IF $x < \emptyset$ OR $x < 255$ THEN ...

in place of the two IF statements that would be required without the use of OR.

Forming logical expressions to test for 'overall' conditions is usually straightforward. However, there are a few traps that even experts fall into. If you want to translate the English statement 'a equals b and c' then you must repeat the condition $=$. In other words you must use

$A=B$ AND $A=C$

and not use

$A=B$ AND C

which will give a result that depends on whether C is $\emptyset$ or -1 . You should also be careful when using NOT. For example:

NOT ($A=B$ AND $A=C$)

isn't the same as

NOT($A=B$) AND NOT ($A=C$)

To see that this is the case try working the two expressions out for a few values of A, B and C. The moral is that you should always beware of using logical expressions without thinking about *exactly* what they mean.

To close the subject of logical expressions it is worth introducing the idea of a truth table. Consider the logical expression

A AND B

where A and B are variables that are either $\emptyset$ for false or -1 for true.

You can draw up a table that lists the results of the expression for all possible values of A and B as follows:

A	B	A AND B
$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	-1	$\emptyset$
-1	$\emptyset$	$\emptyset$
-1	-1	-1

Such a table is called a *truth table* because it lists the conditions under which the logical expression is true or false. You can draw up truth tables for any logical expression, and this is one way to check that you understand what is happening. For example OR and NOT have the following truth tables:

A	B	A OR B	NOT A
$\emptyset$	$\emptyset$	$\emptyset$	-1
$\emptyset$	-1	-1	-1
-1	$\emptyset$	-1	$\emptyset$
-1	-1	-1	$\emptyset$

The OR that we have used so far is not entirely equivalent to the English word 'or'. Most uses of the English 'or' mean 'one or the other but not both'. For example 'You can have jam or marmalade' means that you can pick one but not (normally) both! However, the logical OR means that you can have either one or both (look at the truth table if you are unsure of this). The logical OR is more properly called the *inclusive OR* because it includes the possibility of both. The usual English 'or' is known as the *exclusive OR* because it excludes the possibility of both. You can make up a logical expression that is equivalent to the exclusive OR:

$$\text{exclusive OR} = (\text{NOT}(A) \text{ AND } B) \text{ OR } (A \text{ AND NOT } (B))$$

as can be seen from the truth table:

A	B	(NOT(A) AND B) OR (A AND NOT(B))
$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	-1	-1
-1	$\emptyset$	-1
-1	-1	$\emptyset$

Inside the Dragon – PEEK,POKE,CLEAR and USR

It may come as something of a surprise to learn that BASIC includes a number of commands that allow access to the inner workings of the Dragon. The reason why this might come as a shock is that all the BASIC that we have looked at so far has done its best to avoid getting involved with details of how the machine carries out commands. However, there are some applications where the normal instructions of BASIC are in some way deficient. For example they might be too slow or fail to take account of some important feature of the machine. To allow the programmer to find a way around such difficulties most versions of BASIC include instructions that allow you to gain access to the inside of the machine.

The first problem with trying to get inside the machine is that although it looks as if the Dragon does all its arithmetic in the familiar decimal notation, it in fact does it in *binary*. Humans count in decimal simply because we have ten fingers. If we only had two fingers then we would count in the same way that computers do! Rather than get too involved in the details of binary arithmetic all that you need to know at this stage is that a binary number is a collection of zeros and ones and to every binary number there is a decimal number and *vice versa*. It is true that there is a lot more to learn than this, but the details are beyond the scope of this book.

We learned very early on that a variable is a named area of computer memory. However, it is sometimes necessary to side step the 'variable' method of using computer memory in preference for direct access using PEEK and POKE. PEEK is a function that will return the contents of a memory location and POKE is a command that will alter the contents of a memory location. It is as simple as that except that you need to know how to specify which memory location and what sort of number can be stored in a memory location.

The first problem is easily solved because the Dragon, like all computers, numbers all its memory locations sequentially starting from zero. So PEEK(543) will return the contents of the five hundred and forty fourth memory location. The second problem is also easily solved once you know that a memory location can store a binary number with up to eight zeros or ones in it and binary 11111111 is the same as the decimal number 255, which is the largest number that can be held in a single memory location. So POKE 1000,200 will store 200 in memory location 1000. However, POKE

1000,600 will give an error message because 600 is greater than 255. In general to use PEEK and POKE you have to have a knowledge of what is stored where inside your Dragon and this is often not easy to find out. PEEKing and POKEing are best avoided unless you are absolutely sure you know what you are doing.

However, there are one or two standard applications of PEEK and POKE that are worth knowing about and also demonstrate the typical way that PEEK and POKE are used.

The Dragon uses the memory from 1024 to 1535 to store the codes of the characters to be displayed on the text screen. Each location holds the code of a single character. To see this try:

```
10 FOR I=1024 TO 1535
20 POKE I,ASC("A")
30 NEXT I
```

You will see letter "A"s appearing on the screen one after the other, as the code for A (i.e. ASC("A")) is POKEd into each memory location. In the same way as character codes can be POKEd to the screen so you can find the character code of what is displayed on the screen by PEEKing the correct memory location. You can often use PEEK and POKE to simplify and speed up graphics, but this is another story.

There is another Dragon function that we have not yet looked at, USR. This is not surprising, as the USR function transfers control out of BASIC and into a *machine code* program stored somewhere inside the Dragon. The command CLEAR can be used to reserve some memory for storing such machine code programs so that the Dragon doesn't allocate the area to variable storage. Unless you want to get involved in another computer language altogether the USR function will be of little interest to you.

One function concerned with the internal workings of the Dragon that is of use to every BASIC programmer is MEM. This simple function will return the amount of memory that is left unused. So to find out how much memory is still available to your program you could use

```
PRINT MEM
```

More INPUT and PRINT

There is a very simple problem that with the BASIC we have

looked at so far is very difficult to solve. The problem regards reading in a string that includes commas. If you try this using

```
1Ø INPUT A$
```

you will find that A\$ only stores the string up to the first comma – the rest is lost! The reason for this is that the comma is used as a separator between different strings on input and if you type a comma in the middle of a string the Dragon assumes that you have finished. To get round this problem you have to use another version of the INPUT statement:

```
LINE INPUT "prompt";string variable
```

For example, try

```
1Ø LINE INPUT "TYPE ANY TEXT INCLUDING  
   COMMAS";A$  
2Ø PRINT A$  
3Ø GOTO 2Ø
```

and you will find that nothing is lost after a comma. Notice that you can only input to one variable using LINE INPUT. You might like to ask yourself why this is?

There is also a problem with printing information on the screen that so far is difficult to solve. How do you print a number on the screen to a specific format, so many digits before the decimal point and so many after? The solution to this problem is a modified version of the PRINT command:

```
PRINT USING string;variable list
```

The 'variable list' is the same as the 'variable list' used in a standard PRINT or PRINT @ statement. The 'string' can either be a string variable or a string constant and is best thought of as a 'picture' of the way that you want the number to look. For example, if you wanted to print all numbers to two decimal places then you would use "#." as the string following USING. The "" can be thought of as marking the place that a digit can be printed and in this sense "#.#" is a picture of the way you want the number printed. There are a large number of formatting commands that can be used in the format string and details of these can be found in the *Dragon Manual*. You might like to try the following program to see their effect:

```
1Ø INPUT "FORMAT =";A$  
2Ø FOR I=1 TO 1Ø
```



```

30 PRINT USING A$;RND(0);RND(0)*(-RND(10))
40 NEXT I
50 GOTO 10

```

After entering the format in response to line 10 the program will print ten lines of random numbers using the format specified. Some formats to try are:

```

##.#
####.#
**###.#
**$###.#

```

In general PRINT USING is necessary if you are planning to print neat tables of numbers.

REMark and good programming

The BASIC command REM is the simplest of all in that it does absolutely nothing! Its only purpose is to allow you to include comments that are not part of a program. For example, you could include in every program that you write a first line that tells you what the program is called:

```
10 REM this is my program
```

and the REM would alert the Dragon to the fact that what followed wasn't to be taken as a line of BASIC but as a note to any humans that might read the program.

You might think it strange that such a simple command is left to the last chapter of a book on BASIC. The reason for this is that although REM is a simple command it can be used in some very sophisticated ways. After you have got over the initial difficulty of writing programs in BASIC you should look toward trying to write better programs. At first a program that works is a reward in itself, but later on a well written program is what you should aim for. What constitutes a well written program is something you will discover for yourself as you learn programming by trial and error, and by reading other people's efforts.

The REM statement is part of better programming in that while it certainly isn't necessary it does help to make your programs easier to understand if you include REMs that explain what is happening in each section of your program. Good explanations will help you to

understand your own programs more quickly when you return to them at a later date and will assist other people, to whom you may give them, to grasp what you intend each stage to do.

Where next?

As I have already said many times in this book, the real route to learning programming is to write programs. Certainly, books can help you but only if you are prepared to experiment on your own behalf. Don't be worried if your first programs don't attempt anything very ambitious. It is better to try out your ideas in short and simple routines at first. If you try anything too complicated there is much chance that you'll make mistakes that you can't locate. Try writing program snippets that do just a few things at a time – if you look back through this book you'll find lots of such examples. When all your mini-programs work then it is time to start putting them together to build more extensive ones. The main thing is to go ahead and to have some fun with your Dragon.

Index

ABS, 70
AND, 151
Array, 60
Array element, 61
Arrow keys, 76
ASC, 73
ASCII code, 73
AUDIO ON/OFF, 109

Binary, 154
Boxes, 122
BREAK, 10
Byte, 3

Cassette recorder, 13
CHR\$, 73
CIRCLE, 123
CLEAR, 10, 55, 154
CLOSE, 66
CLS, 91
Colon, 49
COLOR, 118
Comma, 84
Concatenation, 56
Conditional expression, 39
Control key, 16
COS, 72
CPU, 2

DATA, 64
DEF FN, 78
DEL, 18
DRAW, 131
Dummy parameter, 79

Editing, 12, 15
ELSE, 51
EOF, 66
EPROM, 3
Evaluation, 24, 25

Execute, 107
EXP, 70, 71
Exponential number, 70

Fire button, 98
FIX, 71
Flow of control, 35, 53
FN, 78
FOR, 45, 48
Functional units, 68
Functions, 68

GET, 143
GOSUB, 80
GOTO, 36, 80
Graphics character, 89
Graphics pages, 145

HEX\$, 73
Hi res modes, 114

IF, 38, 41, 49
IF...THEN, 38, 82
IF...THEN...ELSE, 51, 82
INKEY\$, 75, 76
INPUT, 27, 66
Input/output, 3
INSTR, 59, 73
INT, 71
I/O, 3
Iteration, 37

Joysticks, 97

Key signature, 104
Keyboard, 10
Kilobyte, 3

LEFT\$, 58, 74
LEN, 58, 73

- LET, 22
- LINE, 121
- LINE INPUT, 156
- LIST, 18
- LOG, 71
- Logarithm, 71
- Looping, 35, 44
- Lower case, 142

- MID\$, 56, 73
- MOTOR ON/OFF, 109
- Musical notes, 102

- NEW, 18
- NEXT, 46
- NOT, 154
- Null string, 56

- Octave, 102
- OPEN, 65
- OR, 151

- PAINT, 142
- Parameters, 69, 79
- Pause, 106
- PCLEAR, 145
- PCLS, 118
- PCOPY, 145
- PEEK, 154
- PLAY, 101
- PMODE, 116, 145
- POINT, 97
- POKE, 154
- PPOINT, 127
- PRESET, 118
- PRINT, 23, 30
- PRINT @, 85
- PRINT USING, 156
- Programs, 5
- Prompting, 29
- PSET, 118
- PUT, 143

- Radian, 72
- RAM, 3
- Random numbers, 75
- READ, 64
- Relations, 39, 59
- REM, 158
- RESET, 94

- RESTORE, 65
- RETURN, 80
- RIGHT\$, 58, 73
- RND, 75
- ROM, 3

- SCREEN, 116
- Semi-colon, 84
- SET, 94
- Setting up, 8
- SGN, 71
- SIN, 72
- SOUND, 100
- Sound effects, 108
- SQR, 71
- Stave, 103
- STEP, 47
- STOP, 8, 51
- STR\$, 74
- String, 54
- STRING\$, 74
- String constant, 54
- String expression, 55
- String variable, 54
- Subroutine, 80
- Substring, 56
- Super alphabet, 59

- TAN, 72
- Tape recorder, 13
- Tempo, 106
- THEN, 38, 44
- TIMER, 75, 77
- Trigonometrical functions, 72
- Truth tables, 153
- Two-dimensional array, 62

- User-defined characters, 137
- User-defined function, 68, 78
- USR, 154

- VAL, 74
- Variable, 20
- Volume, 106

- x co-ordinate, 88

- y co-ordinate, 88

- %, 54

**THE ORIC-1
PROGRAMMER**

M. James and S. M. Gee
0 246 12157 2

**ORIC MACHINE CODE
HANDBOOK**

Paul Kaufman
0 246 12150 5

The TI 99/4A

**GET MORE FROM
THE TI99/4A**

Garry Marshall
0 246 12281 1

The VIC 20

**GET MORE
FROM THE VIC 20**

Owen Bishop
0 246 12148 3

**THE VIC 20
GAMES BOOK**

Owen Bishop
0 246 12187 4

The ZX Spectrum

**THE ZX SPECTRUM
And How To Get
The Most From It**

Ian Sinclair
0 246 12018 5

**THE SPECTRUM
PROGRAMMER**

S. M. Gee
0 246 12025 8

**THE SPECTRUM
BOOK OF GAMES**

M. James, S. M. Gee
and K. Ewbank
0 246 12047 9

**INTRODUCING
SPECTRUM
MACHINE CODE**

Ian Sinclair
0 246 12082 7

**SPECTRUM GRAPHICS
AND SOUND**

Steve Money
0 246 12192 0

**THE ZX SPECTRUM
How to Use and
Program**

Ian Sinclair
0 586 06104 5

**Learning is Fun!
40 EDUCATIONAL GAMES
FOR THE SPECTRUM**

Vince Apps
0 246 12233 1

The ZX81

**THE ZX81
How to Use and
Program**

S. M. Gee and
Mike James
0 586 06105 3

Which Computer?

**CHOOSING A
MICROCOMPUTER**

Francis Samish
0 246 12029 0

Languages

**COMPUTER
LANGUAGES AND
THEIR USES**

Garry Marshall
0 246 12022 3

**EXPLORING
FORTH**

Owen Bishop
0 246 12188 2

Machine Code

**Z-80 MACHINE
CODE FOR HUMANS**

Alan Tootill and
David Barrow
0 246 12031 2

**6502 MACHINE
CODE FOR HUMANS**

Alan Tootill and
David Barrow
0 246 12076 2

Using Your Micro

**COMPUTING FOR
THE HOBBYIST AND
SMALL BUSINESS**

A. P. Stephenson
0 246 12023 1

**DATABASES FOR
FUN AND PROFIT**

Nigel Freestone
0 246 12032 0

**SIMPLE INTERFACING
PROJECTS**

Owen Bishop
0 246 12026 6

**INSIDE YOUR
COMPUTER**

Ian Sinclair
0 246 12235 8

Programming

**THE COMPLETE
PROGRAMMER**

Mike James
0 246 12015 0

**PROGRAMMING
WITH GRAPHICS**

Garry Marshall
0 246 12021 5

Word Processing

**CHOOSING A
WORD PROCESSOR**

Francis Samish
0 246 12347 8

**WORD PROCESSING
FOR BEGINNERS**

Susan Curran
0 246 12353 2

MORE FIRE-POWER TO YOUR DRAGON!



Now you have a Dragon you will want to improve your programming knowledge as soon as possible. This book takes you in easy stages through your first steps in programming until you achieve a good level of competence and are well on the way to becoming a maestro.

Many examples are provided to help you and most chapters contain at least one complete program listing – usually a game – to enjoy as you learn. The book is clearly and logically written and sets out all you need to know. It will be invaluable to all Dragon users.

The Author

S M Gee has written several other books and has contributed frequently to *Computing Today*.

More books for Dragon users from Granada

THE DRAGON 32 and how to make the most of it

Ian Sinclair

The Dragon-tamer's handbook!

0 246 12114 9

THE DRAGON 32 BOOK OF GAMES

Mike James, S M Gee and Kay Ewbank

21 superb, crash-proofed games
for you to enjoy!

0 246 12102 5

DRAGON GRAPHICS AND SOUND

Steve Money

How to create more exciting
graphics and sound effects!

0 246 12147 5

Front cover photograph shows the Dragon 32 Home Computer.

GRANADA PUBLISHING

Printed in Great Britain

0 246 12133 5

£5.95 net